FUN WITH PROGRAMMING LANGUAGES

by

Paul J. Kaiser

Much of the artwork that appears
at the beginning of each chapter
may be found at *classicprogrammerpaintings.com*.

All of the quotations that appear
at the end of each chapter
may be found at *www.AZQUOTES.com*.

First Paperback Edition 2021
ISBN: 978-0-578-81126-0

Printed in the United States of America

Please visit my personal website at:
http://www.cs.lewisu.edu/ kaiserpa/

Contents

Dedication

To Margaret and Adelbert,
to Helen and Anthony,

to Carolyn,

to Michelle and Kevin,
to Zoe, Aidan, Kamron, and McKenna,

but especially to Kathleen.

Prologue

There are two obvious questions when someone picks up a new book. *Who is this book intended for?* And, *Why did the author write it?*

Who is this book intended for?

Anybody who finds programming fun, rewarding, or challenging. Anybody who has some experience with at least one programming language. Anybody who has an interest in what goes on "under the hood" when it pertains to computers.

You could be a teenager, a college student, a young adult, or a senior citizen.

As a professor of mathematics and computer science at Lewis University, I was always preaching *life-long learning*. When I retired from teaching I found myself intrigued by several topics – Alan Turing and the foundations of computing; programming languages and how they are implemented.

So I taught a few courses as a part-time instructor at Lewis that gradually morphed through several incarnations:

- from introduction to compiler construction
  using X86_64
- to introduction to compiler construction
  using AARCH64

- to implementation of a RISC assembly language interpreter
- to implementation of a functional programming language interpreter

Why did I write it?

This book encapsulates much of my most recent journey through mathematics and computer science. And, for better or worse, I share it with you to possibly help on your journey.

I was blessed with so many great mentors along my journey!

- Br. Gilbert Gensler, Saint Joseph High School (Cleveland)
- Dr. Richard Otter, University of Notre Dame
- Dr. Lamberto Cesari, University of Michigan
- Sr. Noel Dreska, Lewis University
- Dr. Phil Hogan, Lewis University
- Dr. Thomas Christopher, Illinois Institute of Technology
- Dr. Bob Goldberg, Illinois Institute of Technology
- Dr. Phil Hatcher, Illinois Institute of Technology
- Br. Thomas Dupré, Lewis University
- Ms. Margaret Juraco, Lewis University
- Dr. Ray Klump, Lewis University

All of these individuals were great teachers and supremely talented – each in his or her own way. Some were brilliant intellectually; some were great explainers; some were young and enthusiastic; some were older and wiser.

Me – I am just a retired teacher with a weird sense of humor who incorporated a little bit from each of the individuals above and combined it into my own teaching style

I thoroughly enjoyed my journey exploring various aspects of programming languages these past few years. I hope this book may prove useful to you.

Please feel free to share this book with others – either digitally or as a hard copy. My only request is that if you find this book beneficial to your progress that you make a $50.00 contribution to the Lewis University Computer and Mathematical Sciences (CAMS) Endowment Fund.

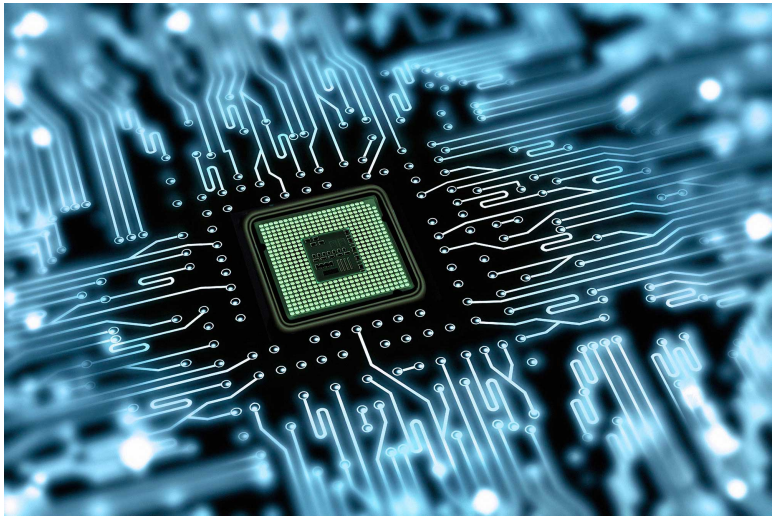
All of the source code files are available for download from:

www.cs.lewisu.edu/~kaiserpa/

Lewis University
One University Parkway
Romeoville, Illinois 60446
(815) 838-0500

Chapter 1

Computer Organization



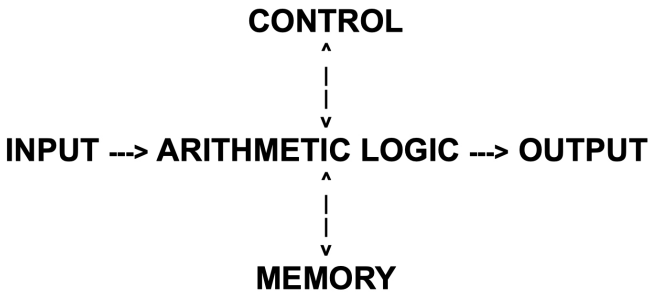
1.1 von Neumann Architecture

Our first chapter will focus on the von Neumann architecture and its implementation in hardware. The basic organization of a computing system is divided into five units:

- MEMORY: This physical device provides sequential locations capable of storage and retrieval of data represented in binary format. Memory locations are accessible by an address, 0 ... MAXMEM - 1.
- ARITHMETIC LOGIC UNIT: This unit is comprised of a finite set of registers (accumulators) which are specially designed to implement mathematical calculations and comparisons. There are typically two categories of registers: integer registers (which utilize simple binary representations) and floating point registers (which utilize more complicated IEEE scientific notation).
- CONTROL UNIT: This unit oversees and coordinates the activities of the other four! Minimally, it must maintain information about the current state of the system: the next statement to execute (i.e., the program counter) and important status indicators (i.e., status flags, such as sign, overflow, zero value, ...).
- INPUT: This physical device enables input information from the outside world to the computing system.
- OUTPUT: This physical device enables output information from the computing system to the outside world.

The Control Unit and the Arithmetic Logic Unit are obviously an inseparable pair; they are collectively referred to as the Central Processing Unit (CPU).

A simple diagram of the von Neumann Architecture follows:



The fundamental building block for information within a computing system is the **binary digit** (or **bit**) which is either a 0 or a 1. Obviously, a single bit is not very informative, but groups of bits quickly become useful. The smallest useful chunk of data is a collection of **eight bits**, called a **byte**. With a single byte we can represent 256 distinct values:

- positive integers from 0 up to and including 255
- positive and negative integers between -127 and +127
- the ASCII character set, including keyboard control characters

But a computing system built around a simple byte of data is quite limited in capability. As computers continue to evolve, expectations have increased regarding both the speed of execution and the size of data manipulated in a single instruction. 8-bit bytes gave way to 16-bit words, which gave way to 32-bit double words, which in turn gave way to 64-bit quad words.

Computing systems have been manufactured and distributed based on each of these data-size units. Each computing system requires its own basic machine language to encode its operations. And each machine language has its own mnemonic shorthand to eliminate the necessity to write 32-bit or 64-bit patterns of 0's and 1's.

These very low level languages have been categorized into one of two types: Complex Instruction Set Computers (CISC) or Restricted Instruction Set Computers (RISC).

1.2 CISC vs RISC

In a most simplified description, a CISC machine will provide you with every imaginable instruction that may help you encode, while a RISC machine limits your options to a smaller instruction set.

Consider the following simple example of multiplying two integer values.

integer multiplication

Let us assume our basic machine has two registers (A and B) and 16 memory cells (with addresses 0 ... 15).

simplified CISC example:

MUL =4 =7

In one CISC assembly language instruction above, the computer will fetch data from address 4, fetch data from address 7, multiply the two values in appropriate registers, and finally store the desired result at address 4.

This calculation would be comparable to the simple C statement:

A = A * B;

simplified RISC example:

The sequence of RISC assembly language instructions below is comparable to the single instruction above! However, each step in the process – the LOADs, the MUL, and the STORE – are all specified individually.

LOAD A =4
LOAD B =7
MUL A B
STORE A =4

A brief comparison of the two styles follows:

CISC	**RISC**
large size instruction set	limited size instruction set
emphasis on hardware concerns	emphasis on software concerns
instructions may vary in length	instructions more uniform in length
instruction execution times vary	instruction execution times uniform
smaller code size	larger code size
register contents cleared	register contents remain

Observe that the execution time for a program is determined by the following formula:

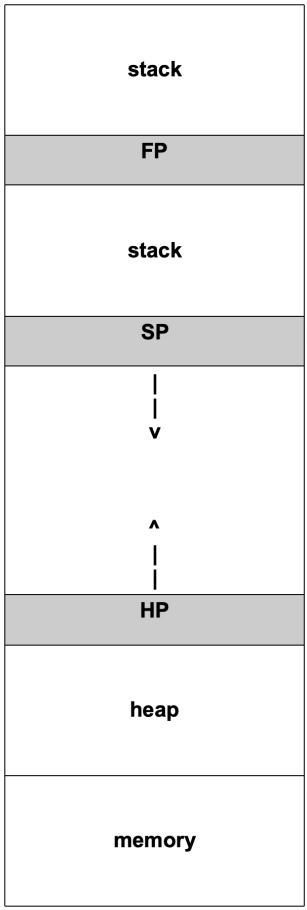
execution time =
(# cycles / instruction) x (# instructions / program)

RISC machines focus on the first ratio and seek to minimize it; CISC machines focus on the second ratio and seek to minimize it! Both perspectives are attempting to improve performance of the underlying computing systems, but using different approaches.

A modern computing system which is an example of a RISC machine would be ARM AARCH64 and a modern computing system which is an example of a CISC machine would be INTEL X86_64.

1.3 Organization of Memory

The following simple diagram summarizes a very traditional utilization of memory for data storage during execution of a program.



Starting at the bottom of the diagram, **memory** for static storage is defined. That is, space is allocated for each variable and an initial data value may be assigned to that location. This memory remains in existence during the entirety of program execution. As memory requirements are specified, addresses are reserved in *increasing* order moving upward in the diagram.

Jumping to the very top of the diagram, a **stack** is available for temporary storage of items. Memory cells within the stack may be inserted and retrieved using standard **push** and **pop** operations. However, it is important to note that the stack top is at the *lowest address* within the stack and that addresses are reserved in *decreasing* order moving downward in the diagram.

The stack area is actually used for two distinct purposes: the first is for holding **temporary results** in mathematical calculations and comparisons, and the second is for implementing a programming language **activation stack** containing activation records. An activation record contains information pertaining to subalgorithms (i.e., subprograms) – its actual arguments, its local variables, its return address, and anything else necessary to correctly return to the subalgorithm **caller**.

In the diagram above, the **stack pointer** is a register, designated SP, which contains the address for the top of the stack. **push** and **pop** operations obviously ***decrease*** and ***increase*** the address contained within the register.

An **activation frame** is a region within memory which contains information pertaining to a specific subalgorithm. An activation frame is anchored by its base address or **frame pointer**, a register designated FP.

Every time a subalgorithm is **called** a new activation frame must be created; every time a subalgorithm **return**s its activation frame must be released. Rather than create two separate stacks within memory, it is simpler to interleave both stacks!

Typically an activation record is created by first allowing for actual arguments, return information, and updating the frame pointer; then the activation record will provide storage for local variables and finally updates the stack pointer. The stack is now available

one again for storing temporary values. This pattern will be repeated for every subalgorithm call.

When a subalgorithm returns, the temporary results should have already been entirely removed from the stack; the previous SP address and the previous FP address need to be recovered and restored; and the caller algorithm can continue with its execution precisely where it left off.

Returning back to the bottom of the diagram, a **heap** is often available for dynamic memory management. The precise operation of a heap depends on the specific memory management techniques chosen for the implementation of that heap. But a simplistic description is when a program needs some dynamic memory, it simply asks for it; and when a program is done with it, it simply returns it. Like a stack, the heap may grow or shrink in size; unlike a stack, the heap is not a last-in, first-out (LIFO) structure.

While the system stack grows in a downward direction in higher memory, the heap structure grow in an upward direction in lower memory. Regardless of the specific implementation of the heap, a **heap pointer**, denoted HP, is necessary to access it.

There is one last register that I want to highlight. The return address for a subalgorithm is a significant piece of information. If it is lost, then the program can not return and resume execution where it left off. Having a designated register (RP), called the **return address pointer**, is a step in the right direction. But it is not a panacea! Every time we call another algorithm or return from the current algorithm, we will be changing the stack pointer, the frame pointer, and the return address pointer.

We need to safeguard these key addresses during algorithm execution. They need to be the same addresses at the end of execution that they were at the beginning. So, where do we put them to keep them safe? The answer is simple! Use the stack!

The first thing a subalgorithm should do is:

- PUSH FP
- PUSH RP
- MOV FP SP

And the last thing a subalgorithm should do is:

- MOV SP FP
- POP RP
- POP FP

1.4 The KBOX

It is at this point that I would like to describe an idealized 64-bit computer which I have chosen to call the KBOX. It falls in the category of being a RISC machine.

The arithmetic logic unit is comprised of 32 64-bit registers.

- Sixteen of the registers will be integer registers, denoted I_0, I_1, ... , I_{15}.
- Sixteen of the registers will be floating point registers, denoted F_0, F_1, ... , F_{15}.

The control unit will contain the program counter (PC) together with the status indicators (FLAGS). All access to the control unit information is indirect through assembly language instructions. The programmer has no direct ability to set the status flags.

The memory unit will also be based on 64-bit chunks. Since everything about the KBOX is built around these 64-bit chunks, we will refer to them as **klunks**.

I believe a novice in assembly language should focus on the big picture initially and slowly fill in the detailed exceptions over time. For the present, let us ignore the fact that actual information can be bytes, words, double words, and quad words; let us also ignore the fact that information is typically retrievable at byte addresses; and lastly, let us ignore the fact that there are alignment issues regarding key addresses (e.g., the stack pointer). Right now our idealized perspective is totally comprised of only **klunks**.

The idealized KBOX instructions need only be 32-bit! We are not going to be doing anything directly with bit patterns that represent instructions, but we should visualize the "typical" instruction as having four elements.

The typical instruction format will be an opcode, followed by the destination register for the result, followed by up to two operand registers. Each item occupies one bytes in the instruction. Other configurations for instructions are possible as well, including "immediate" instructions which include an actual data value within the instruction.

Remember, all this is theoretical! We are describing a figment of my imagination. So, rather than follow Alice deeper into the rabbit hole, let us just stop here.

1.4.1 Important Registers

Assembly languages typically try to anticipate the needs and practices of the programmer. As a result they may pre-assign certain registers to certain functions; or they may integrate certain helpful instructions into others.

From our discussion regarding memory organization in the previous chapter, four items stood out as being important enough and useful enough to suggest some special designation.

The **stack pointer** holds the address to the top of the stack; let us reserve register $\mathbf{I}_{12}$ for the stack pointer **SP**.

The **frame pointer** holds the address to the base of the current activation frame; let us reserve register $\mathbf{I}_{13}$ for the frame pointer **FP**.

The **heap pointer** holds the address to the heap; let us reserve register $\mathbf{I}_{14}$ for the heap pointer **HP**.

The **return pointer** holds the return address for the current activation frame; let us reserve register $\mathbf{I}_{15}$ as the return pointer **RP**.

In KCODE programs, these four important pointer registers may be referenced either by their mnemonic names { SP, FP, HP, RP } or by their integer register designations { I_{12}, I_{13}, I_{14}, I_{15} }.

1.4.2 Basic Data Types

We conclude this overview of the KBOX computer by describing the basic types of data we will be manipulating.

- unsigned integers
- signed integers
- floating point values
- single ASCII characters
- strings of ASCII characters
- locations in memory
- boolean (true/false) values

The first data type is a non-negative integer stored in a single klunk using standard binary representation.

The second data type is a positive or negative integer stored in a single klunk using a standard 2s complement representation (**int64**).

The third data type is a positive or negative real value stored in a single klunk using IEEE standard representation(**flt64**).

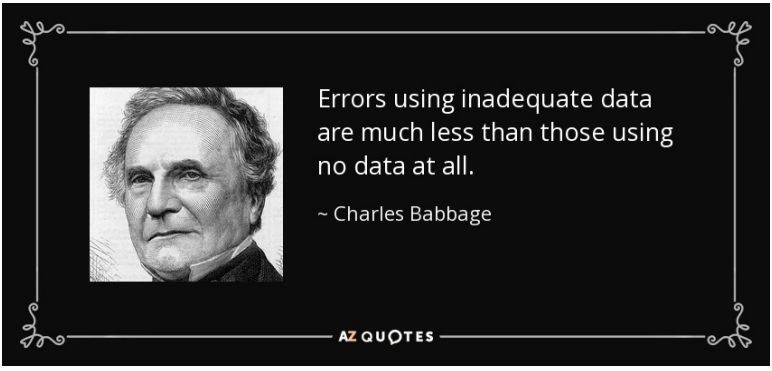
The fourth data type is a single character (using its ASCII representation) stored in the rightmost (lowest) byte of a single klunk (**chr64**).

The fifth data type is a string of characters (each represented by its ASCII code and terminated by '\0'). This is commonly referred to as a cstring (**str64**). However, the KBOX representation is not the actual cstring but a pointer to the actual cstring! The storage requirements for a string obviously depend on the number of characters in the string. A pointer to a cstring is consistent with our limitation of everything being 64-bit chunks.

The sixth data type is a location (address) in memory; it is represented as an unsigned integer (a klunk).

And the last data type is a boolean value; it is represented as an unsigned integer (a klunk) with zero representing **false** and any non-zero value representing **true**.

Our assembly language will supports basic bit manipulations, but all such bit patterns will be represented as unsigned integers (klunks).


Errors using inadequate data are much less than those using no data at all.
~ Charles Babbage
AZ QUOTES

Chapter 2

KCODE

2.1 Program Organization

A KCODE program is organized into two segments: a **Data Segment** and a **Code Segment**.

The Data Segment is used to define static variables in memory. A DEFINE directive is used to define and initialize variables; a RESERVE directive is used to define storage requirements for variables but with no initialization.

The Code Segment is used to define the assembly language instructions to be executed. Two important directives should appear at the very beginning of this section: EXTERN and GLOBAL. Subsequent elements must be either a LABEL directive to define a specific location within the executable code or an actual KCODE instruction.

2.1.1 The Data Segment

_DATA_SEGMENT

_DEFINE *lbl imm*$_{64}$
lbl is a variable name
imm$_{64}$ is the initial value
e.g., 123, -42.95, 'c', "paul kaiser"
i.e., integer, floating point, character, string

_RESERVE *lbl size*
lbl is a variable name
size is the number of klunks to reserve

2.1.2 The Code Segment

_CODE_SEGMENT

_EXTERN *lbl*

_GLOBAL *lbl*

The first directive, EXTERN, identifies externally defined elements that are necessary within this KCODE program.

The second directive, GLOBAL, is the reverse of EXTERN – it announces an internally defined element to others. The name of the main procedure (***main***) requires the use of this directive. Both of these directives are essentially cosmetic! They are included primarily to make KCODE consistent with other reality-based assembly languages!

A third directive, LABEL, is one limitation specific to KCODE! It is necessary to define labels within the executable code of the Code Segment. Of the three directives associated with the Code Segment, the LABEL directive is the only one which may appear among the actual executable instructions.

_LABEL lbl

For readability, I would recommend the following order within your KCODE program:

_DATA_SEGMENT

- _DEFINE directives
- _RESERVE directives

_CODE_SEGMENT

- _EXTERN directives
- _GLOBAL directives
- body of executable instructions
  including any _LABEL directives

2.2 Data Movement / Data Conversion

The following sections of this chapter will focus on describing the instruction set for the assembly language associated with our theoretical KBOX computer. I chose to call this assembly language KCODE – because I probably would have been sued if I had selected "**Special K**" which was my first choice!

For this and subsequent pages, the following shorthand notation will be used:

- **Ireg** represents an integer register: $I_0, \ldots, I_{15}$
- **Freg** represents a floating point register: $F_0, \ldots, F_{15}$
- **reg** represents an arbitrary register of either flavor
- **imm** represents an immediate value
  - integer: 123
  - floating point: -23.56
  - hexadecimal value: 0X3F
  - char value: 'x' (note: use of single quotes)
  - string: "paul" (note: use of double quotes)
  - label: A

This section will highlight the fundamentals of data movement and data conversion within the KBOX.

data movement

One of the characteristics of RISC machines is that data from memory *must be* ***loaded*** to a register before manipulation and data within a register *must be* ***stored*** to memory to save the result. This requires moving the appropriate ***address*** into another register before the actual data transfer.

LDA ***Ireg =lbl***
Ireg ← address (lbl)

LDR ***reg Ireg***
reg ← [Ireg]
data at address in Ireg moved to reg

STR ***reg Ireg***
reg → [Ireg]
data in reg moved to address in Ireg

(note: operand_1 is the source and **not** the destination!)

A second aspect is flexible movement of data between the registers. Movement from one register to another is done without any interpretation or modification of the bit pattern being transferred. There is no promotion from signed integer (Ireg) to floating point (Freg); there is no demotion from floating point (Freg) to signed integer (Ireg). Data conversions will be discussed shortly.

MOV ***reg***$_a$ ***reg***$_b$
$\text{reg}_a \leftarrow \text{reg}_b$

In the previous chapter, we briefly discussed using a portion of memory to implement a system stack to store temporary results and to provide a structure for subalgorithms. KCODE is no exception.

PUSH ***reg***
: top of stack ← reg

POP ***reg***
: reg ← top of stack

data conversion

Explicit promotion and demotion requires two additional instructions specific to the task.

I2F ***Freg Ireg***
: Freg ← Ireg

F2I ***Ireg Freg***
: Ireg ← Freg

2.3 Arithmetic Instructions

We will consider three basic categories of arithmetic instructions: signed (two's complement) arithmetic, unsigned arithmetic, and floating point (IEEE) arithmetic

integer arithmetic

ADD $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b + \text{Ireg}_c$

SUB $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ - Ireg_c

MUL $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ * Ireg_c

DIV $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ / Ireg_c

MOD $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ % Ireg_c

NEG ***Ireg***
Ireg $\leftarrow$ - Ireg

unsigned integer arithmetic

UADD $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b + \text{Ireg}_c$

USUB $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ - Ireg_c

UMUL $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ * Ireg_c

UDIV $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ / Ireg_c

UMOD $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ % Ireg_c

(note: there is no UNEG! think about it!)

floating point arithmetic

FADD ***Freg***$_a$ ***Freg***$_b$ ***Freg***$_c$
Freg$_a$ ← Freg$_b$ + Freg$_c$

FSUB ***Freg***$_a$ ***Freg***$_b$ ***Freg***$_c$
Freg$_a$ ← Freg$_b$ - Freg$_c$

FMUL ***Freg***$_a$ ***Freg***$_b$ ***Freg***$_c$
Freg$_a$ ← Freg$_b$ * Freg$_c$

FDIV ***Freg***$_a$ ***Freg***$_b$ ***Freg***$_c$
Freg$_a$ ← Freg$_b$ / Freg$_c$

FNEG ***Freg***
Freg ← - Freg

2.4 Logical Operations

There are two categories here: bitwise logic (C operators & | $\sim$ $\wedge$) and quadword boolean logic (C operators && || !)

The first set of instructions considers each individual bit within the 64-bit entries; the second set of instructions considers the aggregate bit pattern.

Remember boolean data is:

- false = 0
- true = any non-zero pattern (converted to 1)

bitwise logic operations

BAND $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ & Ireg_c

BOR $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ | Ireg_c

BNOT ***Ireg***
Ireg $\leftarrow \sim$ Ireg

BXOR $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b \wedge \text{Ireg}_c$

quadword boolean logic operations

LAND $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ && Ireg_c

LOR $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b$ || Ireg_c

LNOT ***Ireg***
Ireg $\leftarrow$! Ireg

LXOR $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$ $\boldsymbol{Ireg_c}$
$\text{Ireg}_a \leftarrow \text{Ireg}_b \otimes \text{Ireg}_c$

Note: I have also included a quadword boolean xor symbol – $\otimes$! This is totally my choice of symbol; it is **not** C syntax.

2.5 Comparison / Branching Instructions

unconditional branching instruction

JMP ***lbl***
PC ← lbl
regardless of any status flag being set

comparison instruction

CMP $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$
signed integer comparison of two values

UCMP $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$
unsigned integer comparison of two values

FCMP $\boldsymbol{Freg_a}$ $\boldsymbol{Freg_b}$
floating point comparison of two values

set appropriate status flags (EQ, LT, GT)
only one status flag will be set ON
the other two will be OFF

conditional branching instructions

A comparison instruction should precede any of the following:

JEQ ***lbl***
: PC ← lbl
if equal (EQ flag on)

JNE ***lbl***
: PC ← lbl
if not equal (EQ flag off)

JGT ***lbl***
: PC ← lbl
if greater than (GT on)

JGE ***lbl***
: PC ← lbl
if greater than or equal (LT flag off)

JLT ***lbl***
: PC ← lbl
if less than (LT on)

JLE ***lbl***
: PC ← lbl
if less than or equal (GT flag off)

2.6 Bitwise Manipulations

Like the bitwise logical instructions, the following instructions operate at the lowest level possible – individual bits in the data representation. These are some of the most elementary operations that are possible. There are three distinct categories of manipulations we will present: simple rotations, logical shifts, and arithmetic shifts.

simple rotations

ROL ***Ireg imm***$_6$
: rotate imm_6 bits to the left ($0 \leq \text{imm}_6 < 64$)
bits wrap-around to right (bit 0)
store result in Ireg

ROR ***Ireg imm***$_6$
: rotate imm_6 bits to the right ($0 \leq \text{imm}_6 < 64$)
bits wrap-around to left (bit 63)
store result in Ireg

logical shifts

SHL ***Ireg imm***$_6$
: shift imm_6 bits to the left ($0 \leq \text{imm}_6 < 64$)
bits simply fall off the end!
store result in Ireg

SHR ***Ireg imm***$_6$
: shift imm_6 bits to the right ($0 \leq \text{imm}_6 < 64$)
bits simply fall off the end!
store result in Ireg

arithmetic shifts

ASHL ***Ireg imm***$_6$
: identical to **SHL** ***Ireg*** **,** $\mathbf{imm}_6$

ASHR ***Ireg imm***$_6$
: shift imm_6 bits to the right ($0 \leq \text{imm}_6 < 64$)
sign bit extended!
store result in Ireg

2.7 Subalgorithms

Two very important instructions appear in this section – **call** and **ret**. The first is used to transfer program control to a subalgorithm; the second is used to return control back to the statement immediately following the **call** instruction.

These two instructions are obviously connected with one another and both must consider the following issue: How exactly does the program remember what the program counter (PC) should be when it executes a **ret** instruction. To complicate details even further, it is important to remember that subalgorithm **call** and **ret** may be nested within one another! So we are in reality dealing with a potential history of return locations that must be remembered. Some assembly languages may attempt to help in some little way; others may not.

KCODE puts the return address in register $\mathbf{RP} = \mathbf{I}_{15}$ when executing the **call** instruction; the **ret** instruction will assume that the return address may still be found in register $\mathbf{RP} = \mathbf{I}_{15}$. That is a very big assumption!

CALL *lbl*
    $\mathbf{RP} \leftarrow \mathbf{PC}$
    $\mathbf{PC} \leftarrow$ *lbl*

RET
    $\mathbf{PC} \leftarrow \mathbf{RP}$

It is important to note here that **call** and **ret** are relatively basic KCODE instructions. Setting up and dismantling the activation record is the responsibility of the programmer!

2.8 Miscellaneous Instructions

The following instructions are also available within KCODE. They are included to complete a full spectrum of assembly language instructions, including basic input and output commands and a most simplistic version of dynamic memory allocation and de-allocation

The first instruction is the common **no-op** instruction.

NOP

This instruction does absolutely nothing, but takes up space!

The second instruction is the common **halt** instruction.

HALT

This instruction terminates the KCODE program execution.

The next two instructions are very useful tools in conjunction with the integer registers:

INC ***Ireg***
: the current contents of Ireg is incremented by one

DEC ***Ireg***
: the current contents of Ireg is decremented by one

A programming language is much friendlier if it provides some simple mechanisms for input and output. The following four instructions are provided exactly for that purpose.

GET ***fmt***
data entered from the keyboard
is saved to the top of the stack
fmt references the data type being entered:
INT, FLT, CHR, STR

GETLN
read up to and including the next carriage return

PUT ***fmt***
the data at the top of the stack
is displayed to the monitor
fmt references the data type being displayed:
INT, FLT, CHR, STR, HEX, PTR

PUTLN
write a single carriage return

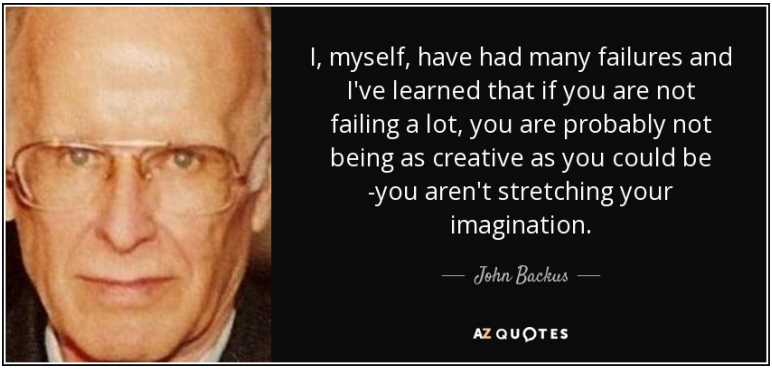
At some point you may want to experiment with dynamic allocation of memory and the **heap**. The following two instructions provide the bare bones necessities.

MALLOC $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$
allocate Ireg_b 64-bit klunks
$\text{Ireg}_a \leftarrow$ base address

DALLOC $\boldsymbol{Ireg_a}$ $\boldsymbol{Ireg_b}$
de-allocate Ireg_b 64-bit klunks
found at base address Ireg_a

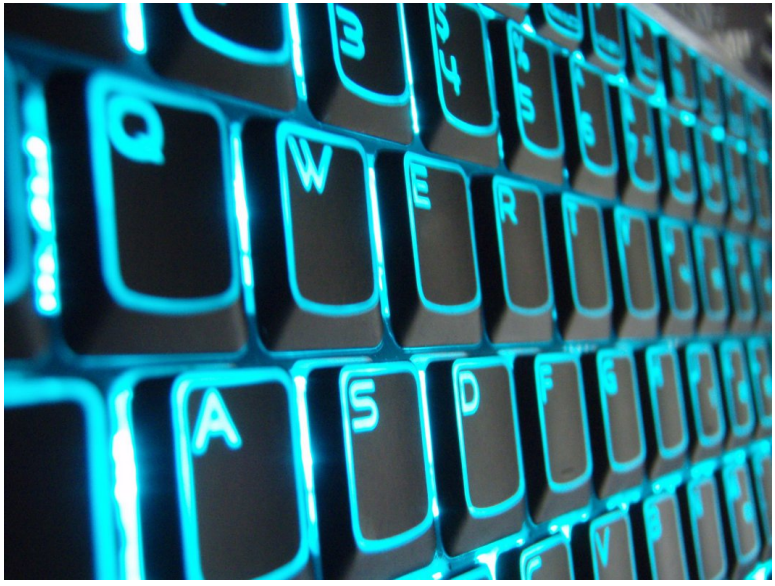
The above two instructions are most minimal.
- MALLOC allocates heap memory until it is exhausted;
- DALLOC does nothing!

However, both could be properly implemented sometime in the future.


I, myself, have had many failures and I've learned that if you are not failing a lot, you are probably not being as creative as you could be -you aren't stretching your imagination.
— John Backus —
AZ QUOTES

Chapter 3

Sample Programs



The following pages contain two straightforward demonstrations of the KCODE assembly language.

The first example is an often assigned programming problem to solve any quadratic equation, given the three real coefficients. I chose this as an illustrative example because it includes a little bit of everything! It has alculations, comparisons, storage and retrieval, the use of the stack for temporary values, static memory addressing, offset addressing using the frame pointer, call and return.

The second example is a simple forward linked list representation for an integer stack structure. I chose this as an illustrative example because it introduces aggregation of data into a record structure (an integer data value together with an address for next value). In addition, I always find pointer manipulations a thought provoking and challenging experience. At the assembly language level, many of the concepts seem to me to be easier to comprehend.

3.1 KCODE: discriminant.k

Sample KCODE: discriminant.k

```
# discriminant.k

    _DATA_SEGMENT

    _RESERVE        A       1
    _RESERVE        B       1
    _RESERVE        C       1
    _RESERVE        DISC    1
    _RESERVE        ROOT1   1
    _RESERVE        ROOT2   1

    _DEFINE         PROMPT          "enter three real values: "
    _DEFINE         ERROR           "first coefficient must not be zero!"
    _DEFINE         DISCMESSAGE     "the discriminant is: "
    _DEFINE         MESSAGE1        "the first  real root is: "
    _DEFINE         MESSAGE2        "the second real root is: "
    _DEFINE         MESSAGE3        "there are no real roots!"
    _DEFINE         EPSILON         0.0000001

    _CODE_SEGMENT

    _GLOBAL         MAIN

# MAIN procedure prologue

    _LABEL          MAIN
    PUSH            RP
    PUSH            FP
    MOV             FP SP

# get the first coefficient
# make sure A non-zero (i.e.  linear equation)

    LDA             SAR =PROMPT
    LDR             IA SAR
    PUSH            IA
    PUT             STR
    GET             FLT
    LDA             DAR =A
    POP             FA
    STR             FA DAR
    MOVI            FB 0.0
    FCMP            FA FB
    JNE             CONTINUE
    LDA             SAR =ERROR
    LDR             IA SAR
    PUSH            IA
    PUT             STR
```

```
    PUTLN
    HALT

# get the second and third coefficient

    _LABEL      CONTINUE
    GET         FLT
    LDA         DAR =B
    POP         FA
    STR         FA DAR
    GET         FLT
    LDA         DAR =C
    POP         FA
    STR         FA DAR

# calculate the discriminant b*b - 4.0*a*c
# note the use of the stack for holding temporary values

    LDA         SAR =B
    LDR         FA SAR
    PUSH        FA
    LDA         SAR =B
    LDR         FA SAR
    PUSH        FA
    POP         FC
    POP         FB
    FMUL        FA FB FC
    PUSH        FA
    LDA         SAR =A
    LDR         FA SAR
    PUSH        FA
    LDA         SAR =C
    LDR         FA SAR
    PUSH        FA
    POP         FC
    POP         FB
    FMUL        FA FB FC
    PUSH        FA
    MOVI        FA 4.0
    PUSH        FA
    POP         FC
    POP         FB
    FMUL        FA FB FC
    PUSH        FA
    POP         FC
    POP         FB
    FSUB        FA FB FC
    PUSH        FA
    LDA         DAR =DISC
    POP         FA
    STR         FA DAR

# display discriminant value
```

```
    LDA         SAR =DISCMESSAGE
    LDR         IA SAR
    PUSH        IA
    PUT         STR
    LDA         SAR =DISC
    LDR         FA SAR
    PUSH        FA
    PUT         FLT
    PUTLN

    LDA         SAR =DISC
    LDR         FB SAR
    MOVI        FC 0.0
    FCMP        FB FC
    JGT         TWOREAL
    JLT         TWOCOMPLEX
    JMP         ONEREAL

# two real roots

    _LABEL      TWOREAL
    LDA         SAR =B
    LDR         FA SAR
    FNEG        FA
    PUSH        FA
    LDA         SAR =DISC
    LDR         FA SAR
    PUSH        FA      # actual arg to stack
    CALL        sqrt
    INC         SP      # delete actual arg from stack
    PUSH        FA      # retrieve sqrt value
    POP         FC
    POP         FB
    FSUB        FA FB FC
    PUSH        FA
    FADD        FA FB FC
    PUSH        FA
    LDA         SAR =A
    LDR         FB SAR
    MOVI        FC 2.0
    FMUL        FA FB FC
    PUSH        FA
    POP         FD
    POP         FB
    POP         FC
    FDIV        FA FB FD
    PUSH        FA
    FDIV        FA FC FD
    PUSH        FA
    LDA         DAR =ROOT1
    POP         FA
    STR         FA DAR
```

```
    LDA             DAR =ROOT2
    POP             FA
    STR             FA DAR
    JMP             DONE

# one real root (repeated root!)

    _LABEL          ONEREAL
    LDA             SAR =B
    LDR             FA SAR
    FNEG            FA
    PUSH            FA
    LDA             SAR =A
    LDR             FB SAR
    MOVI            FC 2.0
    FMUL            FA FB FC
    PUSH            FA
    POP             FC
    POP             FB
    FDIV            FA FB FC
    PUSH            FA
    PUSH            FA
    LDA             DAR =ROOT1
    POP             FA
    STR             FA DAR
    LDA             DAR =ROOT2
    POP             FA
    STR             FA DAR
    JMP             DONE

# two complex conjugate roots

    _LABEL          TWOCOMPLEX
    LDA             SAR =MESSAGE3
    LDR             IA SAR
    PUSH            IA
    PUT             STR
    PUTLN
    JMP             EXITMAIN

# display roots

    _LABEL          DONE
    LDA             SAR =MESSAGE1
    LDR             IA SAR
    PUSH            IA
    PUT             STR
    LDA             SAR =ROOT1
    LDR             FA SAR
    PUSH            FA
    PUT             FLT
    PUTLN
    LDA             SAR =MESSAGE2
```

```
    LDR             IA SAR
    PUSH            IA
    PUT             STR
    LDA             SAR =ROOT2
    LDR             FA SAR
    PUSH            FA
    PUT             FLT
    PUTLN

# MAIN procedure epilogue

    _label          EXITMAIN
    MOV             SP FP
    POP             FP
    POP             RP
    MOV             IA ZR
    RET

# sqrt procedure prologue

    _LABEL          SQRT
    PUSH            RP
    PUSH            FP
    MOV             FP SP
    DEC             SP              # test variable T
    DEC             SP              # new test variable Tnew

# SET Tnew = 1.0

    MOVI            OR -2           # offset for Tnew
    ADD             SAR FP OR
    MOVI            FA 1.0
    STR             FA SAR

    _LABEL          LOOP
    MOVI            OR -2           # offset for Tnew
    ADD             SAR FP OR
    LDR             FA SAR
    PUSH            FA
    MOVI            OR -1           # offset for T
    ADD             DAR FP OR
    POP             FA
    STR             FA DAR

# SET Tnew = (T + S/T)/2.0

    MOVI            OR -1           # offset for T
    ADD             SAR FP OR
    LDR             FA SAR
    PUSH            FA
    MOVI            OR 2            # offset for S
    ADD             SAR FP OR
    LDR             FA SAR
```

```
    PUSH        FA
    MOVI        OR -1           # offset for T
    ADD         SAR FP OR
    LDR         FA SAR
    PUSH        FA
    POP         FC
    POP         FB
    FDIV        FA FB FC
    PUSH        FA
    POP         FC
    POP         FB
    FADD        FA FB FC
    PUSH        FA
    MOVI        FA 2.0
    PUSH        FA
    POP         FC
    POP         FB
    FDIV        FA FB FC
    PUSH        FA
    MOVI        OR -2           # offset for Tnew
    ADD         SAR FP OR
    POP         FA
    STR         FA SAR

# calculate Tnew - T

    MOVI        OR -1           # offset for T
    ADD         SAR FP OR
    LDR         FA SAR
    PUSH        FA
    MOVI        OR -2           # offset for Tnew
    ADD         SAR FP OR
    LDR         FA SAR
    PUSH        FA
    POP         FC
    POP         FB
    FSUB        FA FB FC
    PUSH        FA

# calculate | Tnew - T |

    POP         FA
    MOVI        FB 0.0
    FCMP        FA FB
    JGE         pos
    FNEG        FA
    _LABEL      pos
    PUSH        FA

    POP         FA
    LDA         SAR =EPSILON
    LDR         FB SAR
    FCMP        FA FB
```

```
    JGE             LOOP

# return Tnew

    MOVI            OR -2               # offset for Tnew
    ADD             SAR FP OR
    LDR             FA SAR

# sqrt procedure epilogue

    _LABEL          EXITSQRT
    MOV             SP FP
    POP             FP
    POP             RP
    RET
```

Output from the assembly language program is as expected:

OUTPUT

```
$> kbox discriminant.k

enter three real values: 1.0
0.0
-4.0
the discriminant is: 16.000000
the first  real root is: -2.000000
the second real root is: 2.000000

$> kbox discriminant.k

enter three real values: 1.0
-6.0
9.0
the discriminant is: 0.000000
the first  real root is: 3.000000
the second real root is: 3.000000

$> kbox discriminant.k

enter three real values: 1.0
0.0
5.0
the discriminant is: -20.000000
there are no real roots!

$>
```

3.2 KCODE: linkedlist.k

Sample KCODE: linkedlist.k

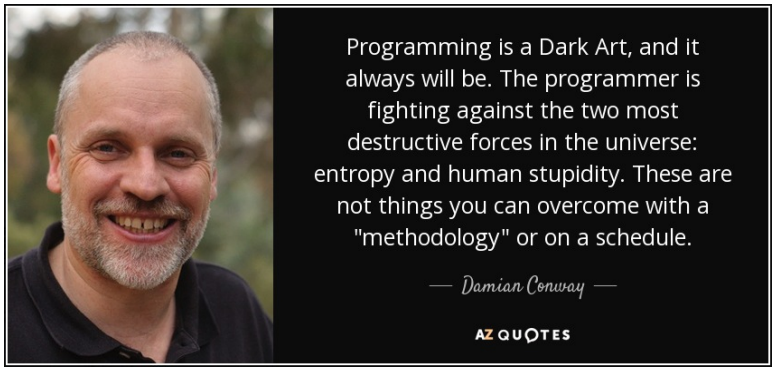
```
# linkedlist.k

    _DATA_SEGMENT

    _DEFINE     PROMPT  "enter a positive integer value: "
    _DEFINE     MESSAGE "your numbers in reverse order are:"
    _DEFINE     HEADER  0

    _CODE_SEGMENT

    _GLOBAL     MAIN

# main procedure prologue

    _LABEL      MAIN
    PUSH        RP
    PUSH        FP
    MOV         FP SP

# build a linked list

    _LABEL      LOOP1
    LDA         SAR =PROMPT
    LDR         IA SAR
    PUSH        IA
    PUT         STR
    GET         INT
    POP         IA
    CMP         IA ZR
    JLT         ENDLOOP1

    PUSH        IA      # save data value
    LDA         SAR =HEADER
    LDR         IA SAR
    PUSH        IA      # save HEADER pointer
    MOVI        IB 2
    MALLOC      DAR IB
    PUSH        DAR     # save node pointer
    LDA         SAR =HEADER
    POP         IA
    STR         IA SAR  # node pointer to HEADER
    MOV         IA DAR
    POP         IA
    STR         IA DAR  # HEADER pointer into node (offset 0)
    INC         DAR
    POP         IA
    STR         IA DAR  # data value into node (offset 1)
    JMP         LOOP1
```

```
    _LABEL      ENDLOOP1

    LDA         SAR =MESSAGE
    LDR         IA SAR
    PUSH        IA
    PUT         STR
    PUTLN

    LDA         SAR =HEADER
    LDR         IA SAR
    PUSH        IA          # save pointer

    _LABEL      LOOP2
    POP         SAR         # get pointer
    CMP         SAR ZR      # is it null?
    JEQ         ENDLOOP2
    PUSH        SAR         # save node pointer
    INC         SAR
    LDR         IA SAR      # get data value
    PUSH        IA
    PUT         INT
    PUTLN
    POP         SAR         # retrieve node pointer
    LDR         IA SAR
    PUSH        IA          # save pointer to next node
    JMP         LOOP2
    _LABEL      ENDLOOP2

# main procedure epilogue

    _LABEL      EXITMAIN
    MOV         SP FP
    POP         FP
    POP         RP
    MOV         IA ZR
    RET
```

Once again, output from the assembly language program is as expected:

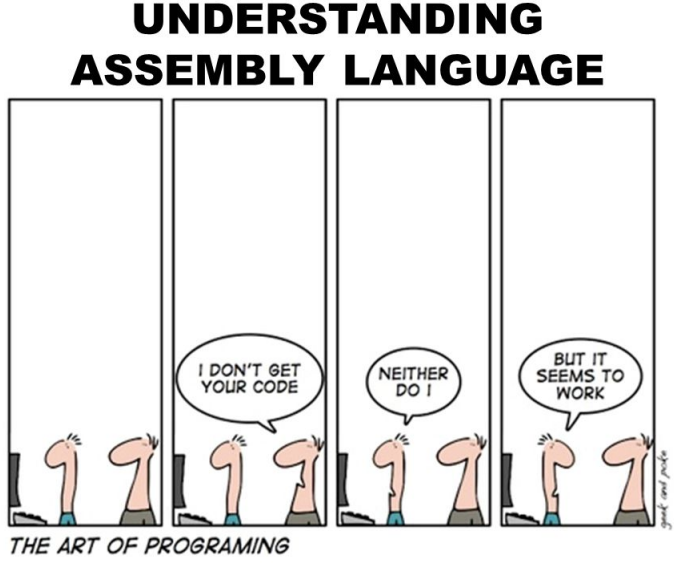
OUTPUT

```
$> kbox linkedlist.k

enter a positive integer value: 9
enter a positive integer value: 2
enter a positive integer value: 6
enter a positive integer value: 4
enter a positive integer value: 23
enter a positive integer value: 11
enter a positive integer value: -1
your numbers in reverse order are:
11
23
4
6
2
9

$>
```


Programming is a Dark Art, and it always will be. The programmer is fighting against the two most destructive forces in the universe: entropy and human stupidity. These are not things you can overcome with a "methodology" or on a schedule.
— Damian Conway —
AZ QUOTES

Chapter 4

Building a Simulator



There are two basic approaches to simulate how a KBOX would execute KCODE: the first is a **compiler** and the second is an **interpreter**.

Typically, a **compiler** will convert the given KCODE source file into another programming language which is available on the user's computer. The resulting program code is then compiled / assembled, linked, and then executed. The resulting executable file may then be run as many times as desired; a second compilation of the original KCODE file is only necessary if changes have been made to that original source file.

An **interpreter**, on the other hand, is written in another programming language which is available on the user's computer. The resulting program is then compiled / assembled and linked and the resulting executable code reads the given KCODE source file and ***simulates*** the behavior of executing the source file on a KBOX computer. The resulting executable file is of the ***simulator*** and **not** of the ***source file*** to be simulated! This has two immediate consequences regarding performance:

- the interpreter must be executed **each time**
  you wish to execute a given program
  not **one time** as would be the case with a compiler
- if the given program has repetition structures
  then the interpreter will **re-translate** these structures
  as often as necessary

We will study the latter of these two techniques in the next sections of the text, which will focus on implementing an interpreter for KCODE written in the programming language C. Such an interpreter should be portable, in theory, to any platform providing a good C compiler.

We will begin a study of compilers and their construction in the next part of the text, where we will discuss **higher level languages** (higher than assembly language) referred to as **procedural languages**.

4.1 Basic Components

Our implementation of a C interpreter for KCODE on a KBOX will be done using several C files. These files are printed in full in the following chapter, **The End Result**.

- the KBOX data types are defined in the two files ***ktype.h*** and ***ktype.c***
- the KBOX memory is defined in the two files ***memory.h*** and ***memory.c***
- the KBOX register set is defined in the two files ***registers.h*** and ***registers.c***
- the KBOX instruction set is defined in the two files ***code.h*** and ***code.c***
- the overall coordination of these four basic components is done in the main program, ***kbox.c***

The compilation of the various pieces that comprise the C simulator is facilitated by a script file: ***buildit***.The following instructions are found within this simple file.

```
#!/bin/bash
gcc -o kbox kbox.c \
     ktype.c memory.c registers.c code.c setup.c fedex.c
```

The latter two files in the sequence above, ***setup.c*** and ***fedex.c***, will be discussed shortly when we present the main program ***kbox.c***.

4.1.1 KBOX Types

One of the first major hurdles that arises in implementing a simulator has to do with data types. Recall the data types in KCODE are:

- int64: 64-bit 2s complement signed integer binary pattern
- flt64: 64-bit IEEE signed floating point binary pattern
- chr64: 8-bit ASCII representation stored in the rightmost (lowest) byte of a 64-bit pattern
- str64: 64-bit pointer to actual string storage
- klunk: basic or vanilla 64-bit binary pattern (unsigned)

Our hurdle is **not** in understanding the various data types and is **not** in understanding the different types of binary patterns. Our hurdle is in **selecting a generic representation** which can on demand switch back and forth between these various interpretations.

In a previous implementation of a KCODE simulator, I chose to represent everything as a C++ string object. Since I also chose to essentially ignore character and string data, I simply used **stoull**, **stod**, and **to_string** for conversion between strings and numeric data. In the current implementation, everything is represented as a 64-bit binary pattern (a klunk) and I rely heavily on C operators **&** and ***** for retrieving either an address or a data value.

However, there are several pitfalls to keep in mind. Compilers for typed languages, like C and C++, often complain at implicit conversions (or recasting) of data values. C++ has four different flavors of **casts**! C has but one.

If **k** is a variable containing a 64-bit pattern, its contents can be interpreted several ways:

as a twos-complement signed integer: ***((int64*) (&k))**
as an IEEE floating point value: ***((flt64*) (&k))**
as a single ASCII character: ***((chr64*) (&k))**
as a C-string sequence of ASCII characters: **(str64) (k)**

First, recast the **address** for **k** as a pointer to the correct data type; and then, **follow** that address to retrieve the data.

For actual promotion / demotion of data type, simple type casts are sufficient:

integer to floating point:	**(flt64) (k)**
floating point to integer:	**(int64) (k)**

Spoiler Alert: If you are attempting to write your own version of a KCODE simulator, refer to the implementation files found in the next chapter only as a last resort! My comments in this subsection provide an overview or an approach to consider; my C header file provides a possible template to build on. But the implementation file gives away "the keys to the kingdom".

4.1.2 KBOX Memory

Recall that KBOX memory is simply a linearly ordered collection of **klunks** with numbered addresses from 0 to a maximum memory size. For simulating memory, we need to define that maximum size (MAXMEM = 1000000). This value is probably fine for testing and debugging initial KCODE programs, but may have to be increased as we move into more serious projects.

Static storage is allocated in low memory, with addresses being assigned in increasing order (0, 1, 2, ...); stack storage is allocated in high memory, with addresses being assigned in decreasing order (MAXMEM - 1, MAXMEM - 2, MAXMEM - 3, ...). Heap storage (which we discuss only in the most superficial terms) is usually allocated above static storage, with heap space also expanding upward in increasing order.

The data that is typically stored in and retrieved from memory are binary bit patterns – it does **not** matter whether the data is integer, or floating point, or character, or whatever. In memory, it is just a bit pattern. For this simulation, the contents of memory must be similarly anonymous but allow for either integer, floating point, character, or string data values. As mentioned in the previous subsection, I chose to represent these values in memory as **klunks**.

In addition to the actual storage space, KBOX will also require a basic symbol table to remember static allocation assignments: identifier, its address in memory, and its data size. Although data size at present will primarily be a single klunk, we will eventually have to consider structured data types built up from atomic data incorporating several klunks. Also, as we allocate static storage, we also need to keep track of the next available address in memory.

Remember, everything in this simulation will be done in 8-byte (64-bit) klunks! PUSH, POP, MALLOC, and DALLOC are also all defined using this fundamental klunk unit.

The C header files for our memory might take the following form:

- memory would be an array of size MAXMEM with each entry being a klunk
- a corresponding memory map containing entries of the form:
  - name
  - address
  - size
- an unsigned integer variable containing the next memory address to allocate
- a method to define new identifier and its initial value
- a method to reserve space for a new identifier having a specified size
- a method to search the memory map for a particular identifier and return its location within the memory map
- a **peek** method which returns the bit pattern found at a specified address
- a **poke** method which stores a given bit pattern at a specified address
- a method to display the bit pattern stored in memory

Spoiler Alert: As I said previously, if you are attempting to write your own version of a KCODE simulator, refer to the implementation files found in the next chapter only as a last resort! My comments in this subsection provide an overview or an approach to consider; my C header file provides a possible template to build on. But the implementation file gives away "the keys to the kingdom".

4.1.3 KBOX Registers

Recall that KBOX provides two register sets: 16 integer registers {I0, . . . , I15} and 16 floating point registers {F0, . . . , F15}.

Moving data from an integer register to a floating point register will perform an *implicit promotion of data*; moving data from a floating point register to an integer register will similarly perform an *implicit demotion of data*.

The C header files for our register set might take the following form:

- Iregs are an array of 16 klunks
- Fregs are an array of 16 klunks

- definitions assigning descriptive names to specific registers (according to their functionality)

- a boolean method to recognize integer register names
- a boolean method to recognize floating point register names

- a method to display the contents of the integer registers
- a method to display the contents of the floating point registers

- a method to convert an integer register name to a usable pointer value within the C program
- a method to convert a floating point register name to a usable pointer value within the C program

Spoiler Alert: As I said previously, . . . aw, you know . . .

4.1.4 KBOX Code

In this section, we are **not** so much talking about **executing** the KCODE instructions as we are talking about **processing** the instructions. How do we store the instructions? How do we organize information? Shortly we will discuss how the interpreter's main program puts all the preceding pieces together in a workable order.

The KCODE instruction are part of the _CODE_SEGMENT which also includes three directives: _EXTERN, _GLOBAL, and _LABEL. Each of these elements must be considered .

_EXTERN items identify entities that will be found in external files that will be linked together with this source code; _GLOBAL items identify entities that will be visible and available to external files that will be linked with this source code.

_EXTERN and _GLOBAL are, in reality, irrelevant in KCODE! We will be simulating stand-alone assembly language which will not be linked to anything else.

_LABEL is a convenience for implementing the simulator. This directive highlights the important locations within the executable code.

A simple representation for a KCODE instruction will be as a four-tuple: instruction, operand1, operand2, and operand3. As we encounter instructions, we create the appropriate four-tuple and add it to the end of our expanding list of instruction and we update an instruction counter (current location within the instruction set).

The C header files for our code list might take the following form:

- labels will be stored as a list of pairs: name and location (i.e., line number)
- instructions will be stores as a list of four-tuples: opcode, operand1, operand2, and operand3
- a complete description of all components in our executable code:
  - the list of globals
  - the list of externs
  - the list of labels
  - the list of instructions
  - the instruction counter
  - the status flags: EQ, GT, and LT
  - a method to define a new global
  - a method to define a new extern
  - a method to define a new label (and its location)
  - a method to define a new instruction
  - a method to search the list of globals
  - a method to search the list of externs
  - a method to search the list of labels
  - a method to display each of the above components
  - a method to fetch an instruction

Spoiler Alert: Enough said!

4.2 KBOX Main Program

The KBOX main program is a two-phase program to process the contents of the KCODE source file.

Phase One is the reading of the assembly language source file in its entirety – to properly recognize and interpret each individual line. This phase of the process is supported and facilitated by two C files: ***setup.h*** and ***setup.c***.

The procedures in setup facilitate opening and closing the source file, reading the current assembly language item in its entirety, and skipping blank lines and comments. And in addition the procedures also assist in the recognition of various assembly language elements: opcodes and operands (including registers, memory addresses, label names, and literal values).

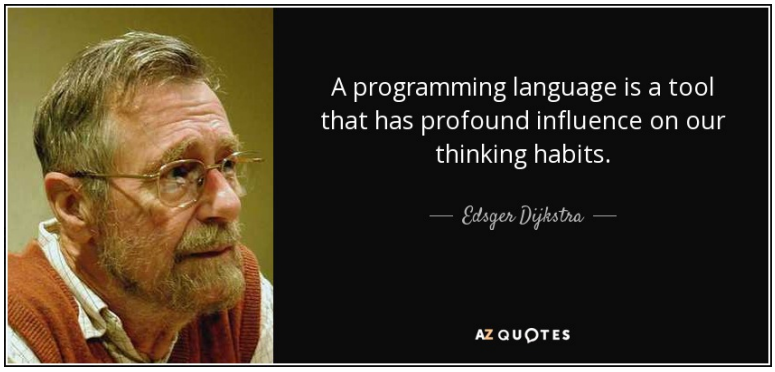
While reading the assembly language source file, Phase One determines the static storage requirements and maintains the ***nextAddress*** location. Upon closing the source file, Phase One properly assigns the initial flag settings, program counter, and key register values.

At this point, Phase Two is ready to begin. We can now simulate execution of the KCODE instruction set to calculate, compare, branch, call and return, malloc and dalloc, and hopefully do something worthwhile. And yet, Phase Two is essentially nothing more than an infinite loop executing a single procedure defined in two files: ***fedex.h*** and ***fedex.c***.

The single procedure I have named FEDEX (a simple anagram for FEtch-DEcode-eXecute). It is nothing more than one humongus selection statement with over 60 possible alternatives to compare! For each opcode, FEDEX will:

- retrieve the appropriate operands
- insure that data interpretation is correct
- simulate proper execution of the opcode
- insure that the program counter references the next instruction

Spoiler Alert: if you are attempting to write your own version of a KCODE simulator, refer to my coding only as a last resort!


A programming language is a tool that has profound influence on our thinking habits.
— Edsger Dijkstra —
AZ QUOTES

Chapter 5

The End Result

5.1 kbox: the main driver

kbox.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "ktypes.h"
#include "registers.h"
#include "memory.h"
#include "code.h"
#include "setup.h"
#include "fedex.h"

/* ------------------------------------------------------------ */

// kbox.c

// KBOX is main driver program for KCODE interpreter.
// It is comprised of two large components:

//    the first component oversees
//        the reading of the source file
//        the setup of memory allocations,
//            register contents, label locations,
//            and listing of instructions

//     the second component is the actual
//        "fetch-decode-execute" cycle

/* ------------------------------------------------------------ */

// The first component is a rather lengthy process which
// requires reading every line in the source code.
// Support for this component may be found in two files
//     setup.h and setup.c.
// Even with much of the work channeled to these helpful
// procedures, reading and processing the source file is
// quite detailed:

//     - is the line of code a directive or an instruction
//     - if it is a directive, is it data or code segment
//     - if it is a label directive,
//         what is its position in the instruction set
//     - if it is an instruction, what are its components

//     - after reading the source file,
//         the important registers must be set
//     - and the user is given the option to view setup
//         prior to execution of the instruction set

/* ------------------------------------------------------------ */
```

```
// The second component appears trivial in comparison!
// The "fetch-decode-execute" cycle is essentially
// an infinite loop.
// The actual procedure that does all the work is found
// in two files
//     fedex.h and fedex.c.
// Furthermore, the deceptively short fragment
// of coding below expands into a huge implementation
// file where the sole procedure single-handedly
// simulates every KCODE instruction!

/* ------------------------------------------------------------ */

int data_definition = 0;
int code_definition = 0;
int data_done = 0;
int code_done = 0;

int main (int argc,char** argv)
{
  if (argc < 2)
  {
    printf ("a source file must be specified!\n");
    exit (EXIT_FAILURE);
  }

// setup code

  initializeMemory ();
  openFile (argv[1]);
  while (!done)
  {
    readLine ();
    if (isBlankLine())
      continue;
    str64 code = getCode ();
    if (strcmp(code,"DATA_SEGMENT") == 0)
    {
      if (code_definition && !code_done)
        code_done = 1;
      if (data_definition || data_done)
      {
        printf ("... redefinition DATA_SEGMENT!\n");
        exit (EXIT_FAILURE);
      }
      data_definition = 1;
    }
    else if (strcmp(code,"DEFINE") == 0)
    {
      if (data_definition && !data_done)
      {
        str64 label = getOperand ();
```

```
      str64 op2 = getOperand ();
      klunk value = literal2klunk(op2);
      _DEFINE (label,value);
    }
    else
      printf ("....move_DEFINE_into_DATA_SEGMENT!\n");
  }
  else if (strcmp(code,"_RESERVE") == 0)
  {
    if (data_definition && !data_done)
    {
      str64 label = getOperand ();
      klunk size = atoll(getOperand());
      _RESERVE (label,size);
    }
    else
      printf ("....move_RESERVE_into_DATA_SEGMENT!\n");
  }

  else if (strcmp(code,"_CODE_SEGMENT") == 0)
  {
    if (data_definition && !data_done)
    data_done = 1;
    if (code_definition || code_done)
    {
      printf ("....redefinition_DATA_SEGMENT!\n");
      exit (EXIT_FAILURE);
    }
    code_definition = 1;
  }
  else if (strcmp(code,"_GLOBAL") == 0)
  {
    if (code_definition && !code_done)
    {
      str64 label = getOperand ();
      _GLOBAL (label);
    }
    else
      printf ("....move_GLOBAL_into_CODE_SEGMENT!\n");
  }
  else if (strcmp(code,"_EXTERN") == 0)
  {
    if (code_definition && !code_done)
    {
      str64 label = getOperand ();
      _EXTERN (label);
    }
    else
      printf ("....move_EXTERN_into_CODE_SEGMENT!\n");
  }
  else if (strcmp(code,"_LABEL") == 0)
  {
    if (code_definition && !code_done)
```

```
    {
      str64 label = getOperand ();
      LABEL (label);
    }
    else
      printf ("\t...\tmove\tLABEL\tinto\tCODE_SEGMENT!\n");
  }

  else if (code[0] == '_')
  {
    printf ("\t...\tinvalid\tdirective:\t%s\n",code);
  }

  else // code is an instruction!
  {
    str64 oper1 = getOperand ();
    str64 oper2 = getOperand ();
    str64 oper3 = getOperand ();
    addInstruction (code,oper1,oper2,oper3);
  }
}
closeFile ();

if (!data_definition)
  printf ("\t...\tno\tDATA_SEGMENT\tfound!\n");
if (!code_definition)
  printf ("\t...\tno\tCODE_SEGMENT\tfound!\n");
*addrReg("FP") = MAXMEM;
*addrReg("SP") = MAXMEM;
*addrReg("RP") = -1;
*addrReg("HP") = getNextAddress ();
*addrReg("ZR") = 0;
PC = 0;
printf ("\t...\tdo\tyou\twish\tto\tview\tsetup\t(Y\tor\tN)?\t");
fgets(buffer,BUFFER_SIZE,stdin);
char ans = toupper(buffer[0]);
if (ans == 'Y')
{
  displayMemory ();
  displayMap ();
  displayNext ();
  displayGlobals ();
  displayExterns ();
  displayLabels ();
  displayCode ();
  printf ("KEY\tREGISTERS\n");
  printf ("%15s\t%15d\n","FP:",*addrReg("FP"));
  printf ("%15s\t%15d\n","SP:",*addrReg("SP"));
  printf ("%15s\t%15d\n","RP:",*addrReg("RP"));
  printf ("%15s\t%15d\n","HP:",*addrReg("HP"));
  printf ("%15s\t%15d\n","ZR:",*addrReg("ZR"));
  printf ("%15s\t%15d\n","PC:",PC);
  printf ("\n");
```

```
  }
  printf (" ... do you wish to view execution (Y or N)? ");
  fgets(buffer,BUFFER_SIZE,stdin);
  ans = toupper(buffer[0]);
  if (ans == 'Y')
    debug = 1;
  else
    debug = 0;

// execute code

  while (PC < INSTRUCTION_COUNTER)
  {
    FEDEX (debug);
  }

  exit (EXIT_SUCCESS);
}

/* ------------------------------------------------------------ */
```

5.1.1 buildit script

buildit

```
#!/bin/bash

gcc -o kbox kbox.c \
            ktypes.c memory.c registers.c code.c \
            setup.c fedex.c
```

5.2 ktypes: type declarations

ktypes.h

```
#ifndef _KTYPES_H
#define _KTYPES_H

/* ------------------------------------------------------------ */

// KBOX is built around 64-bit chunks of memory
//     called klunks!
// all data is represented as a klunk
//     int64 is a 64-bit twos-complement representation
//     flt64 is a 64-bit IEEE floating point representation
//     chr64 is a 56-bit waste of space,
//           storing the 8-bit character in lowest byte
//     str64 is a 64-bit figment of the imagination!
//           like C it is a pointer to array of char
//                terminated by '\0'

#define LITERAL_SIZE 256

typedef unsigned long long int  klunk;

typedef long long int    int64;  // 64-bit signed integer
typedef double           flt64;  // 64-bit signed real
typedef char             chr64;  // 64-bit ASCII character
typedef char*            str64;  // 64-bit pointer

/* ------------------------------------------------------------ */

// display the 64-bit contents of the klunk

void     show_int        (klunk);        // int64 value
void     show_flt        (klunk);        // flt64 value
void     show_chr        (klunk);        // chr64 value
void     show_str        (klunk);        // str64 value

void     show_hex        (klunk);        // raw hex value

/* ------------------------------------------------------------ */

// conversion algorithms between data representations

int64    klunk2int       (klunk);        // klunk -> int64
flt64    klunk2flt       (klunk);        // klunk -> flt64
chr64    klunk2chr       (klunk);        // klunk -> chr64
str64    klunk2str       (klunk);        // klunk -> str64

klunk    int2klunk       (int64);        // int64 -> klunk
klunk    flt2klunk       (flt64);        // flt64 -> klunk
klunk    chr2klunk       (chr64);        // chr64 -> klunk
klunk    str2klunk       (str64);        // str64 -> klunk
```

```
klunk    literal2klunk    (str64);          // literal -> data

/* ------------------------------------------------------------ */

#endif
```

ktypes.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "ktypes.h"
#include "memory.h"

/* ---------------------------------------------------------- */
// display klunk pattern as specified data type

void show_int (klunk k)
{ printf("%lld",k); }

void show_flt (klunk k)
{ printf("%f",klunk2flt(k)); }

void show_chr (klunk k)
{ printf("%c",k); }

void show_str (klunk k)
{ printf("%s",*((str64*)(&k))); }

void show_hex (klunk k)
{ printf("%016llx",k); }

/* ---------------------------------------------------------- */
// interpret klunk pattern as specific data type

int64 klunk2int (klunk k)
{ return *((int64*)(&k)); }

flt64 klunk2flt (klunk k)
{ return *((flt64*)(&k)); }

chr64 klunk2chr (klunk k)
{ return *((chr64*)(&k)); }

str64 klunk2str (klunk k)
{ return (str64)(k); }

/* ---------------------------------------------------------- */
// interpret data type bit pattern as simple klunk

klunk int2klunk (int64 i)
{ return *((klunk*)(&i)); }

klunk flt2klunk (flt64 f)
{ return *((klunk*)(&f)); }

klunk chr2klunk (chr64 c)
{ return *((klunk*)(&c)); }
```

```
klunk str2klunk (str64 s)
{ return (klunk)s; }

/* ------------------------------------------------------------ */
// represent immediate value as simple klunk

klunk literal2klunk (str64 lit)
{
  char symbol = lit[0];
  if (symbol == '\"')
  // string literal
  {
    // remove leading and trailing '\"'
    str64 result = (char*) malloc (LITERAL_SIZE);
    int new_len = strlen(lit)-2;
    for (int i = 0;i < new_len;i++)
      result[i] = lit[i+1];
    result[new_len] = '\0';
    return str2klunk (result);
  }
  else if (symbol == '\'')
  // character literal
    return chr2klunk (lit[1]);
  else if ((symbol == '+') ||
           (symbol == '-') ||
           (isdigit(symbol)))
  // numeric literal
    if (strchr(lit,'.') ||
        strchr(lit,'e') ||
        strchr(lit,'E'))
    // floating point literal
      return flt2klunk (atof(lit));
    else
    // integer literal
      return int2klunk (atoll(lit));
  else
  {
    printf ("invalid literal encountered: %s\n",lit);
    exit (EXIT_FAILURE);
  }
}

/* ------------------------------------------------------------ */
```

5.3 memory: storage definition

memory.h

```
#ifndef _MEMORY_H
#define _MEMORY_H

#include "ktypes.h"

/* ------------------------------------------------------------ */

// The MEMORY unit defines the organization of main memory
// within the KBOX machine. Although this seems almost
// trivial at first glance, there are some special
// considerations to keep in mind.

// Main memory is indeed a simple array of klunks
// of a specified size! Hence the two declarations
// immediately following these comments.

// However, we must keep track of static memory
// as it is either initialized or reserved.
// This requires a memory_map identifying the names of
// variables allocated space in memory, their location in
// memory, and the amount of space set aside for them.

// We need the information found in the memory map to:

//      retrieve information from memory by name
//      store information into memory by name

// Lastly, we need to keep track of last memory location
// assigned so that we know what location in memory
// is the next one available.

// Hence, main memory is actually a triple:
//      actual memory storage
//      a memory map
//      next available address

// Support methods for memory include:

//      _DEFINE                         display
//      _RESERVE                        displayStack

//                      memoryAddress
//                      peek
//                      poke

/* ------------------------------------------------------------ */

#define MAXMEM  1000000
typedef klunk   storageType [MAXMEM];   // array of klunks
```

```
/* ------------------------------------------------------------ */

typedef struct mapEntry* mapPtr;

struct mapEntry
{
  char*          name;          // identifier
  long long      address;       // base address in memory
  long long      size;          // size in 64-bit klunks
  mapPtr         next;
};

void             addEntry       (char*,long long,long long);

/* ------------------------------------------------------------ */

struct memoryType
{
  storageType    data;          // klunk storage area
  mapPtr         info;          // memory map
  long long      nextAddress;   // next available address
};

struct memoryType MEMORY;

/* ------------------------------------------------------------ */

void             initializeMemory(void);
long long        getNextAddress  (void);

void             _DEFINE        (char*,klunk);
void             _RESERVE       (char*,long long);

long long        memoryAddress  (char* ident);

klunk            peek           (long long);
void             poke           (long long,klunk);

void             displayMemory  (void);
void             displayMap     (void);
void             displayNext    (void);
void             displayStack   (long long);

void             reverseDisplay (mapPtr);

/* ------------------------------------------------------------ */

#endif
```

memory.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "memory.h"

/* ------------------------------------------------------------ */

void addEntry (char* name,long long address,long long size)
{
  mapPtr p = (mapPtr) malloc(sizeof(struct mapEntry));
  p->name = name;
  p->address = address;
  p->size = size;
  p->next = MEMORY.info;
  MEMORY.info = p;
}

/* ------------------------------------------------------------ */

void initializeMemory (void)
{
  MEMORY.info = NULL;
  MEMORY.nextAddress = (long long) 1;
  // note: memory address 0 is special! (i.e. NULL pointer)
}

long long getNextAddress (void)
{
  return MEMORY.nextAddress;
}

void _DEFINE (char* name,klunk value)
{
  // reserves space for identifier and initializes data
  long long index = MEMORY.nextAddress;
  addEntry (name,index,1);
  MEMORY.data[index] = value;
  MEMORY.nextAddress += 1;
}

void _RESERVE (char* name,long long size)
{
  // reserves specified space for identifier
  // but no initialization
  long long index = MEMORY.nextAddress;
  addEntry (name,index,size);
  MEMORY.nextAddress += size;
}

long long memoryAddress (char* ident)
{
```

```
  mapPtr p = MEMORY.info;
  while (p != NULL)
    if (strcmp (p->name,ident) == 0)
      return p->address;
    else
      p = p->next;
  return -1;
}

klunk peek (long long address)
{
  return MEMORY.data[address];
}

void poke (long long address,klunk data)
{
  MEMORY.data[address] = data;
}


void displayMemory (void)
{
  // display static memory (0 .. nextAddress-1)
  printf ("MEMORY STORAGE\n");
  for (long long i = 1;i < MEMORY.nextAddress;++i)
  {
    printf ("%7d ",i);
    show_hex (MEMORY.data[i]);
    printf ("\n");
  }
  printf ("\n");
}

void displayMap (void)
{
  printf ("MEMORY MAP\n");
  mapPtr p = MEMORY.info;
  if (p != NULL)
    reverseDisplay (p);
  printf ("\n");
}

void displayNext (void)
{
  printf ("NEXT ADDRESS: %d\n\n",MEMORY.nextAddress);
}

void displayStack (long long stackTop)
{
  for (long long i = MAXMEM-1;i >= stackTop;--i)
  {
    printf ("%7d ",i);
    show_hex (MEMORY.data[i]);
```

```
    printf ("\n");
  }
  printf ("\n");
}

/* ------------------------------------------------------------ */

void reverseDisplay (mapPtr p)
{
  if(p != NULL)
  {
    reverseDisplay (p->next);
    printf ("%15s %7d %7d\n",p->name,p->address,p->size);
  }
}

/* ------------------------------------------------------------ */
```

5.4 registers: calculation / comparison

registers.h

```
#ifndef _REGISTERS_H
#define _REGISTERS_H

#include "ktypes.h"

/* ---------------------------------------------------------- */

// The REGISTERS unit defines the organization and
// data representation for the 32 registers found
// within the KBOX machine.

// Since all calculations and comparisons are performed
// with the KBOX register sets, the two register sets
// IREGS and FREGS are comprised of 64 bit klunks.

// IREGS are used for all
// unsigned and signed integer operations

//      unsigned integer values: calculation and comparison
//      signed integer values: calculation and comparison
//      single ASCII values: comparison and manipulation

// FREGS are used for all floating point operations

//      floating point values: calculation and comparison

// Support procedures are provided display registers:

//      displayIregs
//      displayFregs

/* ---------------------------------------------------------- */

  #define NO_REGS       16

  typedef klunk         regsType[NO_REGS];
  typedef klunk*        klunkPtr;

  regsType              IREGS;
  regsType              FREGS;

// Access to registers is via register names!

//    I0 through I15 for integer registers (IREGS)
//    F0 through F15 for floating point registers (FREGS)

//    I7 references IREGS[7];
//    F12 references FREGS[12]; etc.
```

```
// In addition, many registers have special purpose roles
//     IA, IB, ... , HP and FA, FA, FC, and FD

/* ------------------------------------------------------------ */

int             isIreg          (char*);
int             isFreg          (char*);

klunkPtr        addrReg         (char*);

void            displayIregs    (char);
void            displayFregs    (void);

/* ------------------------------------------------------------ */

char*           checkAlias      (char*);

/* ------------------------------------------------------------ */

#endif
```

registers.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "registers.h"

/* ------------------------------------------------------------ */

int isIreg (char* regName)
{
  regName = checkAlias (regName);
  if (toupper(regName[0]) != 'I')
    return 0;
  int index = atoi (regName+1);
  return (0 <= index < NO_REGS) ? 1 : 0;
}

int isFreg (char* regName)
{
  regName = checkAlias (regName);
  if (toupper(regName[0]) != 'F')
    return 0;
  int index = atoi (regName+1);
  return (0 <= index < NO_REGS) ? 1 : 0;
}

/* ------------------------------------------------------------ */

klunkPtr addrReg (char* regName)
{
  regName = checkAlias (regName);
  int index = atoi (regName+1);
  if (!(0 <= index < NO_REGS))
  {
    printf ("invalid register reference!\n");
    return NULL;
  }
  if (toupper(regName[0]) == 'I')
    return &(IREGS[index]);
  else if (toupper(regName[0]) == 'F')
    return &(FREGS[index]);
  else
  {
    printf ("invalid register reference!\n");
    return NULL;
  }
}

/* ------------------------------------------------------------ */

void displayIregs (char f)
```

```
{
  // show IREGs in hexadecimal or signed integer format
  char fmt = toupper(f);
  if ((fmt != 'H') && (fmt != 'S'))
  {
    printf ("  ...  invalid display option ([h] or s)!\n");
    return;
  }
  printf ("\nInteger Registers:\n");
  for (int i = 0;i < NO_REGS;++i)
  {
    printf ("I%02d  ",i);
    if (fmt == 'H')
      show_hex (IREGS[i]);
    else
      show_int (IREGS[i]);
    printf ("\n");
  }
  printf ("\n");
}

void displayFregs (void)
{
  // show FREGs in floating point format
  printf ("\nFloating Point Registers:\n");
  for (int i = 0;i < NO_REGS;++i)
  {
    printf ("F%02d  ",i);
    show_flt (FREGS[i]);
    printf ("\n");
  }
  printf ("\n");
}

/* ------------------------------------------------------------ */

char* checkAlias (char* regName)
{
  if (strcmp(regName,"IA") == 0)
    return "I0";
  else if (strcmp(regName,"IB") == 0)
    return "I1";
  else if (strcmp(regName,"IC") == 0)
    return "I2";
  else if (strcmp(regName,"ID") == 0)
    return "I3";
  else if (strcmp(regName,"SAR") == 0)
    return "I4";
  else if (strcmp(regName,"DAR") == 0)
    return "I5";
  else if (strcmp(regName,"OR") == 0)
    return "I6";
  else if (strcmp(regName,"IR") == 0)
```

```
    return "I7";
  else if (strcmp(regName,"SP") == 0)
    return "I12";
  else if (strcmp(regName,"FP") == 0)
    return "I13";
  else if (strcmp(regName,"RP") == 0)
    return "I14";
  else if (strcmp(regName,"HP") == 0)
    return "I15";
  else if (strcmp(regName,"ZR") == 0)
    return "I11";
  else if (strcmp(regName,"FA") == 0)
    return "F0";
  else if (strcmp(regName,"FB") == 0)
    return "F1";
  else if (strcmp(regName,"FC") == 0)
    return "F2";
  else if (strcmp(regName,"FD") == 0)
    return "F3";
  else
    return regName;
}

/* ------------------------------------------------------------ */
```

5.5 code: structure / components

code.h

```
#ifndef _CODE_H
#define _CODE_H

#include "ktypes.h"

/* ------------------------------------------------------------ */

// The CODE unit has the responsibility to:

//      organize and save KCODE instructions
//      to maintain extern references within KCODE
//      to maintain global references within KCODE
//      to maintain label references within KCODE
//      to maintain the program counter PC
//      to maintain the program FLAGS

// Initial declarations specify table structures for:

//      extern references
//      global references
//      label references: name and instruction location
//      kcode instructions: opcode, oper1, oper2, oper3

// Final declaration specifies the strucute of CODE:

// Most of the methods listed are self-explanatory.
// However, the following are of special note:

// searchLabels
//      check for duplicates
//         and retrieve instruction location
// addInstruction
//      create instruction list in precise order
//      maintain instruction location (for labels)
// fetchInstruction
//      critical at execution time
//      retrieve proper item from the instruction list

/* ------------------------------------------------------------ */

struct labelData
{
   char*                name;
   long long            location;
};

struct labelEntry
{
```

```
  struct labelData       data;
  struct labelEntry*     next;
};

struct labelEntry*       LABELS;

void              displayLabels    (void);

long long         labelAddress     (char*);

struct nameEntry
{
  char*                  data;
  struct nameEntry*      next;
};

struct nameEntry*        GLOBALS;
struct nameEntry*        EXTERNS;

void              displayGlobals   (void);
void              displayExterns   (void);

void              _LABEL           (char*);
void              _EXTERN          (char*);
void              _GLOBAL          (char*);

int               isLabel          (char*);
int               isExtern         (char*);
int               isGlobal         (char*);

/* ------------------------------------------------------------ */

// important CPU flags
//    PC = program counter
//    EQ = equal flag
//    GT = greater than flag
//    LT = less than flag

// INSTRUCTION_COUNTER maintains instruction count
//    current size of instruction list
//    necessary for proper definition of labels

long long         PC;
int               EQ;
int               GT;
int               LT;

long long         INSTRUCTION_COUNTER;

void              setPC            (long long);

void              setEQ            (void);
void              setGT            (void);
```

```
void            setLT           (void);

/* ------------------------------------------------------------ */

struct codeData
{
   char*        opcode;
   char*        oper1;
   char*        oper2;
   char*        oper3;
};

#define MAXCODE 5000
struct codeData INSTRUCTIONS[MAXCODE];

void            displayCode     (void);

void            addInstruction  (char*,char*,char*,char*);

struct codeData fetchInstruction (void);

/* ------------------------------------------------------------ */

#endif
```

code.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "code.h"

/* ------------------------------------------------------------ */

void displayGlobals (void)
{
  printf ("GLOBALS\n");
  struct nameEntry* p = GLOBALS;
  while (p != NULL)
  {
    printf ("%15s\n",p->data);
    p = p->next;
  }
  printf ("\n");
}

void displayExterns (void)
{
  printf ("EXTERNS\n");
  struct nameEntry* p = EXTERNS;
  while (p != NULL)
  {
    printf ("%15s\n",p->data);
    p = p->next;
  }
  printf ("\n");
}

void displayLabels (void)
{
  printf ("LABELS\n");
  struct labelEntry* p = LABELS;
  while (p != NULL)
  {
    printf ("%15s %15d\n",p->data.name,p->data.location);
    p = p->next;
  }
  printf ("\n");
}

long long labelAddress (char* ident)
{
  struct labelEntry* p = LABELS;
  while (p != NULL)
  {
    if (strcmp(p->data.name,ident) == 0)
      return p->data.location;
    else
```

```
      p = p->next;
  }
  return -1;
}

/* ------------------------------------------------------------ */

void _LABEL (char* ident)
{
  if (isLabel(ident))
    printf ("duplicate label identifier: %s!\n");
  else
  {
    struct labelEntry* p =
      malloc (sizeof(struct labelEntry));
    p->data.name = ident;
    p->data.location = INSTRUCTION_COUNTER;
    p->next = LABELS;
    LABELS = p;
  }
}

void _EXTERN (char* ident)
{
  if (isExtern(ident))
    printf ("duplicate extern identifier: %s!\n");
  else
  {
    struct nameEntry* p =
      malloc (sizeof(struct nameEntry));
    p->data = ident;
    p->next = EXTERNS;
    EXTERNS = p;
  }
}

void _GLOBAL (char* ident)
{
  if (isGlobal(ident))
    printf ("duplicate extern identifier: %s!\n");
  else
  {
    struct nameEntry* p =
      malloc (sizeof(struct nameEntry));
    p->data = ident;
    p->next = GLOBALS;
    GLOBALS = p;
  }
}

/* ------------------------------------------------------------ */

int isLabel (char* ident)
```

```
{
  struct labelEntry* p = LABELS;
  int found = 0;
  while ((p != NULL) & !found)
    if (strcmp(p->data.name,ident) == 0)
      return 1;
    else
      p = p->next;
  return 0;
}

int isExtern (char* ident)
{
  struct nameEntry* p = EXTERNS;
  int found = 0;
  while ((p != NULL) & !found)
    if (strcmp(p->data,ident) == 0)
      return 1;
    else p = p->next;
  return 0;
}

int isGlobal (char* ident)
{
  struct nameEntry* p = GLOBALS;
  int found = 0;
  while ((p != NULL) & !found)
    if (strcmp(p->data,ident) == 0)
      return 1;
    else
      p = p->next;
  return 0;
}

/* ------------------------------------------------------------ */

void setPC (long long instruction)
{ PC = instruction; }

void setEQ (void)
{ EQ = 1; GT = 0; LT = 0; }

void setGT (void)
{ EQ = 0; GT = 1; LT = 0; }

void setLT (void)
{ EQ = 0; GT = 0; LT = 1; }

/* ------------------------------------------------------------ */

void displayCode (void)
{
  printf ("INSTRUCTIONS\n\n");
```

```
  printf ("%15s %15d\n\n","COUNTER:",INSTRUCTION_COUNTER);
  for (long long index=0;index<INSTRUCTION_COUNTER;index++)
  {
    struct codeData instruction = INSTRUCTIONS[index];
    printf ("%15s %15s %15s %15s\n",
      instruction.opcode,
      instruction.oper1,
      instruction.oper2,
      instruction.oper3);
  }
  printf ("\n");
}

void addInstruction (char* op, char* oper1,
                     char* oper2, char* oper3)
{
  struct codeData instruction = {op,oper1,oper2,oper3};
  INSTRUCTIONS[INSTRUCTION_COUNTER++] = instruction;
}

struct codeData fetchInstruction (void)
{ return INSTRUCTIONS[PC++]; }

/* ------------------------------------------------------------ */
```

5.6 setup: process KCODE file

setup.h

```
#ifndef _SETUP_H
#define _SETUP_H

#include "ktypes.h"
#include "registers.h"
#include "memory.h"
#include "code.h"

/* ------------------------------------------------------------ */

// setup.h

// setup.h and setup.c are designed
// to read a KCODE source file

//    - openFile          provide access to the source file
//    - closeFile

//    - readLine          get the next line of code
//    - isBlank           if blank line then read next
//    - skipBlanks        trims leading blanks

//    - getCode           get directive or instruction
//    - getOperand        get operand(s) for code

/* ------------------------------------------------------------ */

#define  BUFFER_SIZE 256

FILE*    infile;
char     buffer[BUFFER_SIZE];
int      pos;
int      done;

void     openFile (char*);
void     closeFile (void);
void     readLine (void);
int      isBlankLine (void);
void     skipBlanks (void);
char*    getCode (void);
char*    getOperand (void);

/* ------------------------------------------------------------ */

#endif
```

setup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "setup.h"

/* ------------------------------------------------------------------ */

void openFile (char* kcodeFile)
{
  infile = fopen(kcodeFile,"r");
  if (infile == NULL)
  {
    printf ("unable to open %s!\n",kcodeFile);
    exit (EXIT_FAILURE);
  }
  done = 0;
}

void closeFile (void)
{
  fclose (infile);
}

/* ------------------------------------------------------------------ */

void readLine (void)
{
  fgets (buffer,BUFFER_SIZE,infile);
  if (!feof(infile))
  {
    pos = 0;
    skipBlanks ();
  }
  else
    done = 1;
}

void skipBlanks (void)
{
  while ((pos < strlen(buffer)) && isblank(buffer[pos]))
    pos++;
}

int isBlankLine (void)
{
  if ((pos < strlen(buffer)) &&
      ((buffer[pos] == '\n')|| (buffer[pos] == '#')))
    return 1;
  else
    return 0;
```

```
}

/* ------------------------------------------------------------ */

char* getCode (void)
{
  char* result = (char*) malloc (LITERAL_SIZE);
  int i = 0;
  if (buffer[pos] == '_')
  {
  // KCODE directive
    while (isalpha(buffer[pos]) || (buffer[pos] == '_'))
      result[i++] = toupper(buffer[pos++]);
    result[i] = '\0';
    skipBlanks ();
  }
  else if (isalpha(buffer[pos]))
  {
  // KCODE instruction
    while (isalnum(buffer[pos]))
      result[i++] = toupper(buffer[pos++]);
    result[i] = '\0';
    skipBlanks ();
  }
  else
  {
  // invalid KCODE entry
    printf ("invalid line: %s\n",buffer);
    exit (EXIT_FAILURE);
  }
  return result;
}

/* ------------------------------------------------------------ */

char* getOperand (void)
{
  char* result = (char*) malloc (LITERAL_SIZE);
  int i = 0;
  if (isBlankLine())
  {
    result[0] = '\0';
    return result;
  }
  else if (isalpha(buffer[pos]))
  {
  // label or identifier
    while (isalnum(buffer[pos]))
      result[i++] = toupper(buffer[pos++]);
    result[i] = '\0';
    skipBlanks ();
  }
  else if ((buffer[pos] == '\"') ||
```

```
          (buffer[pos] == '\''))
  {
  // character or string literal
    char terminal = buffer[pos];
    result[i++] = buffer[pos++];
    while (buffer[pos] != terminal)
      result[i++] = buffer[pos++];
    result[i++] = buffer[pos++];
    result[i] = '\0';
    skipBlanks ();
  }
  else if ((buffer[pos] == '+') ||
           (buffer[pos] == '-') ||
           (isdigit(buffer[pos])))
  {
    // numeric literal: integer
    if ((buffer[pos] == '+') || (buffer[pos] == '-'))
      result[i++] = buffer[pos++];
    while (isdigit(buffer[pos]))
      result[i++] = buffer[pos++];
    // numeric literal: floating point
    if (buffer[pos] == '.')
    {
      result[i++] = buffer[pos++];
      while (isdigit(buffer[pos]))
        result[i++] = buffer[pos++];
    }
    // numeric literal: floating point
    if ((buffer[pos] == 'e') || (buffer[pos] == 'E'))
    {
      result[i++] = buffer[pos++];
      if ((buffer[pos] == '+') || (buffer[pos] == '-'))
      {
        result[i++] = buffer[pos++];
        while (isdigit(buffer[pos]))
          result[i++] = buffer[pos++];
      }
    }
    result[i] = '\0';
    skipBlanks ();
  }
  else if (buffer[pos] == '=')
  {
  // memory address
    result[i++] = buffer[pos++];
    while (isalnum(buffer[pos]))
      result[i++] = toupper(buffer[pos++]);
    result[i] = '\0';
    skipBlanks ();
  }
  else
  {
    result[i] = '\0';
```

```
    skipBlanks ();
  }
  return result;
}

/* ------------------------------------------------------------ */
```

5.7 fedex: execute KCODE file

fedex.h

```
#ifndef _FEDEX_H
#define _FEDEX_H

#include "ktypes.h"
#include "registers.h"
#include "memory.h"
#include "code.h"

/* ------------------------------------------------------------ */

// fedex.h and fedex.c are designed
//       to simulate KCODE execution

// The single procedure (FEDEX)
//    - first recognizes
//    - then simulates each instruction
//        - updates register contents
//        - updates PC and FLAGS
//        - stores/retrieves data to/from memory

/* ------------------------------------------------------------ */

int       debug;

void      FEDEX             (int);

/* ------------------------------------------------------------ */

#endif
```

fedex.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "fedex.h"

/* ------------------------------------------------------------------ */

void FEDEX (int debug)
{
  struct codeData instruction = fetchInstruction ();
  char* code = instruction.opcode;
  char* op1 =  instruction.oper1;
  char* op2 =  instruction.oper2;
  char* op3 =  instruction.oper3;

  if (debug)
    printf ("%s\n",code);

  if (strcmp(code,"HALT") == 0)
    exit (EXIT_SUCCESS);

  else if (strcmp(code,"I2F") == 0)
  {
    if ((!isIreg(op2)) || (!isFreg(op1)))
    {
      printf ("invalid I2F operands: %s,%s\n",op1,op2);
      exit (EXIT_FAILURE);
    }
    klunk* src_address = addrReg(op2);
    klunk* dst_address = addrReg(op1);
    *dst_address =
      flt2klunk ((flt64)(klunk2int(*src_address)));
  }
  else if (strcmp(code,"F2I") == 0)
  {
    if ((!isFreg(op2)) || (!isIreg(op1)))
    {
      printf ("invalid F2I operands: %s,%s\n",op1,op2);
      exit (EXIT_FAILURE);
    }
    klunk* src_address = addrReg(op2);
    klunk* dst_address = addrReg(op1);
    *dst_address =
      int2klunk ((int64)(klunk2flt(*src_address)));
  }

  else if (strcmp(code,"LDA") == 0)
  {
    if (op2[0] != '=')
    {
```

```
      printf ("invalid LDA second operand: %s\n",op2);
      exit (EXIT_FAILURE);
    }
    if (isIreg(op1))
    {
      klunk* reg = addrReg(op1);
      char* memLabel = op2+1;
      long long loc = memoryAddress (memLabel);
      if (loc == -1)
      {
        printf ("invalid LDA second operand: %s\n",op2);
        exit (EXIT_FAILURE);
      }
      *reg = loc;
    }
    else
    {
      printf ("invalid LDA first operand: %s\n",op1);
      exit (EXIT_FAILURE);
    }
  }

  else if (strcmp(code,"LDR") == 0)
  {
    if (!isIreg(op2))
    {
      printf ("invalid LDR second operand: %s\n",op2);
      exit (EXIT_FAILURE);
    }
    klunk* src_address = addrReg(op2);
    klunk* dst_address;
    if (isIreg(op1) || isFreg(op1))
      dst_address = addrReg(op1);
    else
    {
      printf ("invalid LDR first operand: %s\n",op1);
      exit (EXIT_FAILURE);
    }
    *dst_address = peek(*src_address);
  }
  else if (strcmp(code,"STR") == 0)
  {
    if (!isIreg(op2))
    {
      printf ("invalid STR second operand: %s\n",op2);
      exit (EXIT_FAILURE);
    }
    klunk* dst_address = addrReg(op2);
    klunk* src_address;
    if (isIreg(op1) || isFreg(op1))
      src_address = addrReg(op1);
    else
    {
```

```
    printf ("invalid STR first operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  poke (*dst_address,*src_address);
}

else if (strcmp(code,"MOV") == 0)
{
  klunk* dst_address;
  klunk* src_address;
  if (isIreg(op1) || isFreg(op1))
    dst_address = addrReg(op1);
  else
  {
    printf ("invalid MOV first operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  if (isIreg(op2) || isFreg(op2))
    src_address = addrReg(op2);
  else
  {
    printf ("invalid MOV second operand: %s\n",op2);
    exit (EXIT_FAILURE);
  }
  *dst_address = *src_address;
}

else if (strcmp(code,"MOVI") == 0)
{
  klunk* dst_address;
  if (isIreg(op1) || isFreg(op1))
    dst_address = addrReg(op1);
  else
  {
    printf ("invalid MOVI first operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  klunk imm_value = literal2klunk (op2);
  *dst_address = imm_value;
}

else if (strcmp(code,"PUSH") == 0)
{
  if (isIreg(op1) || isFreg(op1))
  {
    (*addrReg("SP"))--;
    poke (*addrReg("SP"),*addrReg(op1));
  }
  else
  {
    printf ("invalid PUSH operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
```

```
  }
  else if (strcmp(code,"POP") == 0)
  {
    if (isIreg(op1) || isFreg(op1))
    {
      *addrReg(op1) = peek (*addrReg("SP"));
      (*addrReg("SP"))++;
    }
    else
    {
      printf ("invalid POP operand: %s\n",op1);
      exit (EXIT_FAILURE);
    }
  }

  else if (strcmp(code,"ADD") == 0)
  {
    if ((!isIreg(op1)) ||
        (!isIreg(op2)) ||
        (!isIreg(op3)))
    {
      printf ("invalid ADD operands: %s, %s, %s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 =
      int2klunk ((klunk2int) (*address2) +
                 (klunk2int) (*address3));
  }
  else if (strcmp(code,"SUB") == 0)
  {
    if ((!isIreg(op1)) ||
        (!isIreg(op2)) ||
        (!isIreg(op3)))
    {
      printf ("invalid SUB operands: %s, %s, %s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 =
      int2klunk ((klunk2int) (*address2) -
                 (klunk2int) (*address3));
  }
  else if (strcmp(code,"MUL") == 0)
  {
    if ((!isIreg(op1)) ||
        (!isIreg(op2)) ||
```

```
      (!isIreg(op3)))
  {
    printf ("invalid mul operands: %s, %s, %s\n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  *address1 =
    int2klunk ((klunk2int) (*address2) *
               (klunk2int) (*address3));
}
else if (strcmp(code,"DIV") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid DIV operands: %s, %s, %s\n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  if ((klunk2int)(*address3) == 0)
  {
    printf ("division by 0 not permitted!\n");
    exit (EXIT_FAILURE);
  }
  *address1 =
    int2klunk ((klunk2int) (*address2) /
               (klunk2int) (*address3));
}
else if (strcmp(code,"MOD") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid MOD operands: %s, %s, %s\n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  if ((klunk2int)(*address3) == 0)
  {
    printf ("division by 0 not permitted!\n");
    exit (EXIT_FAILURE);
  }
```

```
    *address1 =
      int2klunk ((klunk2int) (*address2) %
                 (klunk2int) (*address3));
  }
  else if (strcmp(code,"NEG") == 0)
  {
    if (!isIreg(op1))
    {
      printf ("invalid NEG operand: %s\n",op1);
      exit (EXIT_FAILURE);
    }
    klunk* address = addrReg(op1);
    *address = int2klunk (- (*address));
  }

  else if (strcmp(code,"UADD") == 0)
  {
    if ((!isIreg(op1)) || (!isIreg(op2)) || (!isIreg(op3)))
    {
      printf ("invalid UADD operands: %s, %s, $s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 = *address2 + *address3;
  }
  else if (strcmp(code,"USUB") == 0)
  {
    if ((!isIreg(op1)) ||
        (!isIreg(op2)) ||
        (!isIreg(op3)))
    {
      printf ("invalid USUB operands: %s, %s, $s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 = *address2 - *address3;
  }
  else if (strcmp(code,"UMUL") == 0)
  {
    if ((!isIreg(op1)) ||
        (!isIreg(op2)) ||
        (!isIreg(op3)))
    {
      printf ("invalid UMUL operands: %s, %s, $s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
```

```
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  *address1 = *address2 * *address3;
}
else if (strcmp(code,"UDIV") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid UDIV operands: %s, %s, $s\n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  if (*address3 == 0ULL)
  {
    printf ("division by 0 not permitted!\n");
    exit (EXIT_FAILURE);
  }
  *address1 = *address2 / *address3;
}
else if (strcmp(code,"UMOD") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid UMOD operands: %s, %s, %s\n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  if (*address3 == 0ULL)
  {
    printf ("division by 0 not permitted!\n");
    exit (EXIT_FAILURE);
  }
  *address1 = *address2 % *address3;
}

else if (strcmp(code,"FADD") == 0)
{
  if ((!isFreg(op1)) ||
      (!isFreg(op2)) ||
      (!isFreg(op3)))
  {
    printf ("invalid FADD operands: %s, %s, $s\n",
```

```
            op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 =
      flt2klunk (klunk2flt (*address2) +
                 klunk2flt (*address3));
  }
  else if (strcmp(code,"FSUB") == 0)
  {
    if ((!isFreg(op1)) ||
        (!isFreg(op2)) ||
        (!isFreg(op3)))
    {
      printf ("invalid FSUB operands: %s, %s, $s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 =
      flt2klunk (klunk2flt (*address2) -
                 klunk2flt (*address3));
  }
  else if (strcmp(code,"FMUL") == 0)
  {
    if ((!isFreg(op1)) ||
        (!isFreg(op2)) ||
        (!isFreg(op3)))
    {
      printf ("invalid FMUL operands: %s, %s, $s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 =
      flt2klunk (klunk2flt (*address2) *
                 klunk2flt (*address3));
  }
  else if (strcmp(code,"FDIV") == 0)
  {
    if ((!isFreg(op1)) ||
        (!isFreg(op2)) ||
        (!isFreg(op3)))
    {
      printf ("invalid FDIV operands: %s, %s, $s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
```

```
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    if ((klunk2flt)(*address3) == 0.0)
    {
      printf ("division by 0 not permitted!\n");
      exit (EXIT_FAILURE);
    }
    *address1 =
      flt2klunk (klunk2flt (*address2) /
                 klunk2flt (*address3));
  }
  else if (strcmp(code,"FNEG") == 0)
  {
    if (!isFreg(op1))
    {
      printf ("invalid FNEG operand: %s\n",op1);
      exit (EXIT_FAILURE);
    }
    klunk* address = addrReg(op1);
    *address = flt2klunk (- klunk2flt(*address));
  }

  else if (strcmp(code,"LAND") == 0)
  {
    if ((!isIreg(op1)) ||
        (!isIreg(op2)) ||
        (!isIreg(op3)))
    {
      printf ("invalid LAND operands: %s, %s, %s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 = *address2 && *address3;
  }
  else if (strcmp(code,"LOR") == 0)
  {
    if ((!isIreg(op1)) ||
        (!isIreg(op2)) ||
        (!isIreg(op3)))
    {
      printf ("invalid LOR operands: %s, %s, %s\n",
              op1,op2,op3);
      exit (EXIT_FAILURE);
    }
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    klunk* address3 = addrReg(op3);
    *address1 = *address2 || *address3;
```

```
}
else if (strcmp(code,"LXOR") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid LXOR operands: %s, %s, %s\n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  *address1 =
    ((*address2 && (!*address3)) ||
    ((!*address2) && *address3));
}
else if (strcmp(code,"LNOT") == 0)
{
  if (!isIreg(op1))
  {
    printf ("invalid LNOT operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  klunk* address = addrReg(op1);
  *address = !*address;
}

else if (strcmp(code,"BAND") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid BAND operands: %s, %s, %s/n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  *address1 = *address2 & *address3;
}
else if (strcmp(code,"BOR") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid BOR operands: %s, %s, %s/n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
```

```
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  *address1 = *address2 | *address3;
}
else if (strcmp(code,"BXOR") == 0)
{
  if ((!isIreg(op1)) ||
      (!isIreg(op2)) ||
      (!isIreg(op3)))
  {
    printf ("invalid BXOR operands: %s, %s, %s/n",
            op1,op2,op3);
    exit (EXIT_FAILURE);
  }
  klunk* address1 = addrReg(op1);
  klunk* address2 = addrReg(op2);
  klunk* address3 = addrReg(op3);
  *address1 = *address2 ^ *address3;
}
else if (strcmp(code,"BNOT") == 0)
{
  if (!isIreg(op1))
  {
    printf ("invalid BNOT operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  klunk* address = addrReg(op1);
  *address = ~*address;
}

else if (strcmp(code,"CMP") == 0)
{
  if ((isIreg(op1)) && (isIreg(op2)))
  {
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    int64 diff =
      klunk2int(*address1) - klunk2int(*address2);
    if (diff == 0LL)
      setEQ ();
    else if (diff > 0LL)
      setGT ();
    else // (diff < 0LL)
      setLT ();
  }
  else
  {
    printf ("invalid CMP operands: %s, %s\n");
    exit (EXIT_FAILURE);
   }
}
```

```
else if (strcmp(code,"UCMP") == 0)
{
  if ((isIreg(op1)) && (isIreg(op2)))
  {
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    if (*address1 == *address2)
      setEQ ();
    else if (*address1 > *address2)
      setGT ();
    else // (*address1 < *address2)
      setLT ();
  }
  else
  {
    printf ("invalid UCMP operands: %s, %s\n");
    exit (EXIT_FAILURE);
   }
}
else if (strcmp(code,"FCMP") == 0)
{
  if ((isFreg(op1)) && (isFreg(op2)))
  {
    klunk* address1 = addrReg(op1);
    klunk* address2 = addrReg(op2);
    flt64 diff =
      klunk2flt(*address1) - klunk2flt(*address2);
    if (diff == 0.0)
      setEQ ();
    else if (diff > 0.0)
      setGT ();
    else // (diff < 0.0)
      setLT ();
  }
  else
  {
    printf ("invalid FCMP operands: %s, %s\n");
    exit (EXIT_FAILURE);
   }
}

else if (strcmp(code,"JMP") == 0)
{
  long long pc = labelAddress (op1);
  if (pc < 0LL)
  {
    printf ("undefined label: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  else
    setPC (pc);
}
```

```
else if (strcmp(code,"JEQ") == 0)
{
  long long pc = labelAddress (op1);
  if (pc < 0LL)
  {
    printf ("undefined label: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  if (EQ)
    setPC (pc);
}
else if (strcmp(code,"JGT") == 0)
{
  long long pc = labelAddress (op1);
  if (pc < 0LL)
  {
    printf ("undefined label: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  if (GT)
    setPC (pc);
}
else if (strcmp(code,"JLT") == 0)
{
  long long pc = labelAddress (op1);
  if (pc < 0LL)
  {
    printf ("undefined label: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  if (LT)
    setPC (pc);
}
else if (strcmp(code,"JNE") == 0)
{
  long long pc = labelAddress (op1);
  if (pc < 0LL)
  {
    printf ("undefined label: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  if (!EQ)
    setPC (pc);
}
else if (strcmp(code,"JGE") == 0)
{
  long long pc = labelAddress (op1);
  if (pc < 0LL)
  {
    printf ("undefined label: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  if (!LT)
```

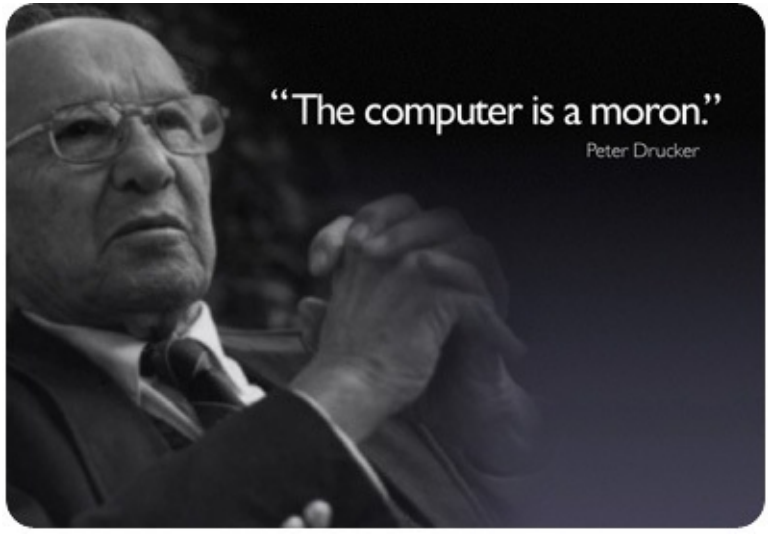
```
      setPC (pc);
  }
  else if (strcmp(code,"JLE") == 0)
  {
    long long pc = labelAddress (op1);
    if (pc < 0LL)
    {
      printf ("undefined label: %s\n",op1);
      exit (EXIT_FAILURE);
    }
    if (!GT)
      setPC (pc);
  }

  else if (strcmp(code,"ROL") == 0)
  {
    if (!isIreg(op1))
    {
      printf ("invalid ROL register: %s\n",op1);
      exit (EXIT_FAILURE);
    }
    klunk* address = addrReg(op1);
    char* imm = op2;
    int shift = atoi(imm) & 0X3F;
    klunk parta = *address << shift;
    klunk partb = *address >> (64-shift);
    *address = parta | partb;
  }
  else if (strcmp(code,"ROR") == 0)
  {
    if (!isIreg(op1))
    {
      printf ("invalid ROR register: %s\n",op1);
      exit (EXIT_FAILURE);
    }
    klunk* address = addrReg(op1);
    char* imm = op2;
    int shift = atoi(imm) & 0X3F;
    klunk parta = *address >> shift;
    klunk partb = *address << (64-shift);
    *address = parta | partb;
  }
  else if (strcmp(code,"SHL") == 0)
  {
    if (!isIreg(op1))
    {
      printf ("invalid SHL register: %s\n",op1);
      exit (EXIT_FAILURE);
    }
    klunk* address = addrReg(op1);
    char* imm = op2;
    int shift = atoi(imm) & 0X3F;
    *address = *address << shift;
```

```
}
else if (strcmp(code,"SHR") == 0)
{    if (!isIreg(op1))
  {
    printf ("invalid␣SHR␣register:␣%s\n",op1);
    exit (EXIT_FAILURE);
  }
  klunk* address = addrReg(op1);
  char* imm = op2;
  int shift = atoi(imm) & 0X3F;
  *address = *address >> shift;
}
else if (strcmp(code,"ASHL") == 0)
{    if (!isIreg(op1))
  {
    printf ("invalid␣ASHL␣register:␣%s\n",op1);
    exit (EXIT_FAILURE);
  }
  klunk* address = addrReg(op1);
  char* imm = op2;
  int shift = atoi(imm) & 0X3F;
  *address = *address << shift;
}
else if (strcmp(code,"ASHR") == 0)
{
  if (!isIreg(op1))
  {
    printf ("invalid␣ROL␣register:␣%s\n",op1);
    exit (EXIT_FAILURE);
  }
  // note: bit pattern must be considered as
  // a signed and not an unsigned value
  int64* address = (int64*)(addrReg(op1));
  char* imm = op2;
  int shift = atoi(imm) & 0X3F;
  *address = *address >> shift;
}

else if (strcmp(code,"CALL") == 0)
{
  long long lbl = labelAddress (op1);
  *(addrReg("RP")) = PC;;
  setPC (lbl);
}
else if (strcmp(code,"RET") == 0)
{
  long long ret = *(addrReg("RP"));
  if (ret >= 0)
    setPC (ret);
  else
    exit (EXIT_SUCCESS);
}
```

```
else if (strcmp(code,"INC") == 0)
{
  if (!isIreg(op1))
  {
    printf ("invalid INC operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  (*(addrReg(op1)))++;
}
else if (strcmp(code,"DEC") == 0)
{
  if (!isIreg(op1))
  {
    printf ("invalid DEC operand: %s\n",op1);
    exit (EXIT_FAILURE);
  }
  (*(addrReg(op1)))--;
}

else if (strcmp(code,"MALLOC") == 0)
{
  if ((!isIreg(op1)) || (!isIreg(op2)))
  {
    printf ("invalid MALLOC operands: %s, %s\n",op1,op2);
    exit (EXIT_FAILURE);
  }
  klunk* reg_addr = addrReg(op1);
  klunk no_klunks = *addrReg(op2);
  *reg_addr = *(addrReg("HP"));
  *(addrReg("HP")) += no_klunks;
}
else if (strcmp(code,"DALLOC") == 0)
{
  if ((!isIreg(op1)) || (!isIreg(op2)))
  {
    printf ("invalid DALLOC operands: %s, %s\n",op1,op2);
    exit (EXIT_FAILURE);
  }
  else
  {
     // yes, do nothing!
  }
}

else if (strcmp(code,"NOP") == 0)
  return;

else if (strcmp(code,"GET") == 0)
{
  (*(addrReg("SP")))--;
  char* fmt = op1;
  if (strcmp(fmt,"INT") == 0)
  {
```

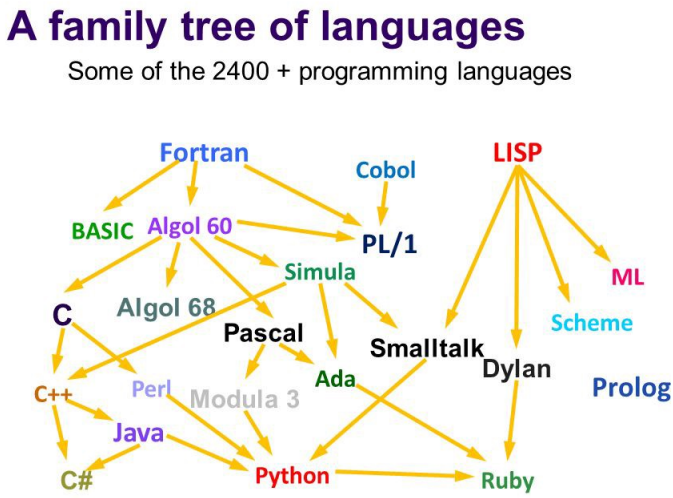
```
      int64 data;
      scanf ("%lld",&data);
      poke (*(addrReg("SP")),int2klunk(data));
    }
    else if (strcmp(fmt,"FLT") == 0)
    {
      flt64 data;
      scanf ("%lf",&data);
      poke (*(addrReg("SP")),flt2klunk(data));
    }
    else if (strcmp(fmt,"CHR") == 0)
    {
      chr64 data;
      scanf ("%c",&data);
      poke (*(addrReg("SP")),chr2klunk(data));
    }
    else if (strcmp(fmt,"STR") == 0)
    {
      char* data = (char*) malloc (LITERAL_SIZE);
      scanf ("%s",data);
      printf ("%s\n",data);
      poke (*(addrReg("SP")),str2klunk(data));
    }
    else
    {
      printf ("invalid format entry: %s\n", op1);
      exit (EXIT_FAILURE);
    }
  }
  else if (strcmp(code,"GETLN") == 0)
  {
    while (getchar()!='\n');
  }

  else if (strcmp(code,"PUT") == 0)
  {
    klunk data = peek (*(addrReg("SP")));
    char* fmt = op1;
    if (strcmp(fmt,"INT") == 0)
      show_int (data);
    else if (strcmp(fmt,"FLT") == 0)
      show_flt (data);
    else if (strcmp(fmt,"CHR") == 0)
      show_chr (data);
    else if (strcmp(fmt,"STR") == 0)
      show_str (data);
    else if ((strcmp(fmt,"HEX") == 0) ||
             (strcmp(fmt,"PTR") == 0))
      show_hex (data);
    else
    {
      printf ("invalid format entry: %s\n",op1);
      exit (EXIT_FAILURE);
```

```
    }
    (*(addrReg("SP")))++;
  }
  else if (strcmp(code,"PUTLN") == 0)
    printf ("\n");

  else
  {
    printf ("invalid instruction: %s\n",code);
    exit (EXIT_FAILURE);
  }
}

/* ------------------------------------------------------------ */
```


"The computer is a moron."
Peter Drucker

Chapter 6

Introduction



6.1 High Level Languages

Assembly languages are very fast and very efficient in implementing algorithms to solve problems. However, it is obvious that assembly language programming is very heavily detail-oriented with very specific instructions that work at the level of binary representation and manipulation. CISC machines may require a bit less detail and a more abstract level of instruction; but machine-level / assembly language coding can be very intense.

High level programming languages are a friendlier alternative. Instructions are are "English-like" in appearance, typically allow more flexibility in their format; and often allow for several distinct operations to be combined into a single statement. Although the ultimate goal is the same for both assembly languages and high level programming languages, the high level languages facilitate accomplishing the task with less annoyance.

For example, consider the following simple assignment statement common to so many programming languages:

```
X  :=  A * B  +  C ;
```

Now consider its equivalent implementation in the KCODE assembly language:

KCODE Equivalent

```
LDA     SAR =A
LDR     IA  SAR
PUSH    IA
LDA     SAR =B
LDR     IA  SAR
PUSH    IA
POP     IC
POP     IB
MUL     IA  IB  IC
PUSH    IA
LDA     SAR =C
LDR     IA  SAR
PUSH    IA
POP     IC
POP     IB
ADD     IA  IB  IC
PUSH    IA
LDA     DAR =X
POP     IA
STR     IA  DAR
```

Hight level languages eliminate some (much!) of the drudgery when encoding an algorithm. However, the obvious next question should be something like:

But how does the computer know what this strange new language is telling it to do?

And the simple answer is:

It has to be told in its native tongue.

When thinking about a new programming language, it is important to also be thinking about some form of translation program **from** the new language **to** the computer's machine or assembly language (or at least some intermediate language that the computer can already execute). As we previously discussed when implementing a C simulator for KCODE, translation techniques come in two flavors:

Definition 6.1.1. A **translator** is a program which analyzes statements found in a given programming language (the *source code*) and performs one of the following two important tasks:

- a **compiler** converts the source code into some form of alternate code (the *destination code*) which is executable by the underlying computer.
- an **interpreter** simulates how the resulting executable would *run* on the underlying computer.

A third closely related term needs to be introduced at this time – that of a parser.

Definition 6.1.2. A **parser** is a program which analyzes statements found in a given programming language (the *source code*) and determines whether or not the given code is a *syntactically correct* sequence of statements in the given programming language.

Its result is either ***success*** (i.e., no errors) or ***failure*** (i.e., errors).

No executable code is generated; no simulation is performed.

6.2 Categories of High Level Languages

High level programming languages are often categorized according to their general organization or "feel".

6.2.1 Procedural Languages

The first compiled programming language to be implemented was **FORTRAN**. It was specifically oriented to mathematical / scientific applications. But its fundamental emphasis was on organizing data into specific variables, defining subprograms (using functions and subroutines), and lastly transferring data among the various components (either using COMMON storage or using argument lists).

A number of other compiled languages quickly followed: **COBOL**, **ALGOL**, **PL/I**, ...

One concern that developed over time as programs grew larger and required more variables and more coding was that argument lists were becoming increasingly lengthy. Global variables tend to have "side effects" that can occur anywhere with the programming code; local variables tend to limit any damage to a specific area. So data transfer became more crucial as program size steadily increased.

6.2.2 Functional Languages

LISP was the first of the interpreted programming languages; its style is referred to as functional programming. Like **FORTRAN**, **LISP** had a mathematical emphasis – not so much in its applications but definitely in its structure. Everything is a function, either a built-in function or a user-defined function. By using simple ***composition*** of functions, that is, plugging the result of one function into a second function, a programmer can construct very powerful algorithms.

The fundamental unit in a **LISP** program is a simple ***list***.

(operator operand$_1$ operand$_2$ operand$_3$ ...)

The first item operates on all the remaining items! Any and all of the items (including the operator) may themselves be lists! Very simple to describe, but very time-consuming balancing parentheses!

Despite its age, **LISP** continues to remain popular in the field of Artificial Intelligence.

6.2.3 Object-Oriented Languages

Object-oriented languages attempt to address the argument list bottleneck that develops in complex algorithms when using a procedural language. Rather than view data and subprograms as granular and separate from one another, a broader focus could group items together as a unit on a much bigger scale than simple array structures and simple record structures.

Consider a typical personal bank account:

What information is appropriate to retain as part of the account:

- owner name
- owner address
- owner city, state, zip
- balance
- list of recent deposits
- list of recent withdrawals
- ... you get the idea

What subprograms are useful for proper maintenance of the account:

- make_deposit
- make_withdrawal
- update_balance
- change_of_address
- ... others

Object-oriented languages allow the programmer to create abstract classes (like bank_account) and bundle together data and subprograms (methods) as a single entity. The class name is now used to create specific objects within that class: paul_account or kathy_account.

Transferring an object through an argument list not only transfers all its information but also transfers all its methods as well)

C++ and JAVA would be current examples of object-oriented languages.

6.3 What Comprises a Programming Language

The developer of a programming language has many options to consider, from the most general to the most specific. Our presentation will focus initially on very general options and subsequently visit more specific characteristics.

6.3.1 Properties / Characteristics

- general categories
  - a procedural, functional, or object-oriented language?
- a typed or type-less language
  - do variables have a specific type attached to them? or not?
- type sizes
  - are data items fixed (static length)? or variable (dynamic length)?
- allocation of storage
  - static: is allocated only once, exists for the duration of the program
  - automatic: allocated and exists only as needed
  - dynamic: allocated and released at the programmer's discretion
- aggregation of data
  - are there constructors: arrays, records, strings, classes, others?
- operators
  - rules for precedence, associativity, type conversions, overloading?
- scope of variables
  - options: global versus local? linear scope or nested scope?

- control structures
  - sequential, selection, and repetition? are **goto**s allowed?
- subprograms / procedures
  - declaration, definition, and reference
- data transfer
  - call by value, call by variable, other?
  - arguments and return type: atomic types? structured types? other?
- recursive or non-recursive subprograms

6.3.2 Basic Building Blocks

All programming languages are built up from a very simple symbol set:

{ A, B, . . . , Z }
{ a, b, . . . , z }
{ 0, 1, . . . , 9 }
{ : ; . ( ) { } [ ] # ” = < > + - * / % }

From these symbols we define the basic building blocks of the language:

Identifiers are an integral component of any programming languages. Identifiers signify the user-named components that will appear within the program. Identifiers may ultimately represent a variable, a named constant, a named type, a function, or a subroutine. Identifiers typically start with an alphabetic character, followed by an arbitrary number of alphanumeric characters.

- Many languages allow for other special ASCII characters to appear within an identifier. The languages we consider in this book will not!
- Many languages are case sensitive. The languages we consider in this book are not!

Constants allow the programmer to include specific data values within a program. Constants generally come in two flavors: literal

constants or explicit values (e.g., 45, -23.75,"enter an int: ") and named constants (e.g., PI, RAD2, MESSAGE).

Atomic data types identify the fundamental or built-in data types that are part of the language. Standard types include integer (INT) and real (REAL), ASCII characters (CHAR) and ASCII strings, and sometimes BOOL or LOGICAL

Keywords appear for all intents and purposes to be identifiers. However, the role of a keyword (or a reserved word) is to define the organization or the structure of the programming language. Keywords should not be used as an identifier within a source program.

Labels are user-defined, like identifiers. However, their purpose is significantly different. Identifiers typically represent data values stored and manipulated within the program; labels identify key positions within the program.

Operators specify what operations are permissible on identifiers and constants. The most common operators are assignment, arithmetic calculation, arithmetic comparison, and the like.

Punctuation accounts for the remaining symbols listed above. Their specific role may vary across different programming languages. Common punctuation elements include:

- squiggly braces { } , parentheses ( ) , square braces [ ]
- a period . , a semicolon ; , a colon :

The nice thing about all these building blocks is that that are all very simple patterns: one or two elements from our allowable symbol set or arbitrarily long strings that adhere to fairly strict guidelines.

For example:

- a very common pattern for an identifier is:
  { A, B, . . . , Z } { A, B, . . . , Z, 0, 1, . . . , 9 }*
- common patterns for constants are:
  integer constant: { 1, 2, . . . , 9 } { 0, 1, . . . , 9 }*
  decimal constant: { 0, 1, . . . , 9 }+ . { 0, 1, . . . , 9 }+
  character: { ASCII character }
  string: { ASCII character }*

All the patterns above are fairly straight-forward. The patterns are commonly referred to as **regular expressions**. Their meaning is defined as follows:

- { some_list_of_symbols } select **exactly one** from the list of symbols
- { some_list_of_symbols }* select **zero or more** from the list of symbols
- { some_list_of_symbols }+ select **one or more** from the list of symbols

At the very onset of describing / defining a programming language, the basic symbol set and the building blocks for the language must be clearly delineated. Case sensitivity and inclusion of special characters can cause confusion and unnecessary grief to the programmer, if they are not clearly outlined.

6.3.3 Basic Structure

How does one adequately explain the structure of a programming language, with all its variations and constructs? The two most common techniques, in my opinion, are: **syntax diagrams** and **context free grammars**.

Syntax diagrams are very descriptive because they are very visual. They remind me of flowcharts showing how the various building blocks may be correctly sequenced within a valid program. Syntax diagrams typically take up several pages with charts and sub charts. But they are very easy to follow.

Context free grammars are production rules (substitution rules) describing how to build a valid program. In order to illustrate the use of production rules, let us consider a very specific component of probably every programming language in existence – evaluation and assigning an arithmetic expression.

Example

An arithmetic expression takes either variable data values or literal constant values and combines them using addition, subtraction, multiplication, and division. We all should know by now that:

- multiplication and division are done before addition and subtraction
  this is called *precedence* or *priority*
- equal priority operators are performed from left to right
  this is called *associativity*
- if there is any possible confusion
  use parentheses (and)

So, how would these principles appear in a context free grammar of production rules?

Simple Expression Grammar

start ⟶ statement

statement ⟶ IDENT := expression ;

expression ⟶ term more_terms

more_terms ⟶ addop term more_terms
more_terms ⟶

term ⟶ factor more_factors

more_factors ⟶ mulop factor more_factors
more_factors ⟶

factor ⟶ IDENT
factor ⟶ CONST
factor ⟶ (expression)

addop ⟶ +
addop ⟶ -

mulop ⟶ *
mulop ⟶ /

The left-hand-side of a production rule is a single item, which is referred to as a **non-terminal symbol**. The right-hand-side of a production rule is a sequence of grammar items in a specific order.

If a grammar item does **not** appear on the left-hand-side of a production rule (only on the right-hand-side) it is referred to as a **terminal symbol**. It may not be replaced.

The production rules provide a template on how to properly create a valid program starting with the very first production rule – its non-terminal symbol is commonly referred to as the **start symbol**.

Beginning with the start symbol, repeat the following until *only terminal symbols* remain:

- Substitute any non-terminal symbol found on the right-hand-side of the current expression with one of its production rules.

When only terminal symbols remain on the right-hand side, a valid program has been found!

For our example under considerations:

start symbol:
start

non-terminal symbols:
start, statement,expression
term, more_terms,
factor, more_factors,
addop, mulop

terminal symbols:
IDENT, := (assignment),
; (semicolon), CONST,
((left parenthesis),) (right parenthesis),
+ (addition), - (subtraction),
* (multiplication), / (division)

Now, let us examine the following question. Is the following statement ***valid*** according to our production rules?

X := A * B + C ;

In order to answer this question, we need to ***derive*** this fact according to the requirements of our production rules! Which is exactly what we will do on the next page.

start ⟶
statement ⟶

IDENT := expression ; ⟶
IDENT := term more_terms ; ⟶
IDENT := factor more_factors more_terms ; ⟶
IDENT := IDENT more_factors more_terms ; ⟶

IDENT := IDENT mulop factor more_factors more_terms ; ⟶
IDENT := IDENT * factor more_factors more_terms ; ⟶
IDENT := IDENT * IDENT more_factors more_terms ; ⟶
IDENT := IDENT * IDENT more_terms ; ⟶

IDENT := IDENT * IDENT addop term more_terms ; ⟶
IDENT := IDENT * IDENT + term more_terms ; ⟶
IDENT := IDENT * IDENT + IDENT more_terms ; ⟶
IDENT := IDENT * IDENT + IDENT ; ⟶

X := IDENT * IDENT + IDENT ; ⟶
X := A * IDENT + IDENT ; ⟶
X := A * B + IDENT ; ⟶
X := A * B + C ;

At each step in the process, one production rule has been applied (the specific non-terminal symbol is underlined on the line immediately above).

Most of the non-terminal symbols in this very elementary grammar have multiple possibilities. It should be obvious that the above example is **not** the **only** possible valid statement. It should also be obvious that finding a derivation for a give statement might involve a lot of trial and error (backtracking if one substitution does not work).

For the time being, though, we know just enough to continue on with our overview of programming language concepts. Hopefully, we will answer many of the questions that will remain on the tip of your tongue for now.

6.3.4 Basic Meaning

In this section we have the unenviable task of trying to explain what all this stuff is supposed to actually do! Grammars define **syntax** or structure for a valid program; **semantics** explain what it all means. The following English sentence might be valid ... but it is certainly an absurd statement!

The enormous shoes in the beehive are very hungry.

Programming languages must not only be syntactically correct; they must also be semantically meaningful. We should all be able to agree when a given program is valid; we should also all be in agreement regarding how it will perform.

A variety of shorthand methods exist for describing the semantics for programming languages. They are certainly valuable and worthy of further study at the graduate level. However, for an introductory course such as this, it would be more of a distraction than an aid to facilitate our progress. Rather, for now we will describe the semantic rules in plain English as best we can.

The following is a very limited summary of some typical semantic rules:

data types

- INT (integer data) is often a 64-bit 2s complement representation
- REAL (floating point data) is often a 64-bit IEEE representation

arithmetic operators

- addition, subtraction, multiplication, and division
- INT mode arithmetic yields INT results
- REAL mode arithmetic yields REAL results
- mixed mode arithmetic **promotes** INT values to REAL values

assignment operator

- INT ⟵ REAL **demotes** REAL value to INT
- REAL ⟵ INT **promotes** INT value to REAL

6.3.5 Putting It All Together

Each of the three previous subsections highlighted a specific aspect of a programming language. Basic Building Blocks emphasized the fundamental elements we need to consider; Basic Structure demonstrated how a valid program can be built up from fundamental elements; and Basic Meaning described how statements should properly execute. The subsequent chapters in this text focus on how these issues allow us to implement useful compilers for a given language.

Basic Building Blocks require an initial phase where an input stream of ASCII characters in free format is broken down into token pairs identify the fundamental components of the programming language. This phase is referred to as *lexical analysis* or *scanning*. The result of lexical analysis is an output stream of lexemes in which each line is a token-name and token-value pair.

Basic Structure requires a second phase where the stream of lexemes is compared against the production rules in the context free grammar. This phase is referred to as *parsing*. Since a typical context free grammar has a large number of production rules and any non-terminal symbol may have many production rules to choose from, it should be obvious that parsing can get very ugly! Fortunately, there is a very useful parsing technique that we will utilize to simplify our work. The result of parsing is a very simple outcome: the program is either VALID or INVALID; it does not generate any other output, unless we instruct it to tell us WHERE it is invalid.

Basic Meaning requires augmenting the parsing process above with additional capabilities. This phase is referred to as *compiling*. In addition to simultaneously parsing the input stream from the scanner, if the compiler recognizes a syntactically correct section of code it must also check whether that section is semantically correct. And if so, only then should it generate the assembly language code which performs the desired action.

In summary

- scanning: breaking down the source code
  into a simple stream of lexemes
- parsing: is the source code syntactically correct?

  Syntax Error Handling
- compiling: is the source code syntactically correct?

  Syntax Error Handling

  is the source code semantically correct?

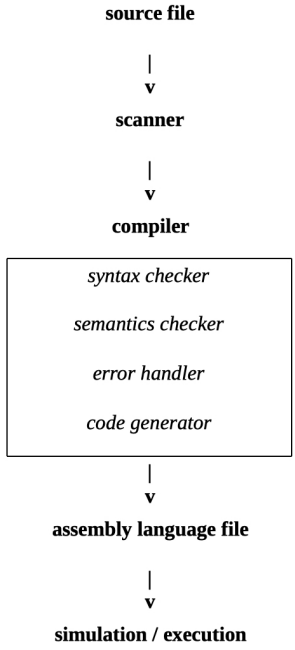
  Semantics Error Handling

  generate the appropriate assembly language code

  Code Generation

6.4 What IS the Compilation Process

The compilation process may be summarized very simply in the following diagram:

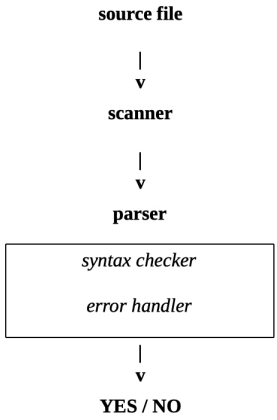


The **scanner** reads the source code file in the programming language and breaks it into a stream of lexeme pairs (a token category and a token value).

The **compiler** reads the incoming stream of lexeme pairs checks in order for:

- any syntax errors: violation of production rules
- any semantics errors: violations or inconsistencies of an instructions meaning
- error handling: how to gracefully shut down if a problem has occurred
- code generation: into the appropriate assembly language or intermediate language

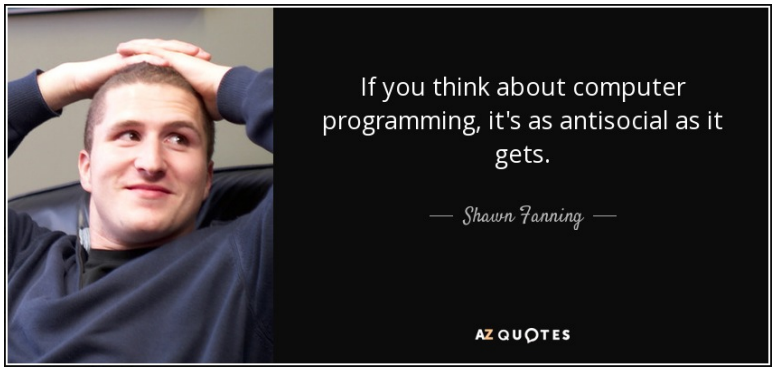
The parsing process is a considerably shorter version of the same diagram:



The **scanner** reads the source code file in the programming language and breaks it into a stream of lexeme pairs (a token category and a token value).

The **parser** reads the incoming stream of lexeme pairs checks in order for:

- any syntax errors: violation of production rules
- error handling: how to gracefully shut down if a problem has occurred
- significant difference is that nothing is generated as a result other than determining whether it is VALID code or not!


If you think about computer programming, it's as antisocial as it gets.
— Shawn Fanning —
AZ QUOTES

Chapter 7

CALC: An Arithmetic Calculator

This chapter introduces a very small programming language to illustrate the concepts essential to building a compiler. The subsequent chapters will then focus specifically on each of the three components we will have just discussed in greater detail. My goal in this part of the text is to provide an overview of the essentials regarding compilers without getting lost in intricate details.

We will build this small programming language around the context free grammar we discussed earlier regarding arithmetic expressions. The name for this programming language is **CALC: An Arithmetic Calculator**. We will define our programming language to evaluate such expressions, perform simple assignment statements, and allow for basic input and output.

Let us now clearly define each of the three important components of our programming language: the building blocks, the basic structure, and the basic meaning.

7.1 Building Blocks

Variables in this language will be very simple: { A ... Z } { A ... Z , 0 ... 9 } *

an initial alphabetic character

followed by any number of alphanumeric characters

variable names are **not** case sensitive!

Constants in this language come in three flavors:

NUMBER: { 0 ... 9 } +

DECIMAL: { 0 ... 9 } + . { 0 ... 9 } +

note the decimal point **must be** surrounded on both sides!

ASTRING: { ASCII character set } *

Operators in this language are basic arithmetic:

+ - * / :=

Punctuation in this language:

: and ;

Keywords in this language include the following:

ACHAR, ASTRING, BEGIN, DECIMAL,
END, GLOBAL, IDENTIFIER, INT,
NUMBER, PROCEDURE, PROGRAM, READLN, REAL,
VARIABLES, WRITELN

Data Types in this language are limited to two options:

INT and REAL

Chapter Eight: Scanning will focus on writing a program to identify these basic elements in an CALC programming language source code file.

7.2 Syntax

The following page contains the context free grammar for the CALC programming language. It is essentially the arithmetic expression grammar we discussed earlier with three additional items:

- input of one data value from the keyboard
- output of one data value to the monitor
- assignment of a data value to a variable

All of these items are within the executable code for the programming language. In addition, the programming language must clarify its basic organization:

- program structure, its start and its finish
- declarations for variables, its name and its data type
- executable code, its start and its finish

Chapter Nine: Parsing will focus on writing a program to verify that the basic elements in a CALC programming language source code file are combined properly to be a syntactically correct CALC program.

calc.grm

```
calc_program            PROGRAM IDENTIFIER
                        global_declarations
                        BEGIN procedure END

global_declarations     var_declarations

var_declarations        VARIABLES var_list
var_declarations

var_list                IDENTIFIER : var_type ;
                        more_var_list

more_var_list           var_list
more_var_list

var_type                atomic_type

atomic_type             INT
atomic_type             REAL

procedure               PROCEDURE IDENTIFIER
                        BEGIN statement_list END

statement_list          statement statement_list
statement_list

statement               executable_statement

executable_statement    read_statement
executable_statement    write_statement
executable_statement    assignment_statement

read_statement          READLN ( read_item ) ;

read_item               named_item

write_statement         WRITELN ( write_item ) ;

write_item              expression
write_item              ASTRING

assignment_statement    named_item := expression ;

expression              term more_terms

more_terms              addop term more_terms
more_terms

addop                   +
addop                   -
```

term	factor more_factors
more_factors	mulop factor more_factors
more_factors	
mulop	*
mulop	/
factor	named_item
factor	(expression)
factor	**const**
const	NUMBER
const	DECIMAL
named_item	IDENTIFIER

7.3 Semantics

The CALC programming language is a very small and very focused programming languages. Most of the semantics associated with this language are well-known, almost second nature!

atomic data types

There are only two allowable data types for variables in the CALC programming language: INT and REAL.

A third data type, ASTRING, is available only for WRITELN statements and only as a literal constant.

precedence and associativity

Multiplication and division take precedence over addition and subtraction.

Equal precedence operators are performed left to right.

Parentheses may be used to override these defaults.

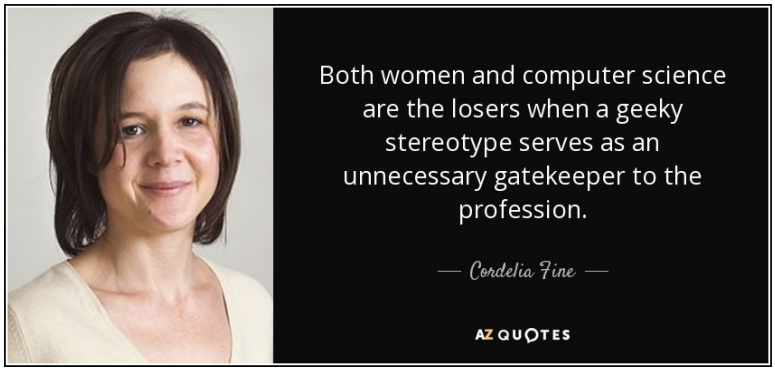
mixing data types

Mixed mode arithmetic follows traditional computing semantics – INT data values are **promoted** to REAL data values. This is done implicitly as part of the semantics of the programming language.

Assignment of a data value to a variable having the other data type will also perform an implicit **promotion** (INT to REAL) or **demotion** (REAL to INT).

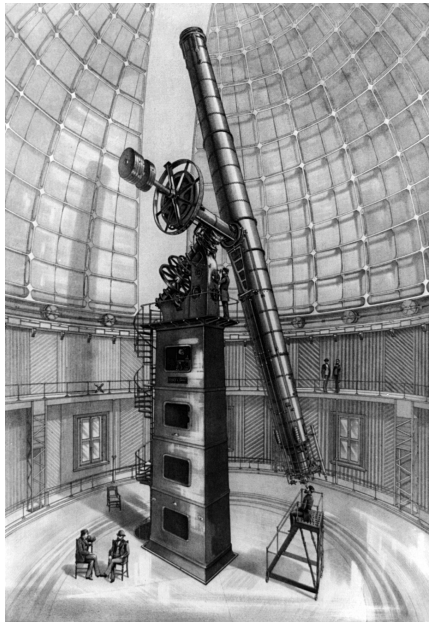
Chapter Ten: Compiling will focus on writing a program to verify both the syntax and the semantics for the CALC programming language source file and ultimately generate the appropriate assembly language code.

Chapter Eleven: The End Result will present all the **Icon** source code implementing our CALC compiler.


Both women and computer science are the losers when a geeky stereotype serves as an unnecessary gatekeeper to the profession.
— Cordelia Fine —
AZ QUOTES

Chapter 8

Scanning



The basic purpose of *scanning* (or *lexical analysis*, which is its more formal title) is to remove trivial time-consuming busy work from having to be encoded within the subsequent *parser* or *compiler*.

This chapter will be rather short because scanning is simple to implement. We introduce special terminology here to make it appear more important.

8.1 Basic Terminology

Definition 8.1.1. A **token name** is a basic component of a context free grammar that needs to be identified within the program source code.

In most general terms, basic categories might include items like: CONSTANT, IDENTIFIER, KEYWORD, OPERATOR, and PUNCTUATION. Some illustrative examples would be:

CONSTANT	12, 23.75, "paul kaiser"
IDENTIFIER	X, BALANCE, DISCRIMINANT
KEYWORD	PROGRAM, PROCEDURE, GLOBAL
OPERATOR	+ - * / :=
PUNCTUATION	: ; ( )

The examples listed above give rise to our next term.

Definition 8.1.2. A **lexeme** is a specific instance of a token name, that is, one of the possible **token value**s.

And this brings us to the final definition.

Definition 8.1.3. A **token** is a *pair* (token name,token value), that is, the token category together with its associated token value.

Example

Consider the following program fragment fairly common in most programming languages:

```
if ( a > 0 )
    then b := a + 5 ;
    else b := 2 ;
```

The input stream of individual ASCII characters would be grouped together appropriately and identified by token name. The resulting output fragment would look like:

TOKEN NAME	TOKEN VALUE
KEYWORD	if
PUNCTUATION	(
IDENTIFIER	a
OPERATOR	>
CONSTANT	0
PUNCTUATION	)
KEYWORD	then
IDENTIFIER	b
OPERATOR	:=
IDENTIFIER	a
OPERATOR	+
CONSTANT	5
PUNCTUATION	;
KEYWORD	else
IDENTIFIER	b
OPERATOR	:=
CONSTANT	2
PUNCTUATION	;

The granularity of tokens, (i.e., the number of categories specified) is an issue that should be considered. The number of token values that appear in the source code is an invariant integer value; hence, the output file will contain the same number of tokens, one per line. Whether we have 5 token categories or 50 token categories will not affect the length of the output file.

However, replacing a single generic token name with more specific names may facilitate the subsequent parsing or compiling process.

For example:

- CONSTANT might be further refined to NUMBER, DECIMAL, ASTRING
- IDENTIFIER is a pretty broad option with numerous lexemes, so leave it as is
- KEYWORD might be refined to the actual set of reserved words, e.g., PROGRAM, PROCEDURE, GLOBAL
- OPERATOR might be refined to the actual operator, e.g., + - * / :=
- PUNCTUATION might be refined to the actual punctuation character, e.g., : ; ( )

The addition of the above token categories results in the following more detailed output stream:

TOKEN NAME	**TOKEN VALUE**
IF	if
(	(
IDENTIFIER	a
>	>
NUMBER	0
)	)
THEN	then
IDENTIFIER	b
:=	:=
IDENTIFIER	a
+	+
NUMBER	5
;	;
ELSE	else
IDENTIFIER	b
:=	:=
NUMBER	2
;	;

The role of a scanner is to examine an input stream of individual ASCII characters in free form and identify the various token categories as they appear. Many tokens are typically one-character (like + or -) or two-character (like := or >=). Larger combinations are typically described by patterns (regular expressions) such as

$$\text{IDENTIFIER} \longrightarrow \{\ \text{A} \ldots \text{Z}\ \}\ \{\ \text{A} \ldots \text{Z}\ ,\ 0 \ldots 9\ \}\ *$$

which are also easily recognized in coding.

White spaces (blanks, tabs, newlines) are essentially ignored, except in situations where it is absolutely necessary between KEYWORDs and/or IDENTIFIERs.

It should be obvious from the above description that string manipulation would be a definite asset in your choice of programming language to implement your scanner. Java and C++ come immediately to mind. But I am sure there are many others as well, such as Perl and Python. Keep in mind that your implementation of a compiler will ultimately utilize a variety of data structures (stacks, tables, sets) as well.

In a previous lifetime, I implemented everything in C++! But only recently I decided to return to a programming language I encountered in graduate school at Illinois Institute of Technology – **Icon**.

Icon is a high-level programming language with extensive facilities for processing strings and structures. **Icon** has several novel features, including expressions that may produce sequences of results, goal-directed evaluation that automatically searches for a successful result, and string scanning that allows operations on strings to be formulated at a high conceptual level. **Icon** also provides high-level graphics facilities.

If interested in learning the **Icon Programming Language**, get a copy of the following text:

The Icon Programming Language, Third Edition
Ralph E. Griswold and Madge T. Griswold
Peer-to-Peer Communications, 1996, ISBN 1-57398-001-3

8.2 A Scanner for CALC

My **Icon** source code for the scanner may be found in **Chapter Eleven: The End Result**. It is comprised of a single file: **calcscanner.icn**.

to compile: $ icont calcscanner

to run: $ ./calcscanner < *infile* > *outfile*

Also note that I do **not** use file I/O for reading input and writing output. Rather I use **file redirection** (< and >). The parser and compiler for **CALC** will continue this practice, so that both parsing and compiling may be done in sequential order with scanning using the technique of **piping**.

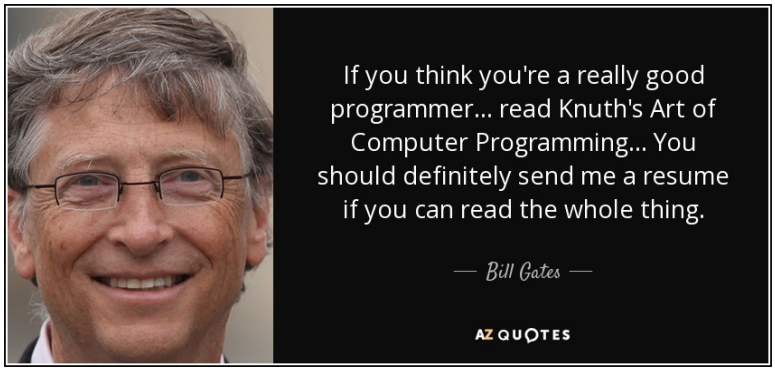
Ultimately, parsing will look like:

$ ./calcscanner < *infile* | ./calcparser

And, compiling will look like:

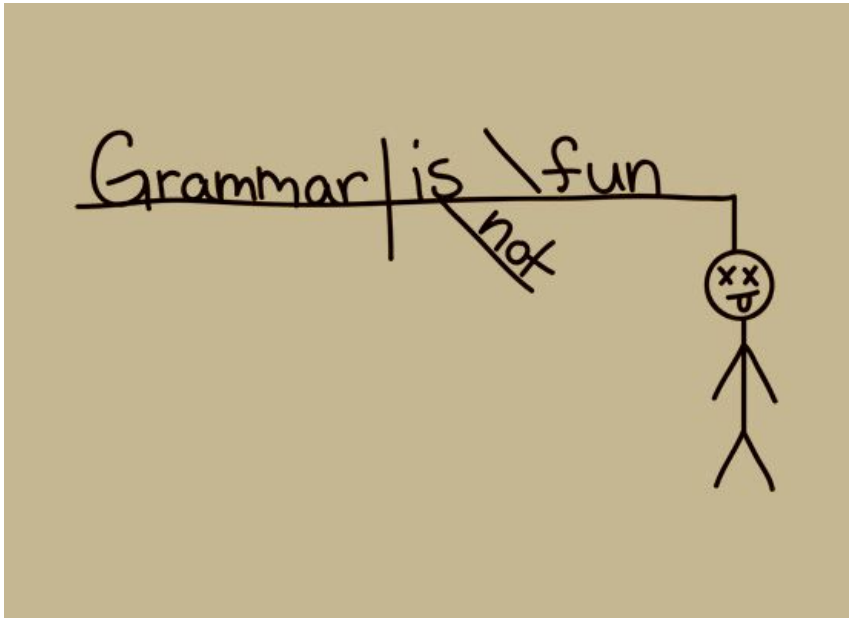
$ ./calcscanner < *infile* | ./calccompiler > *outfile*

An observant reader of my **Icon** code will quickly recognize that I have included an additional wrinkle to my scanner – the line number in the input file where the token pair appears. This was not part of our discussion. However, I chose to implement that feature to assist both the parser and the compiler in identifying in better detail **what** the error is and **where** the error occurred.


If you think you're a really good programmer... read Knuth's Art of Computer Programming... You should definitely send me a resume if you can read the whole thing.
— Bill Gates —
AZ QUOTES

Chapter 9

Parsing



Remember that parsing is not the same as compiling! Parsers are grammar checkers looking **only** for correct *syntax*. Parser output can be as simple as printing **yes** or **no** or, simpler yet, just terminating **EXIT_SUCCESS** or **EXIT_FAILURE**. The only form of error checking that needs to be considered is *syntax checking*.

Compilers augment the parsing process with semantics checking and code generation producing equivalent code in some lower level language. Compiler output is typically lengthy and may require additional manipulation (such as assembling and linking) to achieve executable code.

This chapter focuses on the issue of parsing; the next chapter will focus on the issues of compiling.

9.1 Context Free Grammars

We begin our discussion regarding parsing by continuing with our study of context free grammars in more detail.

Definition 9.1.1. A **context free grammar (CFG)** is a set of *symbols* Σ and a set of *productions rules* Π used to generate patterns of sequences comprised of the underlying symbols.

- productions rules have the form : $\alpha \longrightarrow \beta$
  where $\alpha = \sigma \in \Sigma$, a single symbol in Σ
  and $\beta = \sigma_1\sigma_2 \ldots \sigma_k, \sigma_i \in \Sigma, i = 1, 2, \ldots, k, k \geq 0$, a string of symbols
- non-terminal symbols N:
  any symbol $\sigma \in \Sigma$ which appears as the left-hand-side of a production rule $\mathrm{P} \in \Pi$
- terminal symbols T:
  any symbol $\sigma \in \Sigma$ which is not a non-terminal
  i.e, $T = \Sigma \sim \mathrm{N}$, hence $\mathrm{N} \cup \mathrm{T} = \Sigma$ and $\mathrm{N} \cap \mathrm{T} = \emptyset$
- start symbol S:
  special symbol in N which is the root of all derivations
- empty symbol ϵ
  special symbol in T which represents an empty substitution
  i.e., the non-terminal is simply removed without replacement

Definition 9.1.2. A **sentential form** is a string of symbols from Σ.

Definition 9.1.3. A **sentence** is a string of terminal symbols in T.

Definition 9.1.4. A **simple derivation** of sentential forms, denoted $s_0 \longrightarrow s_1$, is the application of a *single* production rule to *one* non-terminal symbol in s_0 yielding s_1.

Definition 9.1.5. A **derivation** of sentential forms, denoted $s_0 \longrightarrow s_k$, is the application of *a sequence* of production rules, $P_1, P_2, \ldots, P_k$ in Π

- $P_1 : s_0 \longrightarrow s_1$
- $P_2 : s_1 \longrightarrow s_2$
- $\vdots$
- $P_k : s_{k-1} \longrightarrow s_k$

Definition 9.1.6. A **complete derivation**, denoted $S \longrightarrow^*$ s, is a derivation of sentential forms $s_0 \longrightarrow s_k$, where $s_0 = S$ is the unique start symbol and $s_k = s$ is a sentence.

Definition 9.1.7. A **left-most derivation** of a sentence is built using the technique of always substituting a production rule for the left-most non-terminal found in a sentential form.

Definition 9.1.8. A **right-most derivation** of a sentence is built using the technique of always substituting a production rule for the right-most non-terminal found in a sentential form.

Recall that, in general, a derivation of a sentence may be done in any order – not just left-to-right or right-to-left. Hence, it is entirely possible that different substitution patterns might lead to different representations for the same sentence. Such a circumstance would signify that the context free grammar is **ambiguous**. We mention this possibility for completeness, but it is not central to our discussion!

Definition 9.1.9. The **language** of a CFG G, denoted L(G), is the set
$L(G) = \{$ sentence s $| \exists$ complete derivation S $\longrightarrow^*$ s $\}$

Recognizers for context free languages L(G) that are defined by a context free grammar typically use *non-deterministic push down automata (PDAs)*. The concept of non-deterministic machine allows for a variety of choices to be analyzed simultaneously. Unfortunately, such machines do not exist at the present moment! Rather, non-determinism must be simulated by following an individual choice and backtracking to the next option when necessary. Implementation of non-determinism, as you might expect introduces a high cost in terms of run-time efficiency. The good news is: there are **deterministic** techniques that can improve the efficiency of parsing.

Note: Regular Expressions (which we discussed in the previous chapter) are actually a subcategory of Context Free Grammars! We chose to discuss them separately because the technique for recognizing regular expressions is significantly simpler. Scanners for regular expressions are typically very straightforward computer programs; recognizers for context free languages require the use of much more powerful computing techniques.

9.2 Parsing Techniques

This section briefly describes two common approaches to building recognizers for context free languages: **LL(k) parsing** and **LR(k) parsing**. We briefly summarize the meaning for their cryptic names and describe in very simple terms their approaches to parsing.

L L (k) Parsing

L = left-to-right scanning of the input token stream
L = "build" a left-most derivation tree for the source code
k = number of symbols the parser may consider at one time

By looking only at the next symbol, an **LL(1)** parser will *either* match the **terminal symbol** found on the top of the parser stack *or* will know the appropriate production rule to substitute for the **non-terminal symbol** found on the top of the parser stack.

LL(1) parsing requires the use of *either* a simple stack *or* recursion (i.e., **recursive descent** parsing).

L R (k) Parsing

L = left-to-right scanning of the input token stream
R = "build" a right-most derivation tree for the source code, but in *reverse order*
k = number of symbols the parser may consider at one time

There are numerous flavors of **LR(k)** parsers based on the complexity of the grammar. All implementations of LR parsing use a simple stack, but require additional elements: *action* rules and *shift-reduce* rules. The parser considers the input stream token and the symbol at the top of the stack and determines whether to apply an *action* rule or a *shift-reduce* rule.

We will not consider LR parsing in any greater detail – that is best left for graduate school! We will focus instead on the parsing techniques appropriate to LL(1) parsing. Be advised that not every grammar is LL(1); however, very often a grammar for a given programming language may be rewritten in LL(1) form.

While in graduate school at Illinois Institute of Technology, I wrote several grammar analysis programs to identify the type of grammar under consideration and also its key components: NULL-DERIVING sets, FIRST sets, LAST sets, FOLLOW sets, and CHOICE or SELECT sets.

The following pages contain the definition for each of the above sets immediately followed by the output from my LL(1) grammar analysis program which presents the corresponding output generated for the context free grammar **CALC: Arithmetic Calculator**.

Definition 9.2.1. Null-deriving symbols are non-terminal symbols that may potentially generate the empty string.

grammar analysis

```
non-terminals:

     *start* addop assignment_statement atomic_type
     calc_program const executable_statement expression
     factor global_declarations more_factors more_terms
     more_var_list mulop named_item procedure read_item
     read_statement statement statement_list term
     var_declarations var_list var_type write_item
     write_statement

terminals:

     ( ) * *eof* + - / : := ; ASTRING BEGIN DECIMAL END
     IDENTIFIER INT NUMBER PROCEDURE PROGRAM READLN REAL
     VARIABLES WRITELN

null-deriving symbols:

     global_declarations  more_factors more_terms
     more_var_list statement_list var_declarations
```

Definition 9.2.2. First Sets identify terminal symbols that may potentially be the first symbol in a derivation of that non-terminal.

grammar analysis

```
<FIRST SETS>
*start* :
        PROGRAM calc_program
addop :
        + -
assignment_statement :
        IDENTIFIER named_item
atomic_type :
        INT REAL
calc_program : PROGRAM
const :
        DECIMAL NUMBER
executable_statement :
        IDENTIFIER READLN WRITELN assignment_statement
        named_item read_statement write_statement
expression :
        ( DECIMAL IDENTIFIER NUMBER const factor
        named_item term
factor :
        ( DECIMAL IDENTIFIER NUMBER const named_item
global_declarations :
        VARIABLES var_declarations
more_factors :
        * / mulop
more_terms :
        + - addop
more_var_list :
        IDENTIFIER var_list
mulop :
        * /
named_item :
        IDENTIFIER
procedure :
        PROCEDURE
read_item :
        IDENTIFIER named_item
read_statement :
        READLN
statement :
        IDENTIFIER READLN WRITELN assignment_statement
        executable_statement named_item read_statement
        write_statement
statement_list :
        IDENTIFIER READLN WRITELN assignment_statement
        executable_statement named_item read_statement
        statement write_statement
term :
```

```
        ( DECIMAL IDENTIFIER NUMBER const factor named_item
var_declarations :
        VARIABLES
var_list :
        IDENTIFIER
var_type :
        INT REAL atomic_type
write_item :
        ( ASTRING DECIMAL IDENTIFIER NUMBER const
        expression factor named_item term
write_statement :
        WRITELN
```

Definition 9.2.3. Last Sets identify terminal symbols that may potentially be the last symbol in a derivation of that non-terminal.

grammar analysis

```
<LAST SETS>
*start* :
        *eof*
addop :
        + -
assignment_statement :
        ;
atomic_type :
        INT REAL
calc_program :
        END
const :
        DECIMAL NUMBER
executable_statement :
        ; assignment_statement read_statement
        write_statement
expression :
        ) DECIMAL IDENTIFIER NUMBER const factor
        more_factors more_terms named_item term
factor :
        ) DECIMAL IDENTIFIER NUMBER const named_item
global_declarations :
        ; more_var_list var_declarations var_list
more_factors :
        ) DECIMAL IDENTIFIER NUMBER const factor
        more_factors named_item
more_terms :
        ) DECIMAL IDENTIFIER NUMBER const factor
        more_factors more_terms named_item term
more_var_list :
        ; more_var_list var_list
mulop :
        * /
named_item :
        IDENTIFIER
procedure :
        END
read_item :
        IDENTIFIER named_item
read_statement :
        ;
statement :
        ; assignment_statement executable_statement
        read_statement write_statement
statement_list :
        ; assignment_statement executable_statement
        read_statement statement statement_list
```

```
        write_statement
term :
        ) DECIMAL IDENTIFIER NUMBER const factor
                more_factors named_item
var_declarations :
        ; more_var_list var_list
var_list :
        ; more_var_list var_list
var_type :
        INT REAL atomic_type
write_item :
        ) ASTRING DECIMAL IDENTIFIER NUMBER const
        expression factor more_factors more_terms
        named_item term
write_statement :
        ;
```

Definition 9.2.4. Follow Sets identify terminal symbols that may potentially immediately follow a derivation of that non-terminal.

grammar analysis

```
<FOLLOW SETS>
addop :
        ( DECIMAL IDENTIFIER NUMBER
assignment_statement :
        END IDENTIFIER READLN WRITELN
atomic_type :
        ;
calc_program :
        *eof*
const :
        ) * + - / ;
executable_statement :
        END IDENTIFIER READLN WRITELN
expression :
        ) ;
factor :
        ) * + - / ;
global_declarations :
        BEGIN
more_factors :
        ) + - ;
more_terms :
        ) ;
more_var_list :
        BEGIN
mulop :
        ( DECIMAL IDENTIFIER NUMBER
named_item :
        ) * + - / := ;
procedure :
        END
read_item :
        )
read_statement :
        END IDENTIFIER READLN WRITELN
statement :
        END IDENTIFIER READLN WRITELN
statement_list :
        END
term :
        ) + - ;
var_declarations :
        BEGIN
var_list :
        BEGIN
var_type :
        ;
```

```
write_item :
        )
write_statement :
        END IDENTIFIER READLN WRITELN
```

Definition 9.2.5. Selection Sets are essentially functions: for a given non-terminal and a given input stream terminal it identifies the production rule to select and substitute.

grammar analysis

```
<SELECTION SETS>
*start* -> calc_program *eof*
  selected by: PROGRAM
addop -> +
  selected by: +
addop -> -
  selected by: -
assignment_statement -> named_item := expression ;
  selected by: IDENTIFIER
atomic_type -> INT
  selected by: INT
atomic_type -> REAL
  selected by: REAL
calc_program -> PROGRAM IDENTIFIER global_declarations
                BEGIN procedure END
  selected by: PROGRAM
const -> NUMBER
  selected by: NUMBER
const -> DECIMAL
  selected by: DECIMAL
executable_statement -> read_statement
  selected by: READLN
executable_statement -> write_statement
  selected by: WRITELN
executable_statement -> assignment_statement
  selected by: IDENTIFIER
expression -> term more_terms
  selected by: ( DECIMAL IDENTIFIER NUMBER
factor -> named_item
  selected by: IDENTIFIER
factor -> ( expression )
  selected by: (
factor -> const
  selected by: DECIMAL NUMBER
global_declarations -> var_declarations
  selected by: BEGIN VARIABLES
more_factors -> mulop factor more_factors
  selected by: * /
more_factors ->
  selected by: ) + - ;
more_terms -> addop term more_terms
  selected by: + -
more_terms ->
  selected by: ) ;
more_var_list -> var_list
```

```
  selected by: IDENTIFIER
more_var_list ->
  selected by: BEGIN
mulop -> *
  selected by: *
mulop -> /
  selected by: /
named_item -> IDENTIFIER
  selected by: IDENTIFIER
procedure -> PROCEDURE IDENTIFIER
             BEGIN statement_list END
  selected by: PROCEDURE
read_item -> named_item
  selected by: IDENTIFIER
read_statement -> READLN ( read_item ) ;
  selected by: READLN
statement -> executable_statement
  selected by: IDENTIFIER READLN WRITELN
statement_list -> statement statement_list
  selected by: IDENTIFIER READLN WRITELN
statement_list ->
  selected by: END
term -> factor more_factors
  selected by: ( DECIMAL IDENTIFIER NUMBER
var_declarations -> VARIABLES var_list
  selected by: VARIABLES
var_declarations ->
  selected by: BEGIN
var_list -> IDENTIFIER : var_type ; more_var_list
  selected by: IDENTIFIER
var_type -> atomic_type
  selected by: INT REAL
write_item -> expression
  selected by: ( DECIMAL IDENTIFIER NUMBER
write_item -> ASTRING
  selected by: ASTRING
write_statement -> WRITELN ( write_item ) ;
  selected by: WRITELN
```

If you are interested, my ancient grammar analysis programs may be found in the course resource material.

Files included: **ll1.icn**, **lr.icn,** and **grmr.icn** (core routines common to most forms of grammar analysis)

All the files are written in the programming language **Icon**.

to compile and run the ll1 grammar analysis program:

```
$ icont ll1 grmr
$ ll1 filename
```

to compile and run the lr grammar analysis program:

```
$ icont lr grmr
$ lr filename
```

The *filename* for the source grammar file above will be automatically augmented with the extension **.grm**; the extension should not be explicitly typed!

9.3 Recursive Descent Parsing

The primary advantage of an **LL(1)** grammar is that it knows **exactly** which *production rule* is appropriate simply by looking at the next item in the input token stream. Rather than implement an LL(1) parser using an explicit stack, we will hide the stack by using ***recursion***. We are not really eliminating the need for a stack – we are merely replacing **our stack** with the **system activation** stack.

The algorithm to develop a recursive descent parser for a specific LL(1) grammar is as follows:

- each **non-terminal symbol** in the grammar becomes a **procedure** in our parser
- the right-hand-side of each **production rule** essentially becomes the **executable code** for that procedure.
  - **a non-terminal** symbol in the right-hand side of a production rule becomes a **call** to that non-terminal's procedure
  - **a terminal** symbol in the right-hand-side of a production rule **must match** the token currently under consideration
- after processing a token in the input stream, the parser will simply **advance** to the next token
- **non-terminal symbols** having *multiple* **production rules** must use logic and the next token to select the correct rule.

We are now fully prepared to implement our first parser – a **CALC: Arithmetic Calculator parser**. We have all the tools: a functional scanner which transforms a steam of ASCII input into a stream of lexemes output, an LL(1) context free grammar for the **CALC** programming language, an understanding of how to implement LL(1) grammar parsing using the technique of recursive descent. We just have to put it all together.

Chapter Eleven: The End Result contains my **Icon** source code for such a parser, **calcparser.icn**. The parser utilizes two additional **Icon** files: **lexeme.icn** and **error.icn**. We will discuss each of the three files briefly after you take this opportunity to read them first.

9.4 A Parser for CALC

After reviewing the CALC Arithmetic Calculator, please return here to my description of the recursive descent parser. Each of the procedures following the main procedure represents a non-terminal in the context free grammar. Within each such procedure the right-hand-side of the production rules is implemented in Icon source code. For non-terminal symbols having multiple production rules, consideration of the current input symbol allows the program logic to decide the appropriate choice for the production rule to follow.

I have included the grammar production rules as comments within the **Icon** source code. This provides an opportunity for comparison between the elements in the context free grammar and its implementation. Hopefully, the simplicity and the ease of conversion will impress you as much as it impressed me.

9.4.1 Lexeme Support

lexeme.icn provide several support routines for the CALC parser.

The first two

procedure read_lexeme ()
procedure display_lexeme ()

are self-explanatory. However, I would direct the reader to the power of **Icon** in performing string analysis in relatively short code. That was my primary reason for moving away from C++ to **Icon** for the programming language of implementation.

In addition, **lexeme.icn** provides two additional procedures

procedure does_match (lex , target)
procedure must_match (lex , target)

which attempts to match the current lexeme in the input stream with a specific target type. The first flavor is a simple true / false test for matching a lexeme token type with a specific building block; this is very useful in selecting the appropriate production rule to pursue at any given point. The second flavor is a more demanding matching algorithm; this is very useful in matching specific terminal elements in the input stream.

9.4.2 Error Handling

Error handling techniques are best presented in a graduate level course. In an introductory course, the basic technique is simply to terminate the program gracefully while identifying the cause of the problem. A full-feature parser (or compiler) would normally make some attempt to resume processing as soon as possible after recognition of an error. However, this requires significant thought to properly implement.

For example, how deep from recursion must we backtrack ... or ... how much of the stack must we discard in order to resume the algorithm?

Many programmers like to see as many errors as possible to estimate to size of the task that lies ahead; many others prefer to find and correct the errors as they occur, one at a time. So a simple error handling technique should be sufficient for our work.

Our error handling routines are as basic as they can be!

For both a syntax error and a semantics error, the corresponding error handler should indicate the location where the error occurred – remember my addition of line numbers to the output of the scanner! If a syntax error has occurred, the mismatch is highlighted: what was **expected** versus what was **found**. If a semantics error has occurred, an elementary description of the problem is presented.

9.4.3 Generating the Parser

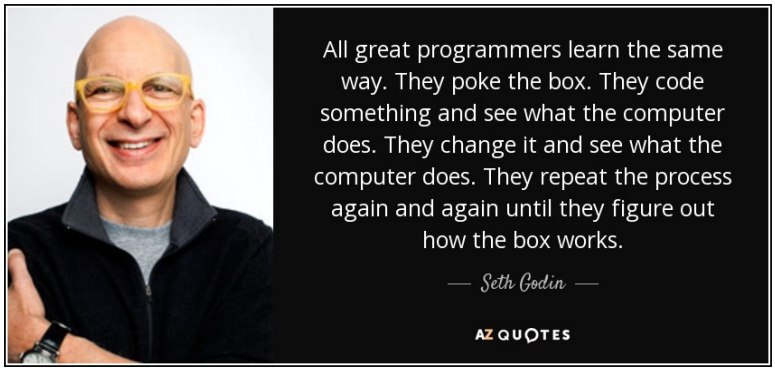
to compile the parser: $ icont calcparser lexeme error

to run the parser: $ ./calcparser < *infile*

to parse a source file (e.g., sample.calc):

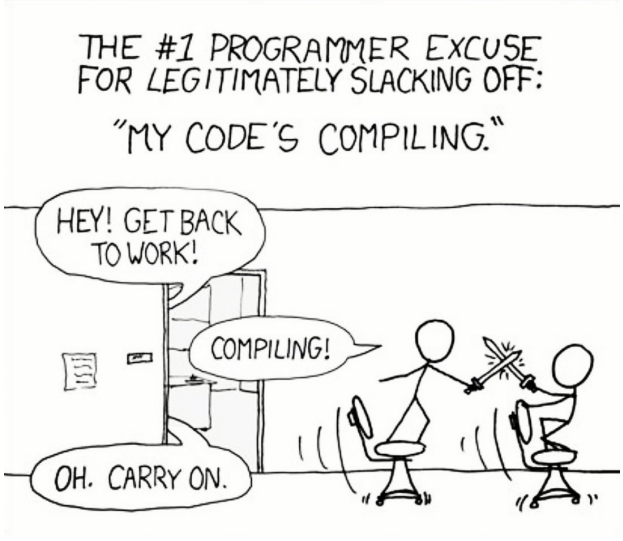
$ ./calcscanner < sample.calc | ./calcparser

$ ./calcscanner < sample.calc > sample.scn
$ ./calcparser < sample.scn


All great programmers learn the same way. They poke the box. They code something and see what the computer does. They change it and see what the computer does. They repeat the process again and again until they figure out how the box works.
— Seth Godin —
AZ QUOTES

Chapter 10

Compiling



We now begin to focus on three of the core issues for this portion of the text – maintaining a database for all the key elements in our programming language, the most fundamental concepts in semantics checking, and the generation of assembly language code which will implement our source code program. This is a very detail-intensive process, particularly with keeping track of identifiers, their role in the program, and their attributes. **Symbol tables** will be an essential component in the process.

It is important to realize at the onset that if a person is implementing a compiler, he or she will be typically approaching the task in an incremental fashion. At each incremental step the programmer will gain new insight into the problem and identify potential enhancements. These insights may require the programmer to rewrite entire sections of previously written code, possibly even filtering down into other coding files that so neatly organized his or her previous work. Such strict adherence to defined properties forces consistency and accuracy across potentially thousands of line of code! Finding subtle errors becomes a non-trivial task as we progress with our study of compiler construction.

I have tried to eliminate some of the frustrations that lie ahead by repeatedly writing and rewriting my own source code, by organizing and reorganizing my notes, and by attempting to write each of the chapters as complete and as self-contained as possible. However, some of the elements we discuss may not be fully understood and applicable until we discuss more high level topics, such as procedures, formal arguments, and structured data types. It is tempting to skip some of the implementation details until it is absolutely required. But I quickly learned that such practice forced me to rethink, repair, rewrite, and retest repeatedly.

The programming language **CALC: Arithmetic Calculator** provides a great opportunity to build a small compiler quickly and with little of the overhead of a full-featured compiler. It will provide the feeling of success at having converted source code into working assembly language code.

The programming language **kize** that lies waiting ahead will require significantly more patience and discipline on our part.

10.1 From Parser to Compiler

At this time I have some very good news. The basic template for our future CALC compiler already exists! It is called the CALC parser! We do not have to start from scratch when developing a compiler; we already have a solid foundation in our parser.

Our goal now is to augment the recursive descent parser with three additional components:

- creating a database of essential elements in the language
- checking for correctness in how these elements are combined
- generating code which properly reflects the source code

Our **error handling** must now include not only **syntax errors** but also **semantic errors**. Semantic error checking includes: *type checking*, *consistency of types*, *promotion / demotion of type*, *proper roles for identifiers*, etc.

These responsibilities forces us to maintain a database containing information regarding every identifier we incorporate into our source code program. This database will initially be very limited, essentially a single **symbol table** containing identifier, data type, and address information. Ultimately, we will require separate tables for everything: global variables, local variables, named types, procedures, and local labels!

Code generation requires that we have a specific target language in mind. **X86_64** and **AARCH64** both come to mind! And I certainly encourage students to experiment with one or other of these in the future. But as I mentioned in the Prologue, both languages suffer from stack alignment issues. I specifically designed KBOX and KCODE to eliminate this distraction. So my choice for this project is ... KCODE.

The following section presents several issues that will impact the conversion of our parser to a compiler. I believe it is vital that we see the big picture before we dive into the specific details of the implementation.

10.2 Issues to Consider

There are several issues that we should discuss prior to incorporating additional features into our parser source code. The following topics will definitely impact how we modify and enhance our existing source code.

10.2.1 Recursive Descent Technique

Recall that our recursive descent parsing algorithm utilized a collection of procedures:

procedure < ***proc_name*** > ( )

There is no communication between the various component procedures that comprise the parser. Since compilation is a more complicated process than parsing, these procedures will require more communication. We will now use the argument lists and the procedure return value for transferring information between the various components. Perhaps the single-most common piece of information that returns from any given operation is the **data type** of the result.

Therefore, I propose that we utilize the formal argument argument list and the return value in the obvious way: all **incoming** information be transferred in *the argument list*; all **outgoing** information be transferred in *the return value*. One very nice feature of the **Icon** programming language is that the return value can be anything! So even multiple values can be returned as a list.

As we implement our CALC compiler in the coming pages, you will note that the most common use of any return value will be to include the data type for the result.

10.2.2 Symbol Table

The first obvious consideration in moving from parser to compiler is introducing a database for information into the mix. Parsers do not need to remember any details they have previously seen; compilers do! Have I seen this token previously? If yes, is it being used consistently; if no, should it be incorporated into the database? Because the CALC Arithmetic Calculator is such a simple language,

we only need a single table in our database – a basic symbol table containing the identifier, its data type, and its location in memory. Any time we encounter an identifier in the CALC source code, we consult the symbol table and **retrieve** the information previously entered or we **create** a new entry.

Being a little bit more specific here, the identifier will be the string of ASCII characters that appears in the source code (commonly called its **external name**); the data type is limited to two categories (INT or REAL); and the location in memory might be either a positive integer address or a mnemonic (what is called its **internal name**).

It is fairly common in compilers to identify an external name with a unique internal name to avoid name collisions within the assembly language code. That is the approach that I will be taking in this text. One of the features that will appear in my assembly language component will be a unique ***name generator*** for internal names.

10.2.3 Type Checking

Although a bit redundant, it is important to remember the basic semantic rules regarding our two data types: INT and REAL.

- INT combined with INT yields INT
- REAL combined with REAL yields REAL
- mixed mode will promote an INT to a REAL
- assignment may promote an INT value to a REAL value to store
- assignment may demote a REAL value to an INT value to store

10.2.4 L-values versus R-values

Anytime we are manipulating variables, we must remain aware that the single word "variable" possibly refers to two distinct items: the *data value* associated with the variable name versus the *memory location* where that variable may be found. Consider the following assignment statement:

A := B + C * D ;

The items on the right-hand-side reference the actual data values for B, C, and D that make up the expression. These data values are typically referred to as **R-values**.

The item on the left-hand-side references the memory location for A, where the result should be stored. This memory location is typically referred to as an **L-value**.

It should be obvious that to recover an R-value for a given variable, one must first determine its L-value! The discussion becomes particularly convoluted when one introduces the concept of memory addresses **as** data – *indirect addressing*, *call by variable*, and eventually *pointers*.

10.2.5 Code Generation

At this point we turn our attention to issues specific to code generation – both general issues that apply to all assembly languages and issues specific to KCODE.

data segment

The data segment in assembly language defines the global static memory storage that resides in low memory throughout the entirety of program execution.

In KCODE the three directives _DATA_SEGMENT, _DEFINE, and _RESERVE are required to configure this information. Recall that the _DEFINE directive initializes storage with a data value; the _RESERVE directive does not.

Assembly languages also provide a variety of methods for accessing memory locations:

- direct addressing: go to the given address to retrieve data
- indirect addressing: go to the given address to find the actual address
- offset addressing: calculate the actual address as frame pointer plus offset

code segment

Code generation is the heart of a compiler. Our goal in this text will be to generate **functional code** – not necessarily fast code, not necessarily optimized code, but certainly code which works.

At this point in our studies we need only concern ourselves with direct addressing using memory which is statically declared within the data segment. So our initial discussion pertains to ***transferring data*** between memory and registers in KCODE. This is a very straightforward, but very specific process.

For an INT variable (with an internal name K15)

```
LDA SAR =K15
LDR IA SAR

LDA DAR =K15
STR IA DAR
```

For a REAL variable, the register FA would be substituted for the register IA in the above lines of code.

Note that Lvalues (addresses) should be moved into an address register – either a source address register (SAR) for retrieval or a destination address register (DAR) for storage. Only then can the actual data transfer occur using either a **LDR** instruction or a **STR** instruction.

A second issue pertains to bracketing our executable code with an introductory ***program prologue*** and a concluding ***program epilogue***.

The **program prologue** is necessary to transition from the _GLOBAL and the _EXTERN directives in the **code segment** into the actual executable code for the assembly language program. As discussed in the first part of this text, the _EXTERN directive is solely cosmetic and is only provided to parallel other assembly language components.

```
_CODE_SEGMENT
_GLOBAL MAIN
```

Similarly, the **program epilogue** transitions from the executable code into the **data segment** where static memory allocation is defined.

```
_DATA_SEGMENT
_RESERVE <var_name> 1
```

At this point it is important that I clarify one aspect of code generation in my implementation of the CALC compiler. Recall that the code segment and the data segment may appear in either order. I have chosen to generate the code segment first and postpone the data segment until the very end! That choice is reflected in the program prologue and the program epilogue above. However, as we progress through this text you will observe that I also display the contents of the symbol table (and any future database tables) within comment statements before generating executable code. Although this is definitely redundant with the CALC compiler, it will help in the future to maintain accuracy in the database and facilitate debugging.

Please note that at this time the only procedure we will be considering is the "main" procedure. All of our coding will be in a single location. Hence, program prologue and the program epilogue could serve as the only bracketing code we need to implement to introduce and to conclude our executable code.

However, in the future, after we discuss subprograms, we will have to consider additional bracketing code. A **procedure prologue** must properly set up the activation stack for each *call* to a subalgorithm in the program and save important register information. A **procedure epilogue** must restore the register information, save any return value, and successfully return back to the *calling* procedure.

I wanted to emphasize this requirement very early on; so I have included it as a third issue here. Even though it is a bit of overkill at this point, I chose to incorporate a procedure prologue and a procedure epilogue for the main procedure in the CALC compiler.

main procedure prologue

```
_LABEL MAIN
PUSH RP
PUSH FP
MOV FP SP
```

main procedure epilogue

```
_LABEL EXITMAIN
MOV SP FP
POP FP
POP RP
MOV IA ZR
RET
```

A fourth issue pertains to **input** and **output** operations within our executable code. Again, one of the nice features in KCODE is rudimentary read and write facilities (using GET and PUT). Remember that KCODE uses the top of the stack for both input and output – the data value read becomes the new top of the stack; the data value written comes from the top of the stack. KCODE input and output operations require specifying the appropriate formatting to be used.

Let us specifically focus on input for a moment. Reading an **integer** from the terminal and saving it in a variable X.

```
LDA SAR =X
PUSH SAR
GET INT
POP IA
POP DAR
STR IA DAR
```

A similar sequence of instructions would display the contents of a **real** variable Y to the terminal.

```
LDA SAR =Y
LDR FA SAR
PUSH FA
PUT REAL
PUTLN
```

It is important to remember two important elements of input and output:

- both GET and PUT require a formatting specification
- GET stores the input data value to the top of the stack
- PUT displays the data value at the top of the stack

A fifth and final issue pertains to **arithmetic expressions** and how to use the stack to facilitate calculations.

The stack is a very useful construct to take advantage of! Anytime you want to save a value for later use, *push it on the stack*! Anytime you want to retrieve a value to use it now, *pop it from the stack*!

Consider for a moment evaluating an arithmetic expression. For each variable in the expression we need to retrieve the data (into a register). Before we can combine it with other data values, we need to temporarily save it. So we push onto the stack. Suppose we now see an operator – we need to remember it. Now we need to retrieve the second piece of data (into a register) and, like the first piece of data, we temporarily save it on the stack. Only now are we capable of performing the desired operation.

The following sequence of instructions will probably become very familiar to you:

binary operation:

- pop second operand
- pop first operand
- promote one of the operands?
- perform operation
- push result

Please note that every single data value, whether from retrieving data from a variable or from a calculation, is put on the stack!

Now consider for a moment how to perform an assignment instruction.

We first need to determine the receiving variable (the target) and save its address (lvalue, not rvalue). We next need to recognize the assignment operator. And lastly, we need to evaluate the arithmetic expression. And where will this resulting data value reside? On the top of the stack of course!

The following sequence of instructions will also become very familiar to you:

assignment:

- pop expression value
- pop target address (lvalue)
- promote expression value? demote expression value?
- perform assignment operation

Assignment does not return anything to the top of the stack!

10.2.6 End-of-Phrase Markers

Now that we have identified some of the new elements to consider in converting a parser to a compiler, I would also like to suggest the incorporation of **end-of-phrase markers** into the grammar that defines the programming language.

End-of-phrase markers indicate key locations within the grammar where some appropriate action needs to be taken: either the storage / retrieval of information within the symbol tables; or type checking and promotion / demotion of data; or generation of assembly language code.

Although it is certainly possible to jump straight into writing a compiler without ever thinking about end-of-phrase markers, I consider it a useful device to help analyze all the details required in building the compiler. Unless you approach compiler construction in a most patient, organized, and thorough manner, you will find yourself rewriting a lot of code!

Been there ... done that.

Observe the augmented grammar for the CALC programming language that I utilized to implement the compiler.

calc with eop markers

```
calc_program            PROGRAM IDENTIFIER
                        !save_prog_name
                        global_declarations
                        !display_symbol_table
                        !emit_program_prologue
                        BEGIN procedure END
                        !emit_program_epilogue

global_declarations     GLOBAL var_declarations

var_declarations        VARIABLES var_list
var_declarations

var_list                IDENTIFIER !save_id_name :
                        var_type !save_id_type
                        !save_symbol_table_entry ;
                        more_var_list

more_var_list           var_list
more_var_list

var_type                atomic_type !return_atomic_type

atomic_type             INT !return_int
atomic_type             REAL !return_real

procedure               PROCEDURE IDENTIFIER
                        !save_proc_name
                        !emit_procedure_prologue
                        BEGIN statement_list END
                        !emit_procedure_epilogue

statement_list          statement statement_list
statement_list

statement               executable_statement

executable_statement    read_statement
executable_statement    write_statement
executable_statement    assignment_statement

read_statement          READLN ( read_item !save_type )
                        !emit_read ;

read_item               named_item !save_address

write_statement         WRITELN ( write_item !save_type )
                        !emit_write ;

write_item              expression
write_item              ASTRING !emit_constant
```

```
assignment_statement    named_item !save_address
                        := expression
                        !promote_demote !emit_assign ;

expression              term !emit_term more_terms

term                    factor !emit_factor more_factors

more_terms              addop !save_addop term
                        !emit_term !resolve_types
                        !emit_addop more_terms
more_terms

addop                   + !return_plus
addop                   - !return_minus

factor                  named_item !emit_rvalue
factor                  ( expression )
factor                  const

more_factors            mulop !save_mulop factor
                        !emit_factor !resolve_types
                        !emit_mulop more_factors
more_factors

mulop                   * !return_multiply
mulop                   / !return_divide

const                   NUMBER !emit_constant
const                   DECIMAL !emit_constant

named_item              IDENTIFIER
                        !retrieve_info !emit_lvalue
```

10.3 A Compiler for CALC

We are now fully prepared to implement our first compiler – a CALC compiler. We have all the tools: a functional scanner which transforms a steam of ASCII input into a stream of lexeme output, an LL(1) context free grammar for the CALC programming language, an understanding of how to implement LL(1) grammar parsing using the technique of recursive descent, and an overview of the additional elements we need to incorporate. We just have to put it all together.

Chapter Eleven: The End Result contain the **Icon** source code for my version of the compiler. The compiler utilizes files that we introduced previously for the parser: **lexeme.icn** and **error.icn**; and the compiler utilizes two additional files for symbol tables and code generation: **tables.icn** and **assembler.icn**.

We will discuss my source code in the following order:

- First, the file tables.icn because it is the simplest topic of the three!
- Second, the file calccompiler.icn because it will reinforce the importance of and the usefulness of end-of-phrase markers.
- Third, the file assembler.icn to highlight the advantage of keeping assembly language code separate from from the compiler code.

tables.icn

The first support file for our compiler is the simplest. It defines a basic symbol table holding information: the name of an identifier, its data type, and lastly its static address in memory. The static address is its unique internal name within KCODE. In addition to a symbol table, this support file also defines the contents of static memory as a simple list of define statements (which initializes memory locations) and reserve statements (which do not).

I prefer to display the contents of the symbol table prior to processing any executable code. This is done using numerous comment lines in KCODE. The code segment appears prior to the data segment in my compiler. The list of static memory elements is retained

until the last phase of the compiler when it is used to define the structure of main memory.

compiler.icn

As with the parser, I have included the grammar elements within its corresponding procedure. This time, however, I have included the end-of-phrase markers as well.

This should help you identify the additional coding that:

- stores and retrieves important data base information necessary for compilation
- performs semantic checking at key locations
- initiates code generation where suitable

You will obviously notice that actual code generation is done not within this compiler file but within the assembler file. I will talk about my decision to do this shortly in the next portion of the text. For the time being simply be aware that all those "emits" are generating assembly language code.

One last general observation before examining my source code in the next chapter. In the procedure **AssignmentStatement** you will note two end-of-phrase markers adjacent to one another: !promote_demote and !emit_assign. When we discussed the semantics for assignment statements, we noted that both the target location and the data value to be assigned must be the same data type. The data value may have to be promoted or demoted in its data representation to satisfy this requirement. Only then can the assignment transfer be performed. However, I chose to do nothing with regard to specifically promoting or demoting the data value. Instead I chose to incorporate that feature into the !emit_assign procedure in the assembler.

My point is: many aspects within compiler construction will overlap one another. Syntax and semantics may both address precedence of operators or associativity. Several end-of-phrase markers may identify tasks that need to be done. But the compiler designer / writer has to decide where and how to do them! End-of-phrase markers are simply a reminder that something needs to be done. They are not legally binding contracts!

assembler.icn

First off, let me answer the question of why I chose to generate explicit code in a new file **assembler.icn** here and not in the **calccompiler.icn** source code. Suppose in the near future you decide that you had so much fun writing a compiler for CALC in KCODE that we wanted to do the same thing for CALC in X86_64. All that would be necessary would be to update the assembler.icn file to use X86_64 to perform the same task that KCODE previously performed! Voila!! We have another compiler!!

By separating the specific and detailed assembly language code from the broader issues of the compiler, we have focused any adaptation or modification of the compiler to one file specific to the generation of assembly code.

With that having been said, allow me to clarify the organization and purpose of the various components in the assembler.icn file.

The first few procedures found in this file are intended to facilitate utilization of KCODE and the features available in the assembly language.

The procedure **initialize_assembler** sets up our special purpose registers prohibit simultaneous attempts to use the same register. The procedure also creates our name generator to create a sequence of unique internal names for our KCODE. The sequence is simply K0, K1, K2, ...

The next four procedures, **emit_blank_line**, **emit_comment**, **emit_label**, and **emit_line**, perform the actual code generation. The last procedure **emit_line** is the all-purpose component which includes an opcode, up to three operands, and an optional comment!)

Two procedures, **emit_program_prologue** and **emit_program_epilogue**, generate the KCODE program prologue and program epilogue respectively. And the two procedures, **emit_procedure_prologue** and **emit_procedure_epilogue**, similarly generate the KCODE procedure prologue and procedure epilogue respectively.

All of the items on the preceding page have been included to create a simple and user-friendly KCODE programming environment. The items which follow implement in KCODE specific tasks that are useful in implementing our compiler.

- **emit_Lvalue**: move the address of an identifier into the source address register
- **save_address**: save the address currently found in the source address register to the stack
- **emit_Rvalue**: retrieve the data value at the address currently in the source address register and save that data value to the stack
- **emit_constant**: move a constant value to the stack
- **emit_promote**: promote the data value found in an integer register and store the result in its corresponding floating point register
- **emit_demote**: demote the data value found in a floating point register and store the result in its corresponding integer register
- **emit_read**: read a data value from the keyboard and initially store it on the stack, then retrieve the data value and an address previously stored on the stack, and transfer the data to the specified location
- **emit_write**: display the data value currently found on the stack to the monitor
- **emit_get_operands**: retrieve the last two data values stored on the stack, promote one or the other as necessary, leave data values in their current registers
- **emit_addop**: perform the specified binary operation and save the result to the stack
- **emit_mulop**: perform the specified binary operation and save the result to the stack
- **emit_assign**: retrieve the data value and the target address from the stack, promote or demote the data value as required, store the result to the target address in memory

10.4 Testing our Compiler

to compile the compiler: $ icont calccompiler lexeme error tables assembler

to run the compiler: $ ./calccompiler < *infile* > *outfile*

to compile a source file (e.g., sample.calc):

$ ./calcscanner < sample.calc | ./calccompiler > sample.k

$ ./calcscanner < sample.calc > sample.scn
$ ./calccompiler < sample.scn > sample.k

To test whether compilation was successful:

$ ./calcscanner < sample.calc | ./calccompiler > sample.k

$ **kbox** sample.k

To provide these items with greater clarity, the following pages contain in order:

- a sample CALC source file (**sample.calc**)
- the output from the compiler (**sample.k**)
- the results from simulating execution on KBOX

sample.calc

```
program sample

  variables
    i      : int;
    j      : int;
    k      : int;
    dos      : int;
    p      : real;
    q      : real;
    r      : real;
    pi     : real;

begin

  procedure main

  begin
    writeln ("enter two integers:");
    readln (i);
    readln (j);
    writeln ("enter two reals:");
    readln (p);
    readln (q);
    writeln ("here are your values:");
    writeln (i);
    writeln (j);
    writeln (p);
    writeln (q);

    writeln ("now for some work!");
    dos := 2;
    pi := 3.1416;
    k := i + 2*j;
    r := p + 3.0*q;
    writeln ("dos");
    writeln (dos);
    writeln ("pi");
    writeln (pi);
    writeln ("i + 2*j");
    writeln (k);
    writeln ("p + 3.0*q");
    writeln (r);
    writeln ("dos*pi");
    writeln (dos*pi);
    k := pi;
    r := dos;
    writeln ("pi demoted");
    writeln (k);
    writeln ("dos promoted");
    writeln (r);
```

```
    writeln ("one horrendous mess");
    writeln ((i+j)*(p-q)-pi*(dos-i));
  end # main

end # program
```

sample.k

```
# SYMBOL TABLE
#     P                 REAL              K4
#     J                 INT               K2
#     Q                 REAL              K6
#     K                 INT               K3
#     R                 REAL              K7
#     DOS               INT               K4
#     I                 INT               K1
#     PI                REAL              K8

# BEGIN PROGRAM: SAMPLE

            _CODE_SEGMENT

            _GLOBAL           MAIN


# BEGIN PROCEDURE: MAIN

            _LABEL            MAIN
            push              I14
            push              I13
            mov               I13 I12

            movi              I0 "enter two integers:"
            push              I0
            put               STR
            putln
            lda               I4 =K1
            push              I4
            get               INT
            getln
            pop               I0
            pop               I5
            str               I0 I5
            lda               I4 =K2
            push              I4
            get               INT
            getln
            pop               I0
            pop               I5
            str               I0 I5
            movi              I0 "enter two reals:"
```

```
push            I0
put             STR
putln
lda             I4 =K5
push            I4
get             FLT
getln
pop             F0
pop             I5
str             F0 I5
lda             I4 =K6
push            I4
get             FLT
getln
pop             F0
pop             I5
str             F0 I5
movi            I0 "here are your values:"
push            I0
put             STR
putln
lda             I4 =K1
ldr             I0 I4
push            I0
put             INT
putln
lda             I4 =K2
ldr             I0 I4
push            I0
put             INT
putln
lda             I4 =K5
ldr             F0 I4
push            F0
put             FLT
putln
lda             I4 =K6
ldr             F0 I4
push            F0
put             FLT
putln
movi            I0 "now for some work!"
push            I0
put             STR
putln
lda             I4 =K4
push            I4
movi            I0 2
push            I0
nop
pop             I0
pop             I5
str             I0 I5
```

```
lda             I4 =K8
push            I4
movi            F0 3.1416
push            F0
nop
pop             F0
pop             I5
str             F0 I5
lda             I4 =K3
push            I4
lda             I4 =K1
ldr             I0 I4
push            I0
movi            I0 2
push            I0
lda             I4 =K2
ldr             I0 I4
push            I0
pop             I2
pop             I1
mul             I0 I1 I2
push            I0
pop             I2
pop             I1
add             I0 I1 I2
push            I0
nop
pop             I0
pop             I5
str             I0 I5
lda             I4 =K7
push            I4
lda             I4 =K5
ldr             F0 I4
push            F0
movi            F0 3.0
push            F0
lda             I4 =K6
ldr             F0 I4
push            F0
pop             F2
pop             F1
fmul            F0 F1 F2
push            F0
pop             F2
pop             F1
fadd            F0 F1 F2
push            F0
nop
pop             F0
pop             I5
str             F0 I5
movi            I0 "dos"
```

```
push            I0
put             STR
putln
lda             I4 =K4
ldr             I0 I4
push            I0
put             INT
putln
movi            I0 "pi"
push            I0
put             STR
putln
lda             I4 =K8
ldr             F0 I4
push            F0
put             FLT
putln
movi            I0 "i + 2*j"
push            I0
put             STR
putln
lda             I4 =K3
ldr             I0 I4
push            I0
put             INT
putln
movi            I0 "p + 3.0*q"
push            I0
put             STR
putln
lda             I4 =K7
ldr             F0 I4
push            F0
put             FLT
putln
movi            I0 "dos*pi"
push            I0
put             STR
putln
lda             I4 =K4
ldr             I0 I4
push            I0
lda             I4 =K8
ldr             F0 I4
push            F0
pop             F2
pop             I1
i2f             F1 I1
fmul            F0 F1 F2
push            F0
put             FLT
putln
lda             I4 =K3
```

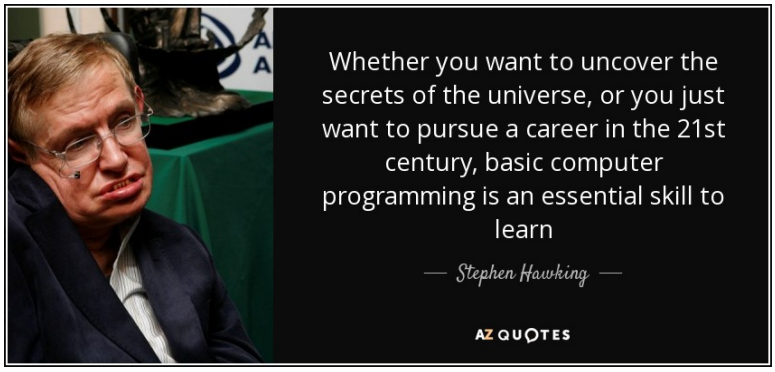
```
push            I4
lda             I4 =K8
ldr             F0 I4
push            F0
nop
pop             F0
f2i             I0 F0
pop             I5
str             I0 I5
lda             I4 =K7
push            I4
lda             I4 =K4
ldr             I0 I4
push            I0
nop
pop             I0
i2f             F0 I0
pop             I5
str             F0 I5
movi            I0 "pi demoted"
push            I0
put             STR
putln
lda             I4 =K3
ldr             I0 I4
push            I0
put             INT
putln
movi            I0 "dos promoted"
push            I0
put             STR
putln
lda             I4 =K7
ldr             F0 I4
push            F0
put             FLT
putln
movi            I0 "one horrendous mess"
push            I0
put             STR
putln
lda             I4 =K1
ldr             I0 I4
push            I0
lda             I4 =K2
ldr             I0 I4
push            I0
pop             I2
pop             I1
add             I0 I1 I2
push            I0
lda             I4 =K5
ldr             F0 I4
```

```
        push        F0
        lda         I4 =K6
        ldr         F0 I4
        push        F0
        pop         F2
        pop         F1
        fsub        F0 F1 F2
        push        F0
        pop         F2
        pop         I1
        i2f         F1 I1
        fmul        F0 F1 F2
        push        F0
        lda         I4 =K8
        ldr         F0 I4
        push        F0
        lda         I4 =K4
        ldr         I0 I4
        push        I0
        lda         I4 =K1
        ldr         I0 I4
        push        I0
        pop         I2
        pop         I1
        sub         I0 I1 I2
        push        I0
        pop         I2
        pop         F1
        i2f         F2 I2
        fmul        F0 F1 F2
        push        F0
        pop         F2
        pop         F1
        fsub        F0 F1 F2
        push        F0
        put         FLT
        putln

        _LABEL      EXITMAIN
        mov         I12 I13
        pop         I13
        pop         I14
        ret

# END PROCEDURE: MAIN


        _DATA_SEGMENT

        _RESERVE    K1 1
        _RESERVE    K2 1
        _RESERVE    K3 1
        _RESERVE    K4 1
```

```
        _RESERVE        K5  1
        _RESERVE        K6  1
        _RESERVE        K7  1
        _RESERVE        K8  1

# END PROGRAM: SAMPLE
```

sample.out

```
$> kbox sample.k

 ... do you wish to view setup (Y or N)?
 ... do you wish to view program execution (Y or N)?

enter two integers:
9
5
enter two reals:
23.5
11.5
here are your values:
9
5
23.500000
11.500000
now for some work!
dos
2
pi
3.141600
i + 2*j
19
p + 3.0*q
58.000000
dos*pi
6.283200
pi demoted
3
dos promoted
2.000000
one horrendous mess
189.991200

$>
```


Whether you want to uncover the secrets of the universe, or you just want to pursue a career in the 21st century, basic computer programming is an essential skill to learn
— Stephen Hawking —
AZ QUOTES

Chapter 11

The End Result

11.1 calcscanner

calcscanner.icn

```
# ---------------------------------------------------------- #

# calcscanner.icn
#
# this program is a basic lexical analyzer
# for the programming language calc
# it recognizes the basic building blocks:
#     - keywords
#     - identifiers
#     - literal           constants
#     - arithmetic        operators
#     - punctuation
#     - other             symbols
#
# input file may be any ASCII text file in free format
# output file will be an ASCII text file of triplets
#     - line number within the original input file
#     - token type  for a given lexeme
#     - token value for a given lexeme
#
# to generate the executable code:
#     $ icont calcscanner
#
# to execute the code, use IO redirection:
#     $ ./calcscanner < input_file > output_file

# ---------------------------------------------------------- #

global keywords       # set of keywords in calce
global symbols        # cset of single character symbols
global line           # a single line from input file
global line_count     # integer line counter

# ---------------------------------------------------------- #

procedure main ()

  keywords := set ([
    "ASTRING", "BEGIN", "DECIMAL", "END", "GLOBAL",
    "IDENTIFIER", "INT", "NUMBER", "PROCEDURE",
    "PROGRAM", "READLN", "REAL", "VARIABLES", "WRITELN"
  ])

  symbols := '()+-*/:=;#'

  line_count := 0
  while (line := read ()) do
  {
    line_count +:= 1
```

```
    process_tokens ()
  }
  exit ()
end

procedure process_tokens ()
  line ? repeat
  {
    tab (many (' '))
    if (pos(0)) then break
    if (any (&ucase++&lcase)) then
    {
      token_value := tab (many (&ucase++&lcase++&digits))
      token_value := map (token_value,&lcase,&ucase)
      if (member (keywords,token_value))
        then token_type := token_value
        else token_type := "IDENTIFIER"
    }
    else if (any (&digits)) then
    {
      token_value := tab (many (&digits))
      token_type := "NUMBER";
      if (any ('.')) then
      {
        token_value := token_value || "."
        move (1)
        token_type := "DECIMAL"
        if (any (&digits))
          then token_value ||:= tab (many (&digits))
          else stop ("invalid decimal encountered")
      }
    }
    else if (any ('\"')) then
    {
      move (1)
      token_value := tab (upto ('\"'))
      move (1)
      token_value := "\"" || token_value || "\""
      token_type  := "ASTRING"
    }
    else if (any ('#')) then
       return
    else if ("":="") then
      token_type := token_value := ":="
    else if (any (symbols)) then
      token_type := token_value := move (1)
    else
    {
      symbol := move(1)
      stop (left (line_count,7),
        "unexpected symbol encountered: ",symbol)
    }
    write (left (line_count,7),
```

```
        left (token_type,12),token_value)
    }
    return
end

# ------------------------------------------------------------ #
```

11.2 calcparser: the main program

calcparser.icn

```
# ---------------------------------------------------------- #

# calcparser.icn
#
# this program is a recursive descent parser
# for the programming language calc
#
# input file will be an ASCII text file of triplets
#     - line number within the original input file
#     - token type  for a given lexeme
#     - token value for a given lexeme
#
# to generate the executable code:
#     $ icont calcparser lexeme error
#
# to execute the code, use IO redirection and/or piping:
# for a previously scanned calc source file
#     $ ./calcparser < scanner_output
# for an unscanned calc source file
#     $ ./calcscanner < input_file | ./calcparser

# ---------------------------------------------------------- #

global current

procedure main ()

  current := read_lexeme ()
  CalcProgram ()
  return
end

# ---------------------------------------------------------- #

procedure CalcProgram ()
# calc_program --> PROGRAM IDENTIFIER
#                  global_declarations
#                  BEGIN procedure END

  must_match (current,"PROGRAM") &
    current := read_lexeme ()
  must_match (current,"IDENTIFIER") &
    current := read_lexeme ()
  GlobalDeclarations ()
  must_match (current,"BEGIN") &
    current := read_lexeme ()
  Procedure ()
  must_match (current,"END") &
    current := read_lexeme ()
```

```
end

# ------------------------------------------------------------ #

procedure GlobalDeclarations ()
# global_declarations --> var_declarations

  VarDeclarations ()
end

procedure VarDeclarations ()
# var_declarations --> VARIABLES var_list
# var_declarations -->

  if (not does_match (current,"VARIABLES")) then
    return
  current := read_lexeme ()
  VarList ()
end

procedure VarList ()
# var_list --> IDENTIFIER : var_type ; more_var_list

  must_match (current,"IDENTIFIER") &
    current := read_lexeme ()
  must_match (current,":") &
    current := read_lexeme ()
  VarType ()
  must_match (current,";") &
    current := read_lexeme ()
  MoreVarList ()
end

procedure MoreVarList ()
# more_var_list --> var_list
# more_var_list -->

  if (does_match (current,"BEGIN")) then
    return
  VarList ()
end

procedure VarType ()
# var_type --> atomic_type

  AtomicType ()
end

procedure AtomicType ()
# atomic_type --> INT
# atomic_type --> REAL

  if (does_match (current,"INT")) then
```

```
    current := read_lexeme ()
  else if (does_match (current,"REAL")) then
    current := read_lexeme ()
  else
    SemanticsError (current.line_no,
      "invalid atomic type specified (" +
      current.tok_value + ")")
end

# ------------------------------------------------------------ #

procedure Procedure ()
# procedure --> PROCEDURE IDENTIFIER
#               BEGIN statement_list END

  must_match (current,"PROCEDURE") &
    current := read_lexeme ()
  must_match (current,"IDENTIFIER") &
    current := read_lexeme ()
  must_match (current,"BEGIN") &
    current := read_lexeme ()
  StatementList ()
  must_match (current,"END") &
    current := read_lexeme ()
end

# ------------------------------------------------------------ #

procedure StatementList ()
# statement_list --> statement statement_list
# statement_list

  if (does_match (current,"END")) then
    return
  Statement ()
  StatementList ()
end

procedure Statement ()
# statement --> executable_statement

  ExecutableStatement ()
end

procedure ExecutableStatement ()
# executable_statement --> read_statement
# executable_statement --> write_statement
# executable_statement --> assignment_statement

  if (does_match (current,"READLN")) then
    ReadStatement ()
  else if (does_match (current,"WRITELN")) then
    WriteStatement ()
```

```
  else if (does_match (current,"IDENTIFIER")) then
    AssignmentStatement ()
  else
    SemanticsError (current.line_no,
      "unexpected executable statement (" +
      current.tok_value + ")")
end

# ------------------------------------------------------------ #

procedure ReadStatement ()
# read_statement --> READLN ( read_item ) ;

  must_match (current,"READLN") &
    current := read_lexeme ()
  must_match (current,"(") &
    current := read_lexeme ()
  ReadItem ()
  must_match (current,")") &
    current := read_lexeme ()
  must_match (current,";") &
    current := read_lexeme ()
end

procedure ReadItem ()
# read_item --> named_item

  NamedItem ()
end

procedure WriteStatement ()
# write_statement --> WRITELN ( write_item ) ;

  must_match (current,"WRITELN") &
    current := read_lexeme ()
  must_match (current,"(") &
    current := read_lexeme ()
  WriteItem ()
  must_match (current,")") &
    current := read_lexeme ()
  must_match (current,";") &
    current := read_lexeme ()
end

procedure WriteItem ()
# write_item --> expression
# write_item --> ASTRING

  if (does_match (current,"ASTRING")) then
    current := read_lexeme ()
  else
    Expression ()
end
```

```
# ------------------------------------------------------------ #

procedure AssignmentStatement ()
# assignment_statement --> named_item := expression ;

  NamedItem ()
  must_match (current,":=") &
    current := read_lexeme ()
  Expression ()
  must_match (current,";") &
    current := read_lexeme ()
end

procedure Expression ()
# expression --> term more_terms

  Term ()
  MoreTerms ()
end

procedure Term ()
# term --> factor more_factors

  Factor ()
  MoreFactors ()
end

procedure MoreTerms ()
# more_terms --> addop term more_terms
# more_terms -->

  if ((does_match (current,"+")) |
      (does_match (current,"-"))) then
  {
    AddOp ()
    Term ()
    MoreTerms
  }
end

procedure AddOp ()
# addop --> +
# addop --> -

  # current previously recognized as + or -!
  current := read_lexeme ()
end

procedure Factor ()
# factor --> named_item
# factor --> ( expression )
# factor --> const
```

```
  if (does_match (current,"IDENTIFIER")) then
    NamedItem ()
  else if (does_match (current,"(")) then
  {
    current := read_lexeme ()
    Expression ()
    must_match (current,")") &
      current := read_lexeme ()
  }
  else
    Const ()
end

procedure MoreFactors ()
# more_factors --> mulop factor more_factors
# more_factors -->

  if ((does_match (current,"*")) |
      (does_match (current,"/"))) then
  {
    MulOp ()
    Factor ()
    MoreFactors
  }
end

procedure MulOp ()
# mulop --> *
# mulop --> /

  # current previously recognized as * or /!
  current := read_lexeme ()
end

# ------------------------------------------------------------ #

procedure Const ()
# const --> NUMBER
# const --> DECIMAL

  if (does_match (current,"NUMBER")) then
    current := read_lexeme ()
  else if (does_match (current,"DECIMAL")) then
    current := read_lexeme ()
  else
    SemanticsError (current.line_no,
      "invalid constant ("+current.tok_value+")")
end

procedure NamedItem ()
# named_item --> IDENTIFIER
  must_match (current,"IDENTIFIER") &
```

```
    current := read_lexeme ()
end

# ------------------------------------------------------------ #
```

11.3 calccompiler: the main program

calccompiler.icn

```
# ---------------------------------------------------------- #

# calccompiler.icn
#
# this program is a recursive descent compiler
# for the programming language calc
#
# input file will be an ASCII text file of triplets
#     - line number within the original input file
#     - token type  for a given lexeme
#     - token value for a given lexeme
#
# to generate the executable code:
#     $ icont calccompiler lexeme error tables assembler
#
# to execute the code, use IO redirection and/or piping:
# for a previously scanned calc source file
#     $ ./calccompiler < scanned_file > output_file
# for an unscanned ac source file
#     $ ./calcscanner < input_file | \
#         ./calccompiler > output_file

# ---------------------------------------------------------- #

global current

procedure main ()

   initialize_structures ()
   initialize_assembler ()
   current := read_lexeme ()
   CalcProgram ()
   return
end

# ---------------------------------------------------------- #

procedure CalcProgram ()
# calc_program --> PROGRAM IDENTIFIER !save_prog_name
#                  global_declarations
#                  !display_symbol_table
#                  !emit_program_prologue
#                  BEGIN procedure END
#                  !emit_program_epilogue

   must_match (current,"PROGRAM") &
     current := read_lexeme ()
   must_match (current,"IDENTIFIER")
   # !save_prog_name
```

```
  prog_name := current.tok_value &
    current := read_lexeme ()
  GlobalDeclarations ()
  # !display_symbol_table
  display_symbol_table ()
  # !emit_program_prologue
  emit_program_prologue (prog_name)
  must_match (current,"BEGIN") &
    current := read_lexeme ()
  Procedure ()
  must_match (current,"END") &
    current := read_lexeme ()
  # !emit_program_epilogue
  emit_program_epilogue (prog_name)
end

# ------------------------------------------------------------ #

procedure GlobalDeclarations ()
# global_declarations --> var_declarations

  VarDeclarations ()
end

procedure VarDeclarations ()
# var_declarations --> VARIABLES var_list
# var_declarations -->

  if (not does_match (current,"VARIABLES")) then
    return
  current := read_lexeme ()
  VarList ()
end

procedure VarList ()
# var_list --> IDENTIFIER !save_id_name :
#              var_type !save_id_type
#              !save_symbol_table_entry ;
#              more_var_list

  must_match (current,"IDENTIFIER")
  # !save_id_name
  id_name := current.tok_value &
    current := read_lexeme ()
  must_match (current,":") &
    current := read_lexeme ()
  id_type := VarType ()
  # !save_id_type
  # !save_symbol_table_entry
  id_addr := @name_generator
  symbol_table [id_name] :=
    variable_entry(id_name,id_type,id_addr)
  put (static_memory,reserve_entry(id_addr,1))
```

```
  must_match (current,";") &
    current := read_lexeme ()
  MoreVarList ()
end

procedure MoreVarList ()
# more_var_list --> var_list
# more_var_list -->

  if (does_match (current,"BEGIN")) then
    return
  VarList ()
end

procedure VarType ()
# var_type --> atomic_type !return_atomic_type

  return AtomicType ()
end

procedure AtomicType ()
# atomic_type --> INT !return_int
# atomic_type --> REAL !return_real

  if (does_match (current,"INT")) then
  {
    current := read_lexeme ()
    return "INT"
  }
  else if (does_match (current,"REAL")) then
  {
    current := read_lexeme ()
    return "REAL"
  }
  else
    SemanticsError (current.line_no,
      "invalid atomic type specified ("+
      current.tok_value+")")
end

# ------------------------------------------------------------ #

procedure Procedure ()
# procedure --> PROCEDURE IDENTIFIER !save_proc_name
#               !emit_procedure_prologue
#               BEGIN statement_list END
#               !emit_procedure_epilogue

  must_match (current,"PROCEDURE") &
    current := read_lexeme ()
  must_match (current,"IDENTIFIER")
  # !save_proc_name
  proc_name := current.tok_value &
```

```
    current := read_lexeme ()
  # !emit_procedure_prologue
  emit_procedure_prologue (proc_name)
  must_match (current,"BEGIN") &
    current := read_lexeme ()
  StatementList ()
  must_match (current,"END") &
    current := read_lexeme ()
  # !emit_procedure_epilogue
  emit_procedure_epilogue (proc_name)
end

procedure StatementList ()
# statement_list --> statement statement_list
# statement_list

  if (does_match (current,"END")) then
    return
  Statement ()
  StatementList ()
end

procedure Statement ()
# statement --> executable_statement

  ExecutableStatement ()
end

procedure ExecutableStatement ()
# executable_statement --> read_statement
# executable_statement --> write_statement
# executable_statement --> assignment_statement

  if (does_match (current,"READLN")) then
    ReadStatement ()
  else if (does_match (current,"WRITELN")) then
    WriteStatement ()
  else if (does_match (current,"IDENTIFIER")) then
    AssignmentStatement ()
  else
    SemanticsError (current.line_no,
      "unexpected executable statement (" +
      current.tok_value + ")")
end

# ------------------------------------------------------------ #

procedure ReadStatement ()
# read_statement --> READLN ( read_item !save_type)
#                    !emit_read ;

  must_match (current,"READLN") &
    current := read_lexeme ()
```

```
  must_match (current,"(") &
    current := read_lexeme ()
  # !save_type
  read_type := ReadItem ()
  must_match (current,")") &
    current := read_lexeme ()
  must_match (current,";") &
    current := read_lexeme ()
  # !emit_read
  emit_read (read_type)
end

procedure ReadItem ()
# read_item --> named_item !save_address

  data_type := NamedItem ()
  # !save_address
  save_address ()
  return data_type
end

procedure WriteStatement ()
# write_statement --> WRITELN ( write_item !save_type )
#                     !emit_write ;

  must_match (current,"WRITELN") &
    current := read_lexeme ()
  must_match (current,"(") &
    current := read_lexeme ()
  # !save_type
  write_type := WriteItem ()
  must_match (current,")") &
    current := read_lexeme ()
  must_match (current,";") &
    current := read_lexeme ()
  # !emit_write
  emit_write (write_type)
end

procedure WriteItem ()
# write_item --> expression
# write_item --> ASTRING !emit_constant

  if (does_match (current,"ASTRING")) then
  {
    const_value := current.tok_value &
      current := read_lexeme ()
    # !emit_constant
    emit_constant ("STR",const_value)
    return "STR"
  }
  else
    return Expression ()
```

```
end

# ------------------------------------------------------------ #

procedure AssignmentStatement ()
# assignment_statement --> named_item !save_address
#                          := expression !promote_demote
#                          !emit_assign ;

  var_type := NamedItem ()
  # !save_address
  save_address ()
  must_match (current,":=") &
    current := read_lexeme ()
  expr_type := Expression ()
  must_match (current,";") &
    current := read_lexeme ()
  # !promote_demote
  #   is built in to the emit_assign code
  # !emit_assign
  emit_assign (var_type,expr_type)
end

procedure Expression ()
# expression --> term_term more_terms

  data_type := Term ()
  return MoreTerms (data_type)
end

procedure Term ()
# term --> factor more_factors

  data_type := Factor ()
  return MoreFactors (data_type)
end

procedure MoreTerms (b_type)
# more_terms --> addop !save_addop term
#                !resolve_types !emit_addop
#                more_terms
# more_terms -->

  if ((does_match (current,"+")) |
      (does_match (current,"-"))) then
  {
    # !save_addop
    op := AddOp ()
    c_type := Term ()
    # !resolve_types
    if ((b_type == "REAL") |
        (c_type == "REAL")) then
      a_type := "REAL"
```

```
    else
      a_type := "INT"
    # !emit_addop
    emit_addop (op,a_type,b_type,c_type)
    return MoreTerms (a_type)
  }
  else
    return b_type
end

procedure AddOp ()
# addop --> + !return_plus
# addop --> - !return_minus

  # current previously recognized as + or -!
  result := current.tok_value &
    current := read_lexeme ()
  return result
end

procedure Factor ()
# factor --> named_item !emit_rvalue
# factor --> ( expression )
# factor --> const

  if (does_match (current,"IDENTIFIER")) then
  {
    data_type := NamedItem ()
    # !emit_rvalue
    emit_Rvalue (data_type)
  }
  else if (does_match (current,"(")) then
  {
    current := read_lexeme ()
    data_type := Expression ()
    must_match (current,")") &
      current := read_lexeme ()
  }
  else
    data_type := Const ()
  return data_type
end

procedure MoreFactors (b_type)
# more_factors --> mulop !save_mulop factor
#                  !resolve_types !emit_mulop
#                  more_factors
# more_factors -->

  if ((does_match (current,"*")) |
      (does_match (current,"/"))) then
  {
    # !save_mulop
```

```
    op := MulOp ()
    c_type := Factor ()
    # !resolve_types
    if ((b_type == "REAL") |
        (c_type == "REAL")) then
      a_type := "REAL"
    else
      a_type := "INT"
    # !emit_mulop
    emit_mulop (op,a_type,b_type,c_type)
    return MoreFactors (a_type)
  }
  else
    return b_type
end

procedure MulOp ()
# mulop --> * !return_multiply
# mulop --> / !return_divide

  # current previously recognized as * or /!
  result := current.tok_value &
    current := read_lexeme ()
  return result
end

# ------------------------------------------------------------ #

procedure Const ()
# const --> NUMBER !emit_rvalue
# const --> DECIMAL !emit_rvalue

  if (does_match (current,"NUMBER")) then
  {
    const_value := current.tok_value &
      current := read_lexeme ()
    emit_constant ("INT",const_value)
    return "INT"
  }
  else if (does_match (current,"DECIMAL")) then
  {
    const_value := current.tok_value &
      current := read_lexeme ()
    emit_constant ("REAL",const_value)
    return "REAL"
  }
  else
    SemanticsError (current.line_no,
      "invalid constant (" + current.tok_value + ")")
end

procedure NamedItem ()
# named_item --> IDENTIFIER !retrieve_info !emit_lvalue
```

```
  must_match (current,"IDENTIFIER")
  # !retrieve_info
  identifier := current.tok_value &
    current := read_lexeme ()
  info := symbol_table[identifier]
  data_type := info.data_type
  kbox_addr := info.address
  # !emit_lvalue
  emit_Lvalue (kbox_addr)
  return data_type
end

# ------------------------------------------------------------ #
```

11.3.1 buildit script

buildit

```
#!/bin/bash

icont calccompiler \
      lexeme error tables assembler
```

11.4 lexeme: basic building blocks

lexeme.icn

```
# ---------------------------------------------------------- #

# lexeme.icn

# read an input file generated by the scanner
# - line number
# - token type
# - token value

# support routines
# - read_lexeme: primary component of this file!
# - display_lexeme: useful for debugging (if necessary)
# - does_match: does token type match a given target

# ---------------------------------------------------------- #

record lexeme (line_no,tok_type,tok_value)

# ---------------------------------------------------------- #

procedure read_lexeme ()
  line := read () | fail
  line ?
  {
    line_no   := tab (upto (' '))
    tab (many (' '))
    tok_type  := tab (upto (' '))
    tab (many (' '))
    tok_value := tab (0)
  }
  result := lexeme (line_no,tok_type,tok_value)

# uncomment the following line to facilitate debugging
#  display_lexeme (result)

  return result
end

procedure display_lexeme (current)
  write (left (current.line_no,7),
         left (current.tok_type,12),
         current.tok_value)
end

# ---------------------------------------------------------- #

procedure does_match (lex,target)
  return (lex.tok_type == target)
end
```

```
procedure must_match (lex,target)
  if (lex.tok_type ~== target) then
    SyntaxError (current.line_no,target,lex.tok_type)
  else
    return
end

# ---------------------------------------------------------- #
```

11.5 error: error handling

error.icn

```
# ---------------------------------------------------------- #

# error.icn

# simple error handler ("one and done")
# - syntax error is a mismatch
#     between the current token and what grammar expects
# - semantics error is an inability to implement
#     the desired operation as defined by the language

# ---------------------------------------------------------- #

procedure SyntaxError (line_no,expected,found)
  stop ("Syntax Error (line " || line_no ||
        "): " || "expected (" || expected ||
        "), found (" || found || ")")
end

procedure SemanticsError (line_no,description)
  stop ("Semantics Error (line " || line_no||
        "): " || description)
end

# ---------------------------------------------------------- #
```

11.6 tables: symbol table

tables.icn

```
# ---------------------------------------------------------- #

# tables.icn

# defines the data structures to support the calc compiler
# - symbol table
# - static memory

# support routes for the data structures
# - initialize structures
# - display symbol table
# - setup static memory

# ---------------------------------------------------------- #

record variable_entry   (identifier,data_type,address)
record define_entry     (identifier,value)
record reserve_entry    (identifier,size)

global symbol_table
global static_memory

# ---------------------------------------------------------- #

procedure initialize_structures ()
  symbol_table := table ()
  static_memory := []
end

# ---------------------------------------------------------- #

procedure display_symbol_table ()
  emit_comment("SYMBOL TABLE")
  every (entry := !symbol_table) do
    write (left ( "#",5),
           left (entry.identifier,15),
           left (entry.data_type,15),
           left (entry.address,10))
  emit_blank_line ()
end

# ---------------------------------------------------------- #

procedure setup_static_memory ()
  every entry := !static_memory do
    if (type(entry) == "define_entry") then
      emit_line ("_DEFINE",entry[1],entry[2])
    else # (type(entry) == "reserve_entry")
      emit_line ("_RESERVE",entry[1],entry[2])
```

```
end

# ———————————————————————————————————————— #
```

11.7 assembler: code generation

assembler.icn

```
# ---------------------------------------------------------- #

# assembler.icn

# a collection of procedures to aid in
# generating kcode assembly language

# ---------------------------------------------------------- #

global  IA,IB,IC,ID,SAR,DAR,OR,IR,ZR,FP,SP,RP,HP
global  FA,FB,FC,FD
global  name_generator

# ---------------------------------------------------------- #

procedure initialize_assembler ()
# procedure to initialize register sets
# and to create the name generator

  IA        := "I0"         # integer registers
  IB        := "I1"
  IC        := "I2"
  ID        := "I3"

  SAR       := "I4"         # source      address register
  DAR       := "I5"         # destination address register
  OR        := "I6"         # offset register
  IR        := "I7"         # index  register

  ZR        := "I11"        # zero   register
  SP        := "I12"        # frame  pointer
  FP        := "I13"        # stack  pointer
  RP        := "I14"        # return pointer
  HP        := "I15"        # heap   pointer

  FA        := "F0"         # floating point registers
  FB        := "F1"
  FC        := "F2"
  FD        := "F3"

  name_generator          := create ("K"||seq())
end

# ---------------------------------------------------------- #

procedure emit_blank_line ()
# generate a blank line within the kbox assembly code

  write ()
```

```
end

procedure emit_comment (message)
# insert a single comment line in the kbox assembly code

  write ("# ",message)
end

procedure emit_label (label)
# insert an internal label name into the kbox assembly code

  emit_line ("_LABEL",label)
end

procedure emit_line (opcode,oper1,oper2,oper3,comment)
# generate a single line of code in the kbox assembly code

  writes (left ("",10))
  writes (left (opcode,15))
  if (\oper1) then
  {
    ops := oper1
    if (\oper2) then
    {
      ops := ops||" "||oper2
      if (\oper3) then
        ops := ops||" "||oper3
    }
  }
  else
    ops := ""
  if (opcode == "_DEFINE") then
    writes (ops)
  else
    writes (left (ops,30))
  if (/comment) then
    write ()
  else
    write ("# "||comment)
end

# ---------------------------------------------------------- #

procedure emit_program_prologue (prog_name)
# emit kbox assembly code to:
#   - define text section
#   - identify entry point "main" for kbox assembly code

  emit_comment ("BEGIN PROGRAM: "||prog_name)
  emit_blank_line ()
  emit_line ("_CODE_SEGMENT")
  emit_blank_line ()
  emit_line ("_GLOBAL","MAIN")
```

```
  emit_blank_line ()
end

procedure emit_program_epilogue (prog_name)
# emit kbox assembly code to:
#   - define data section
#   - display
#     . symbol table
#     . static memory

  emit_blank_line ()
  emit_line ("_DATA_SEGMENT")
  emit_blank_line ()
  setup_static_memory ()
  emit_blank_line ()
  emit_comment ("END PROGRAM: " || prog_name)
  emit_blank_line ()
end

# ------------------------------------------------------------ #

procedure emit_procedure_prologue (proc_name)
# emit kbox assembly code to:
#   - display new procedure name as a comment
#   - set up a new activation stack item
#     - save return address
#     - save old frame pointer
#     - move current stack pointer to new frame pointer

# remember that the "caller" has the responsibility
#   - evaluate the actual arguments
#   - push appropriate information onto activation stack
#   - prior to transfer of control

  emit_blank_line ()
  emit_comment ("BEGIN PROCEDURE: " || proc_name)
  emit_blank_line ()
  emit_label (proc_name)
  emit_line ("push",RP,,,"save the RP to stack")
  emit_line ("push",FP,,,"save the FP to stack")
  emit_line ("mov",FP,SP,,"current SP become the new FP")
  emit_blank_line ()
end

procedure emit_procedure_epilogue (proc_name)
# emit kbox assembly code to:
#   - display procedure name as a comment
#   - clean up obsolete activation stack item
#     - retrieve old stack pointer from frame pointer
#     - retrieve old frame pointer
#     - retrieve return address

  emit_blank_line ()
```

```
  emit_label ("EXIT"||proc_name)
  emit_line ("mov",SP,FP,,
    "retrieve old SP from current FP")
  emit_line ("pop",FP,,,"retrieve FP from stack")
  emit_line ("pop",RP,,,"retrieve RP from stack")
  emit_line ("ret",,,,,"return to calling procedure")
  emit_blank_line ()
  emit_comment ("END PROCEDURE: "||proc_name)
  emit_blank_line ()
end

# ------------------------------------------------------------ #

procedure emit_Lvalue (addr)
# emit kbox assembly code to:
# move the address of the identifier
# into SAR

  emit_line ("lda",SAR,"="||addr,,"Lvalue -> SAR")
end

procedure save_address ()
# emit kbox assembly code to:
# save the address found in SAR
# to the top of the stack

  emit_line ("push",SAR,,,"SAR -> stack")
end

procedure emit_Rvalue (data_type)
# emit kbox assembly code to:
# retrieve the data value found at the memory address
# currently found in SAR
# and push the data value onto the top of the stack

  if (data_type == "INT") then
  {
    emit_line ("ldr",IA,SAR,,"Rvalue -> stack")
    emit_line ("push",IA)
  }
  else # (data_type == "REAL")
  {
    emit_line ("ldr",FA,SAR,,"Rvalue -> stack")
    emit_line ("push",FA)
  }
end

procedure emit_constant (const_type,const_value)
# emit kbox assembly code to:
# move a constant value to the top of the stack

  if (const_type == "INT") then
  {
```

```
    emit_line ("movi",IA,const_value,,"int constant")
    emit_line ("push",IA)
  }
  else if (const_type == "REAL") then
  {
    emit_line ("movi",FA,const_value,,"real constant")
    emit_line ("push",FA)
  }
  else # (const_type == "ASTRING")
  {
    emit_line ("movi",IA,const_value,,"string constant")
    emit_line ("push",IA)
  }
end

# ------------------------------------------------------------ #

procedure emit_promote (reg)
  if (reg[1] == "I") then
  {
    dest := "F" || reg[2:0]
    emit_line ("i2f",dest,reg,,"promote")
  }
  else
    stop ("invalid int register ("||reg||")")
end

procedure emit_demote (reg)
  if (reg[1] == "F") then
  {
    dest := "I" || reg[2:0]
    emit_line ("f2i",dest,reg,,"demote")
  }
  else
    stop ("invalid real register ("||reg||")")
end

# ------------------------------------------------------------ #

procedure emit_read (data_type)
# emit kbox assembly code to:
#   - read value from the keyboard
#   - store value on the top of the stack
#   - data type determines fmt

  if (\data_type) then
  {
    if (data_type == "INT") then
      fmt := "INT"
    else
      fmt := "FLT"
    emit_line ("get",fmt,,,"READLN")
    emit_line ("getln")
```

```
    if (data_type == "INT") then
      reg := IA
    else # (data_type == "REAL")
      reg := FA
    emit_line ("pop",reg,,,"store input value")
    emit_line ("pop",DAR)
    emit_line ("str",reg,DAR)
  }
end

# ------------------------------------------------------------ #

procedure emit_write (data_type)
# emit kbox assembly code to:
#   - write value to the monitor
#   - value is found on the top of the stack
#   - data type determines fmt

  if (\data_type) then
  {
    if (data_type == "INT") then
      fmt := "INT"
    else if (data_type == "REAL") then
      fmt := "FLT"
    else
      fmt := "STR"
    emit_line ("put",fmt,,,"WRITELN")
    emit_line ("putln")
  }
end

# ------------------------------------------------------------ #

procedure emit_get_operands (a_type,b_type,c_type)
# emit kbox code to retrieve a two operands
#   from the top of the stack
# b_type and c_type determine registers for the data
#   and a_type determines any promotion

  if (c_type == "INT") then
    emit_line ("pop",IC,,,"get operands")
  else # (c_type == "REAL")
    emit_line ("pop",FC,,,"get operands")
  if (b_type == "INT") then
    emit_line ("pop",IB)
  else # (b_type == "REAL")
    emit_line ("pop",FB)
  if ((c_type == "INT") & (a_type == "REAL")) then
    emit_promote (IC)
  if ((b_type == "INT") & (a_type == "REAL")) then
    emit_promote (IB)
end
```

```
# ------------------------------------------------------------ #

procedure emit_addop (op,a_type,b_type,c_type)
# emit kbox code to add/subtract two data values
# a_type, b_type, and c_type determine registers

   emit_get_operands (a_type,b_type,c_type)
   if (a_type == "INT") then
   {
      if (op == "+") then
         inst := "add"
      else # (op == "-")
         inst := "sub"
      emit_line (inst,IA,IB,IC," perform addop "||op)
      emit_line ("push",IA)
   }
   else if (a_type == "REAL") then
   {
      if (op == "+") then
         inst := "fadd"
      else # (op == "-")
         inst := "fsub"
      emit_line (inst,FA,FB,FC," perform addop "||op)
      emit_line ("push",FA)
   }
   else
      stop ("invalid arithmetic data type ("||a_type||")")
end

procedure emit_mulop (op,a_type,b_type,c_type)
  # emit kbox code to multiply/divide two data values
  # a_type, b_type, and c_type determine registers

   emit_get_operands (a_type,b_type,c_type)
   if (a_type == "INT") then
   {
      if (op == "*") then
         inst := "mul"
      else # (op == "/")
         inst := "div"
      emit_line (inst,IA,IB,IC," perform mulop "||op)
      emit_line ("push",IA)
   }
   else if (a_type == "REAL") then
   {
      if (op == "*") then
         inst := "fmul"
      else # (op == "/")
         inst := "fdiv"
      emit_line (inst,FA,FB,FC," perform mulop "||op)
      emit_line ("push",FA)
   }
   else
```

```
    stop ("invalid arithmetic data type ("||a_type||")")
end

# ------------------------------------------------------------ #

procedure emit_assign (target_type,expr_type)
# emit kbox code to assign data value at top of stack
#   to target location also found at top of stack
# target_type specifies the data type for the transfer

  emit_line ("nop",,,,"assignment")
  if (target_type == "INT") then
  {
    if (expr_type == "REAL") then
    {
      emit_line ("pop",FA)
      emit_demote (FA)
    }
    else # (expr_type == "INT")
      emit_line ("pop",IA)
    emit_line ("pop",DAR)
    emit_line ("str",IA,DAR)
  }
  else # (target_type == "REAL")
  {
    if (expr_type == "INT") then
    {
      emit_line ("pop",IA)
      emit_promote (IA)
    }
    else # (expr_type == "REAL")
      emit_line ("pop",FA)
    emit_line ("pop",DAR)
    emit_line ("str",FA,DAR)
  }
end

# ------------------------------------------------------------ #
```


Coding is today's language of creativity. All our children deserve a chance to become creators instead consumers of computer science.
— Maria Klawe —
AZ QUOTES

Chapter 12

Phase Zero: We Begin

Johnny Depp Begins Work on His KIZE Compiler

Fun With Programming Languages

We begin by presenting a new programming language that is not trivial – like the CALC programming language that we just discussed. It as not so overwhelming as some popular modern languages – like C++ or Java. It is a basic educational programming language that I originally called **klump** in recognition of my colleague and very close friend, Dr. Ray Klump, Professor at Lewis University. But I decided after two years of abusing my students with many very bad lectures that I should change the name to **kize** and take some ownership for my creation (ala Dr. Frankenstein).

Having previously considered the basic components in implementing an CALC Arithmetic Calculator, we already have a very good idea of what needs to be done initially.

- define the building blocks for the language
- present the context free grammar for the language
- identify semantic rules for the language

We address each of these elements in the sections immediately following.

12.1 KIZE: Basic Building Blocks

The building blocks for the **kize** programming language are very similar to those of the CALC programming language. It was always my intention that the introductory small programming language was to be a subset of this larger programming language. This was to minimize any necessary modifications to the scanner software – just add a few more operators, insert a few more keywords, and include a bit more punctuation!

Variables in this language remain simple: { A ... Z } { A ... Z , 0 ... 9 } *

an initial alphabetic character

followed by any number of alphanumeric characters

variable names are **not** case sensitive!

Constants in this language now come in three flavors:

NUMBER: { 0 ... 9 } +

DECIMAL: { 0 ... 9 } + . { 0 ... 9 } +

note the decimal point **must be** surrounded on both sides!

ASTRING: { ASCII character set } *

Operators in this language are basic arithmetic and comparison and logic:

+ - * / % := = <> > < >= <= AND OR NOT

Punctuation in this language now has a few more elements:

: ; , : ( ) [ ] ∧ &

Keywords in this language have been expanded and now include the following:

ARRAY, ASTRING, BEGIN, BREAK, CALL, CASE, CONSTANTS, DECIMAL, DEFAULT, DISPOSE, DO, DOWNTO, ELSE, END, FOR, GOTO, IDENTIFIER, IF, INT ,LABELS,NEW, NEXT, NOT, NULL, NUMBER, OF, OR, PROCEDURE, PROCEDURES, PROGRAM, READ, READLN, REAL, RECORD, REPEAT, RETURN, STRING, THEN, TYPES, UNTIL, UPTO, VA,L VAR, VARIABLE,S WHILE, WRITE, WRITELN

Data Types in this language are limited to three options:

INT, REAL, and STRING

The data type string is only for the convenience of incorporating strings as literal constants or named constants within the source code for use as input prompts or for use as output formatting.

kizescanner.icn

```
# ---------------------------------------------------------- #

# kizescanner.icn
#
# this program is a basic lexical analyzer for
# the programming language kize.
# it recognizes the basic building blocks:
#     - keywords
#     - identifiers
#     - literal           constants
#     - arithmetic        operators
#     - comparison        operators
#     - address           operators
#     - punctuation
#     - other             symbols
#
# input file may be any ASCII text file in free format
# output file will be an ASCII text file of triplets
#     - line number within the original input file
#     - token type  for a given lexeme
#     - token value for a given lexeme
#
# to generate the executable code:
#     $ icont kizescanner
#
# to execute the code, use IO redirection:
#     $ ./kizescanner < input_file > output_file

# ---------------------------------------------------------- #

global keywords       # set of keywords
global symbols        # cset of single character symbols
global line           # storage for a single line of input
global line_count     # integer current pos within input

procedure main ()

  keywords := set ([
    "AND", "ARRAY", "ASTRING", "BEGIN", "BREAK", "CALL",
    "CASE", "CHAR", "CONSTANTS", "DECIMAL", "DEFAULT",
    "DISPOSE", "DO", "DOWNTO", "ELSE", "END", "FOR",
    "GOTO", "IDENTIFIER", "IF", "INT", "LABELS", "NEW",
    "NEXT", "NOT", "NULL", "NUMBER", "OF", "OR",
    "POINTER", "PROCEDURE", "PROCEDURES", "PROGRAM",
    "READ", "READLN", "REAL", "RECORD", "REPEAT",
    "RETURN", "STRING", "THEN", "TYPES", "UNTIL", "UPTO",
    "VAL", "VAR", "VARIABLES", "WHILE", "WRITE", "WRITELN"
  ])

  symbols := '!+-*/%=<>.,:;()[]^&'
```

```
  line_count := 0
  while (line := read ()) do
  {
    line_count +:= 1
    process_tokens ()
  }
  exit ()
end

# ------------------------------------------------------------ #

procedure process_tokens ()
  line ? repeat
  {
    tab (many (' '))
    if (pos(0)) then break
    if (any (&ucase++&lcase)) then
    {
      token_value := tab (many (&ucase++&lcase++&digits))
      token_value := map (token_value,&lcase,&ucase)
      if (member (keywords,token_value))
        then token_type := token_value
        else token_type := "IDENTIFIER"
    }
    else if (any (&digits)) then
    {
      token_value := tab (many (&digits))
      token_type := "NUMBER";
      if (any ('.')) then
      {
        token_value := token_value || "."
        move (1)
        token_type := "DECIMAL"
        if (any (&digits))
          then token_value ||:= tab (many (&digits))
          else stop ("invalid decimal encountered")
      }
    }
    else if (any ('\"')) then
    {
      move (1)
      token_value := tab (upto ('\"'))
      move (1)
      token_value := "\"" || token_value || "\""
      token_type  := "ASTRING"
    }
    else if (any ('\'')) then
    {
      move (1)
      token_value := tab (upto ('\''))
      if (*token_value = 1) then
      {
```

```
         move (1)
         token_value := "\'" || token_value || "\'"
         token_type  := "ACHAR"
      }
      else
         stop ("invalid ASCII character encountered")
   }
#  as in the icon programming language
#  the pound symbol (#) serves as a comment indicator
   else if (any ('#')) then
      return
   else if (=":=") then
     token_type := token_value := ":="
   else if (="<>") then
     token_type := token_value := "<>"
   else if (="<=") then
     token_type := token_value := "<="
   else if (=">=") then
     token_type := token_value := ">="
   else if (="->") then
   {
     token_type := token_value := "."
     write (left (line_count,7),
            left (token_type,12),
            left (token_value,60))
     token_type := token_value := "^"
   }
   else if (any (symbols)) then
     token_type := token_value := move (1)
   else
   {
     symbol := move(1)
     stop (left (line_count,7),
           "unexpected symbol encountered: ",
            symbol)
   }
   write (left (line_count,7),
          left (token_type,12),
          token_value)
  }
  return
end

# ---------------------------------------------------------- #
```

12.2 KIZE: Context Free Grammar

The following pages contain the complete context free grammar for the **kize** programming language. This programming language is not as comprehensive as most modern programming languages, but it does have sufficient options to provide an educational challenge to someone just beginning his or her study of compilers.

Among the options found within the language:

- global and local declarations
- variables and named constants
- user-declared data types: ARRAYs and RECORDs
- type checking: name equivalence and shallow copy
- several atomic types: INT, REAL, STRING, and pointers
- use of labels
- procedures: FUNCTIONs and SUBROUTINEs
- activation: CALL and RETURN
- data transfer: CALL BY VAL and CALL BY VAR
- return value
- scope: linear rather than nested
- expressions: arithmetic, comparison, and logical
- promotion and demotion of data types
- control structures: DO, IF, CASE,
  WHILE, REPEAT, FOR, NEXT, BREAK
- identifier qualification: ARRAY, RECORD, PROCEDURE, pointer
- dynamic memory allocation: NEW and DISPOSE

kize.grm

```
kize_program            PROGRAM IDENTIFIER
                        global_declarations
                        BEGIN procedure_list END

global_declarations     const_declarations
                        type_declarations
                        var_declarations
                        proc_declarations

local_declarations      label_declarations
                        var_declarations

const_declarations      CONSTANTS const_list
const_declarations

const_list              IDENTIFIER : literal ;
                        more_const_list

more_const_list         const_list
more_const_list

literal                 NUMBER
literal                 DECIMAL
literal                 ASTRING
literal                 NULL

type_declarations       TYPES type_list
type_declarations

type_list               IDENTIFIER : declaration_type ;
                        more_type_list

more_type_list          type_list
more_type_list

declaration_type        array_type
declaration_type        record_type

array_type              ARRAY [ ub ] OF declared_type

ub                      NUMBER
ub                      IDENTIFIER

record_type             RECORD fld_list END

fld_list                IDENTIFIER : declared_type ;
                        more_fld_list

more_fld_list           fld_list
more_fld_list
```

```
declared_type           atomic_type
declared_type           IDENTIFIER
declared_type           ^ declared_type

atomic_type             INT
atomic_type             REAL
atomic_type             STRING

var_declarations        VARIABLES var_list
var_declarations

var_list                IDENTIFIER : declared_type ;
                        more_var_list

more_var_list           var_list
more_var_list

label_declarations      LABELS label_list
label_declarations

label_list              label ; more_label_list

more_label_list         label_list
more_label_list

label                   NUMBER

proc_declarations       PROCEDURES proc_list
proc_declarations

proc_list               proc_signature more_proc_list

proc_signature          IDENTIFIER ( formal_arguments )
                        return_type ;

more_proc_list          proc_list
more_proc_list

formal_arguments        formal_argument
                        more_formal_arguments
formal_arguments

more_formal_arguments   , formal_arguments
more_formal_arguments

formal_argument         call_by IDENTIFIER : defined_type

call_by                 VAL
call_by                 VAR
call_by

return_type             : atomic_type
return_type
```

```
actual_arguments          actual_argument
                          more_actual_arguments
actual_arguments

more_actual_arguments     , actual_arguments
more_actual_arguments

actual_argument           expression

procedure_list            procedure more_procedure_list

more_procedure_list       procedure_list
more_procedure_list

procedure                 PROCEDURE IDENTIFIER
                          local_declarations
                          BEGIN statement_list END

statement_list            statement statement_list
statement_list

statement                 opt_label executable_statement

opt_label                 label
opt_label

executable_statement      read_statement
executable_statement      write_statement
executable_statement      assignment_statement
executable_statement      goto_statement
executable_statement      empty_statement

executable_statement      compound_statement
executable_statement      if_statement
executable_statement      while_statement
executable_statement      repeat_statement
executable_statement      case_statement
executable_statement      for_statement
executable_statement      next_statement
executable_statement      break_statement

executable_statement      call_statement
executable_statement      return_statement

executable_statement      dispose_request

read_statement            READ   ( actual_arguments ) ;
read_statement            READLN ( actual_arguments ) ;

write_statement           WRITE   ( actual_arguments ) ;
write_statement           WRITELN ( actual_arguments ) ;
```

```
assignment_statement     named_item := two_options ;

two_options              expression
two_options              new_request

expression               simple_expression comparison

comparison               compop simple_expression
comparison

simple_expression        sign term more_terms

more_terms               addop term more_terms
more_terms

term                     factor more_factors

more_factors             mulop factor more_factors
more_factors

factor                   named_item
factor                   ( expression )
factor                   NOT factor
factor                   literal
factor                   & named_item

named_item               IDENTIFIER qualifier

addop                    +
addop                    -
addop                    OR

mulop                    *
mulop                    /
mulop                    %
mulop                    NOT

sign                     +
sign                     -
sign

compop                   =
compop                   <>
compop                   >
compop                   >=
compop                   <
compop                   <=

goto_statement           GOTO label ;

empty_statement          ;

compound_statement       DO statement_list END ;
```

```
if_statement            IF ( expression )
                        then_clause else_clause

then_clause             THEN statement

else_clause             ELSE statement
else_clause

case_statement          CASE ( expression ) OF case_list

case_list               NUMBER : statement case_list
case_list               DEFAULT : statement

while_statement         WHILE ( expression ) statement

repeat_statement        REPEAT statement_list
                        UNTIL ( expression ) ;

for_statement           FOR IDENTIFIER := initial
                        direction final statement

initial                 expression

final                   expression

direction               UPTO
direction               DOWNTO

next_statement          NEXT ;

break_statement         BREAK ;

call_statement          CALL named_item ;

return_statement        RETURN opt_value ;

opt_value               expression
opt_value

qualifier               procedure_qualifier
qualifier               structure_qualifier

procedure_qualifier     ( actual_arguments )

structure_qualifier     array_qualifier
structure_qualifier     record_qualifier
structure_qualifier     pointer_qualifier
structure_qualifier

array_qualifier         [ expression ] structure_qualifier

record_qualifier        . IDENTIFIER structure_qualifier
```

pointer_qualifier	^ structure_qualifier
new_request	NEW defined_type
dispose_request	DISPOSE named_item

12.3 KIZE: Semantic Rules

In this section we have the unenviable task of trying to explain what all this stuff is supposed to do! Grammars define **syntax** or structure for a valid program; **semantics** explains what it all means. In this section we will summarize our basic semantic rules in English as best we can.

data types

- INT is a 64-bit integer (signed twos-complement)
- REAL is an IEEE 64-bit floating point representations
- STRING is C string representation
  (8-bit ASCII characters terminated by 0x0h)
- BOOL does not exist!
  rather we use INT: 0 = FALSE and non-zero = TRUE

arithmetic operators

- addition: +
- subtraction: -
- multiplication: *
- division or quotient: /
- remainder or modulus: % (INT only)
- INT mode arithmetic yields INT results
- REAL mode arithmetic yields REAL results
- mixed mode arithmetic **promotes** INT values to REAL values

logical operators

- logical AND
- logical OR
- logical NOT
- both AND and OR to be implemented using ***short-circuit evaluation***

comparison operators

- equality: =
- inequality: <>
- greater than: >
- less than: <
- greater than or equal: >=
- less than or equal: <=
- mixed mode comparisons **promote** INT values to REAL values

assignment operator

- assign REAL to INT **demotes** REAL value to INT value
- assign INT to REAL **promotes** INT value to REALvalue
- no assignments involving STRING!
- named types require *name equivalence* (to be discussed later)
- named types utilize *shallow copy* (to be discussed later)

actual arguments matched to formal arguments

- must agree in **number** and **type**
  no promotion / demotion
- **call by value**: any expression is acceptable for actual argument
- **call by variable**: only an L-value (address) is acceptable for actual argument

return type

- must be INT or REAL
  no named types
- must agree in **type**
  no promotion / demotion
- we will use **I0** register to return INT value
- we will use **F0** register to return REAL value

named types

- multi-dimension arrays are **not directly** definable
- nested records are **not directly** definable
- **but** a previously defined named types **may be used** to nest structured types
- named types **must** use call by variable
- named types require *name equivalence* (to be discussed later)
- named types utilize *shallow copy* (to be discussed later)

labels

- labels highlight important locations within the source code
- goto statements are the most fundamental of all branching commands
  although most modern textbooks consider them taboo!
- **all** labels in **kize** are local!
- **jumping out of** a procedures is **not** allowed
- **jumping into** another procedure is **not** allowed

12.4 Phase Zero Implementation

Create a clean working directory for the development of your compiler.

The first step is to copy over into this directory useful files from CALC Arithmetic Calculator:

- calcscanner.icn
- calcparser.icn
- lexeme.icn
- error.icn

All of these files will have to be enhanced and expanded in the following pages. The other files are not really necessary and need not be copied into your working directory.

Item One: Please update the **calcscanner** to incorporate the new elements introduced at the beginning of this chapter (building blocks). You now have your **kizescanner**!

calcscanner.icn $\longrightarrow$ kizescanner.icn.

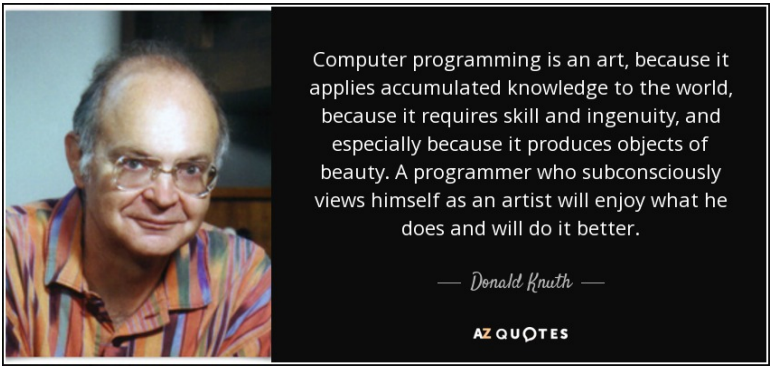
Item Two: This is an extremely ***very time consuming*** step – write a recursive descent parser for the **kize** context free grammar above! Once again, you may build upon the coding found in your **calcparser**.

calcparser.icn $\longrightarrow$ kizeparser.icn.

From you experience with the CALC programming language, you know this is a very straightforward process to convert from the grammar to the actual parser. The problem with this step is the size of the context free grammar. I guarantee that you will be an excellent touch-typist by the time you have completed this parser!

Items Three and Four: Consider any enhancements you might want to include for **lexeme** or **error**. Although both files are essentially fine as is!

These four files provide the starting point for implementing the **kize** compiler. All **Icon** source files for the **kize compiler** may be found in **Chapter Twenty-One: The End Result**.


Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.
— Donald Knuth —
AZ QUOTES

Chapter 13

Phase One: Symbol Tables

Compiling is Just Bookkeeping!

The sad truth about compilers is that much of the work is just basic bookkeeping.

Correct information must be maintained about everything: ***global information*** which is accessible across the entire program and ***local information*** which is accessible only within a specific area, called a **procedure** (or more generally a **subprogram**).

A **kize** source code program is organized in a linear fashion, with global information being defined first and followed by a non-empty list of procedure definitions each having its own collection of local information.

The first procedure definition must be that of the **main procedure**. All programs will contain at least this component to initiate program execution.

It is important to remember that a specific identifier may be defined and redefined several times across the entire source code program. However, an identifier must be uniquely defined in any given area, either globally or within a given procedure. A locally defined identifier has ***precedence*** over any other definition that may appear elsewhere.

I am including at the very onset of this chapter those elements of our context free grammar that will be our focus for the coming pages.

Please note: Our initial focus is on very basic declarations (named constants and variables). But we must be aware that structured data types and pointers exist; we must be aware that procedures, argument lists, and return types also exist. Although these topics need not be implemented at this time, we should definitely keep in mind that they too will eventually be incorporated into the coding we write at this time.

kize.grm (phase one)

```
kize_program              PROGRAM IDENTIFIER
                          global_declarations
                          BEGIN procedure_list END

global_declarations       const_declarations
                          type_declarations
                          var_declarations
                          proc_declarations

local_declarations        label_declarations
                          var_declarations

const_declarations        CONSTANTS const_list
const_declarations

const_list                IDENTIFIER : literal ;
                          more_const_list

more_const_list           const_list
more_const_list

literal                   NUMBER
literal                   DECIMAL
literal                   ASTRING
literal                   NULL

type_declarations         TYPES type_list
type_declarations

type_list                 IDENTIFIER : declaration_type ;
                          more_type_list

more_type_list            type_list
more_type_list

// initially only atomic types: INT, REAL, and STRING
// later we discuss structured types: ARRAYs and RECORDs

        declaration_type array_type
        declaration_type record_type

        array_type        ARRAY [ ub ] OF declared_type

        ub                NUMBER
        ub                IDENTIFIER

        record_type       RECORD fld_list END

        fld_list          IDENTIFIER : declared_type ;
                          more_fld_list
```

```
        more_fld_list    fld_list
        more_fld_list

declared_type            atomic_type
        declared_type    IDENTIFIER
        declared_type    ^ declared_type

atomic_type              INT
atomic_type              REAL
atomic_type              STRING

var_declarations         VARIABLES var_list
var_declarations

var_list                 IDENTIFIER : declared_type ;
                         more_var_list

more_var_list            var_list
more_var_list

label_declarations       LABELS label_list
label_declarations

label_list               label ; more_label_list

more_label_list          label_list
more_label_list

label                    NUMBER

proc_declarations        PROCEDURES proc_list
proc_declarations

proc_list                proc_signature more_proc_list

proc_signature           IDENTIFIER ( formal_arguments )
                         return_type ;

more_proc_list           proc_list
more_proc_list

formal_arguments         formal_argument
                         more_formal_arguments
formal_arguments

more_formal_arguments    , formal_arguments
more_formal_arguments

formal_argument          call_by IDENTIFIER : defined_type

call_by                  VAL
call_by                  VAR
call_by
```

```
return_type              : atomic_type
return_type

// we are only concerned with declaring a procedure
// later we discuss its definition and its activation

        actual_arguments actual_argument
                         more_actual_arguments
        actual_arguments

        more_actual_arguments , actual_arguments
        more_actual_arguments

        actual_argument expression

        procedure_list  procedure more_procedure_list

        more_procedure_list procedure_list
        more_procedure_list

procedure                PROCEDURE IDENTIFIER
                         local_declarations
                         BEGIN statement_list END
```

13.1 The Importance of Tables

The **kize** programming language requires a fairly extensive database of information to keep track of all the various components that comprise the language.

- global named constants
- global named types
- global variables
- global procedures

- local labels
- local variables

If one so desired, ***everything*** could be stored within a ***single*** table! Another option would certainly be to store each individual component in a ***separate*** table. I would like to postpone sharing my decisions regarding the database implementation until after summarizing the information that needs to be saved in each category.

named constants

Named constants are very similar to global variables in that an identifier has been assigned an associated data value. However, the significance of a named constant lies in that fact that it is immutable, i.e., unchangeable. Named constants allow the programmer to define key parameters within the source code to help with readability and/or flexibility. Common examples would be:

- MAXSIZE = 100
- MONTHS = 12
- PI = 3.1416
- RADTWO = 1.414
- NAME = "paul kaiser"

A named constant entry requires saving three pieces of information:

- its identifier
- its data type
- its value

Named constants must be declared globally; there can be no local named constants. If the identifier is redefined locally, it will be as a local variable and the named constant value will not be accessible.

variables

Variables enable the programmer to associate a date value with an identifiers. However, unlike a named constant, a variable's data value can be be changed.

Variables can be globally defined and available throughout the entirety of the source code program; variables can also be locally defined and available only within the procedure where it is declared.

As we will discuss in the next section, global variables are statically allocated storage in main memory; local variables are allocated storage in the memory stack.

A variable entry requires saving four pieces of information:

- its identifier
- its data type
  initially only INT, REAL, or STRING
  but eventually structured types and pointers as well
- its location in memory (either as a static address or as an offset address)
- whether its location in memory is a direct address or indirect address

Global variables will have a mnemonic static address in main memory (low numbered addresses).

Local variables will not have a static address. Its location will be on the system stack, relative to key address called the frame pointer. Note that the offset value is an integer constant for a specific local variable, but its ultimate address in memory need not always be the same. As the frame pointer changes, so does the offset address.

named types

Data types within a programming language typically come in two flavors: those that are built-in or ***atomic***, those that may be built-up or ***structured***.

The **kize** programming language has three atomic types: INT (64-bit integer), REAL (64-bit floating point), and STRING (64-bit pointer to a C-string). The **kize** programming language has two structured elements: the very traditional constructs of ARRAYs and RECORDs.

A third flavor of data type will be discussed in the final chapter of PART THREE – pointers. Pointers consider memory addresses not only as locations for accessing data but as an actual data value in its own right. Dynamic allocation and de-allocation and memory management become important topics in the study of compiler construction.

At this point in our study the implementation of a type table is not absolutely essential. However, even if the database is limited to just the three atomic types, it is good practice to be aware that type information will be crucial at a later point in compiler construction and to anticipate what future enhancements might entail.

A data type entry requires three pieces of information:

- its identifier
- its type size (in 64-bit klunks)
- its type information
  whether the type is an array or a record
  together with its associated detail

Remember that STRING is an ***internal*** data type to **kize**. Named constant declarations may include STRING literals for prompts and output elements; but STRING may not be used in any variable declarations.

In a similar vein, there is no data type BOOL for truth values! Comparison operators and logical operators yield INT results (FALSE corresponds to zero and TRUE corresponds to any non-zero value). The tokens **BOOL, TRUE, FALSE** should never appear in any **kize** source code file.

procedures

The last category of global information is that regarding **procedures** within the source code program. Procedure is a synonym for subprogram, which typically come in two distinct flavors:

- subprograms that return a single value
- subprograms that return nothing or more than one value

The former were traditionally referred to as a **function**; the latter as a **subroutine**. Languages today may retain the distinction (Pascal), or may call everything a function (C++), or may call everything a procedure (PL/I). **kize** calls everything a **procedure**.

A procedure entry will ultimately require three pieces of information:

- its identifier
- its optional formal argument list
  argument identifier
  argument data type
  data transfer mechanism: either call by val or call by var
- its optional return data type
  which in **kize** must be atomic

Procedure declarations are contained within the global declarations. This ensures that they are fully declared prior to the subsequent definition of the procedure within the procedure list, which also contains its local declarations and its executable code.

At this point in our study a type table for procedures is not absolutely essential. However, even if the database is limited to just the mandatory **main procedure**, it is good practice to be aware that procedure information will be crucial at a later point in compiler construction.

The **main procedure** has identifier "MAIN", an empty formal argument list, and an INT return type (typically set to the value 0 representing no errors).

labels

In addition to local variables, a procedure may also have local labels – defined positions within the source code where control may be transferred to. A label in the **kize** programming language is a simple numeric value; a label in assembly language is usually a mnemonic identifier. As with variables, a local label may duplicate a label that appears elsewhere, but a local label must be unique within a given procedure. And as with global variables, each label name must have a unique internal label name.

A label entry requires two pieces of information:

- its external label – a number found in **kize** source code
- its internal label – a unique mnemonic identifier used within the **kcode** assembler file

One concluding comment regarding local labels and local variables – local information must be kept separate from global information. Similarly local information for one procedure must also be kept separate from local information for another procedure.

However, it is important to note that since procedures are listed in linear (sequential) order. There is no overlap between or among procedures. As a result, when one moves from a procedure into the next one, one can simply discard the local information in the database and replace it with the new local information for the current procedure.

literal constants

One last topic to consider is what to do with literal constants.

At first glance, one might assume that any literal constant that appears within the **kize** source code would simply be represented as an immediate value within the assembly language code. Such immediate values have always caused me difficulty, especially when applied to floating point representations for real data values. Immediate values are imbedded within the instruction itself; hence, they do not have the same bit storage and characteristics as a full 64-bit representation. Because immediate values have fewer bits available, they have a more restrictive range and accuracy limitations.

My original solution may possibly be considered overkill, but I chose to store every individual literal constant found within the executable code as a unique global variable, essentially creating its own named constant. Hence the delay in generating the **data segment** until the very end of the compilation process. Literal constants appear within the executable code in the source program and are not recognized earlier in the declarations.

Later still, when my frustrations with existing assembly languages regarding stack alignment and lack of clarity regarding the limits on immediate items, I chose to simulate an assembly language (**kcode**) where immediate values preserve full 64-bit representation.

As a result of all this trial and error, I would finally summarize my thoughts as follows:

- ***named constants*** are entered into the global symbol table an an immutable variable entry. The static address for the named constant and its literal value are included in the setup for static memory using a _DEFINE directive.
- ***variables*** are entered into the global symbol table or local symbol table (as appropriate) but it is not an immutable variable entry. If it is a global variable declaration, an entry is included in the setup for static memory using a _RESERVE directive.
- ***literal constants*** need not be stored in either symbol table, but will be incorporated into **kcode** assembly language as an immediate value.

As a result, with **kcode** there is no reason to delay generating the **data segment** until after the **code segment**. However, remember that X86_64 and AARCH64 are not as flexible and be forewarned that handling literal constants in the future may be a bit more complicated!

so, back to how many tables ...

When I originally wrote my parsers and compilers in C++, I used a lot of tables. Each table held record entries with very specific components for different categories. There is no inherent problem with this strategy, but it did require a lot of customized coding (in C++) unique to each table.

After I rediscovered my graduate school experience using the programming language **Icon**, I decided that the flexibility of a typeless language and the immediate availability of useful storage structures made compiler writing so much simpler and more pleasant. Quite possibly a modern programming language, like **Python** might provide a similar experience.

The **Icon** programming code which follows in Chapter 21 utilizes only two tables – one global and one local. Even though the actual entries that are stored within the table may be very dissimilar, they are discernible from one another and easily maintainable.

13.2 Assembly Language Redux

Recall our previous discussion of assembly language. Assembly language programs are divided into two parts: a *data segment* and a *code segment*. The data segment is used to define (static) storage requirements; the code segment is used to define the executable code.

Remember that **global / static storage** grows from the bottom up (in the direction of increasing memory addresses) and that access to information typically utilizes *absolute addressing*; on the other hand, **stack / automatic** storage grows from the top down (in the direction of decreasing memory addresses) and that access to information utilizes *offset addressing* relative to the *frame pointer*.

One implication of allowing the repetition of identifiers within the **kize** programming language is that this duplication **can not** carry over into the assembly language code. This limitation forces us to guarantee that all identifiers in our assembly language source code are unique. To assist in this endeavor, I strongly suggest implementing a simple internal name generator which cranks out unique identifiers at will! The name generator I use creates the sequence of strings: K0, K1, K2, ...

Remember also that **kize** is organized in a linear fashion with global information appearing at the very beginning of the source code file. This is very helpful because this information will obviously be used throughout the entire assembly language file. It is very tempting to immediately generate the global / static memory allocation upon completion of the global declarations. However, as I have mentioned several times, it may be beneficial to delay this output until the **code segment** has been completed in its entirety.

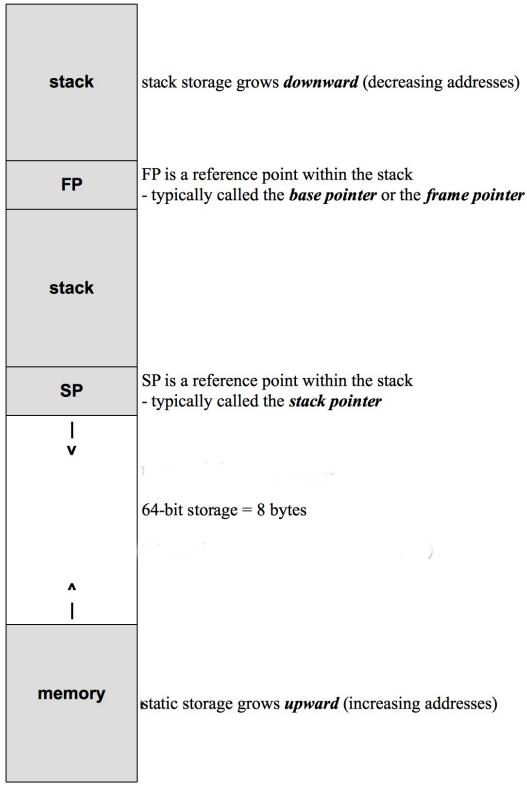
Since **kcode** allows for the **data segment** and the **code segment** to appear in either order, it really does not matter. But it is a decision to be made. Even though my choice for assembly language is **kcode**, I have chosen to postpone code generation for the data segment until the very end.

However, delaying the **data segment** does not mean that the information gleaned can not be displayed earlier in the compilation process. I like to display the global declaration information and the local declaration information upon completion of those sections of

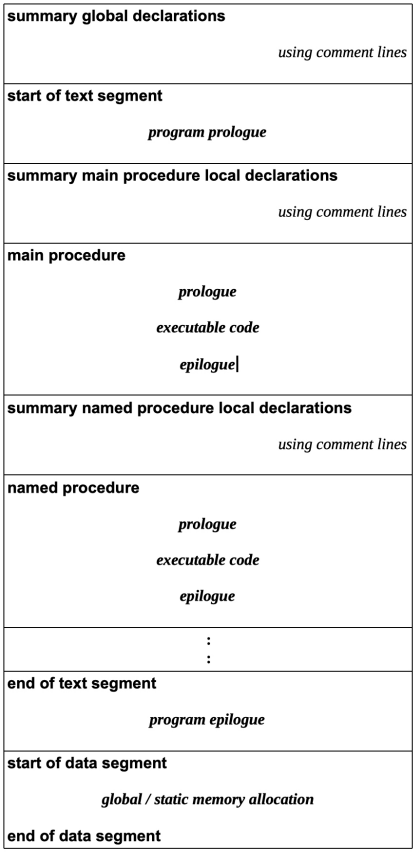
the **kize** source code. I do so by using comment lines. I find this very useful in the early stages of compiler development when so little executable code is visible. I use a global variable called DEBUG in my compiler program to toggle the display of these comment lines ON and OFF. This helps ensure that information in the symbol tables is correct from the very beginning of compiler construction; and this provides a solid foundation for subsequent coding.

Memory Organization

Recall the common utilization of memory implemented in many assembly language implementations.



Our **kcode** assembly language program should adhere to the following organizational structure:



Our previous work on implementing a compiler for the **calc** programming language will serve us well as we move forward.

Dedicated Registers

As we begin writing a serious compiler, it is necessary for us to organize our thoughts. **kcode** provides us with a large number of general purpose 64-bit registers: **I0** through **I15** for integer values and **F0** through **F15** for floating point real values.

For integer calculations and comparisons I suggest reserving four integer registers

- IA = **I0** (*result*)
- IB = **I1** (*operand1*)
- IC = **I2** (*operand2*)
- ID = **I3** (*spare*)

For floating point calculations and comparisons I suggest reserving four floating point registers

- FA = **F0** (*result*)
- FB = **F1** (*operand1*)
- FC = **F2** (*operand2*)
- FD = **F3** (*spare*)

For memory addressing I suggest reserving four integer registers

- SAR = **I4** (*source address register*)
- DAR = **I5** (*destination address register*)
- IR = **I6** (*index register*)
- OR = **I7** (*offset register*)

Lastly, for specialized address registers I suggest reserving four integer registers

- SP = **I12** (*stack pointer*)
- FP = **I13** (*frame pointer*)
- RP = **I14** (*return pointer*)
- HP = **I15** (*heap pointer*)

13.3 Phase One Implementation

13.3.1 Organization and Structure

The previous two sections highlight the focus for this chapter: creating an accurate database to support our compiler and utilizing this information appropriately.

Obviously the file ***tables.icn*** requires significant rewriting and expansion! We are going from one table in our database to two all-purpose tables! Furthermore, if we are to display the information found within them, we need several variations on the previous display algorithm, each one customized for a specific table entry. These displays algorithms will generate comment lines within the **kcode** assembly language file.

The static memory components should include both _RESERVE and _DEFINE directives. Also, please note, static memory definition components should not be commented! These lines are definitely part of the **data segment** and not informational.

And finally, the display of our database elements should typically be done immediately after processing the global declarations and again after processing each set of local declarations, i.e., as part of the program prologue and each procedure prologue.

So, we need to update and augment the ***tables.icn*** file to incorporate the following new features:

- global variables
  including named constant entries
  - named constants are immutable
  - variables are not
- global types
  at least include atomic types INT, REAL, and STRING
  - structured types (ARRAYs and RECORDs) can be added later
  - pointers can be added later -
- global procedures
  at least include the main procedure
  - empty argument list
  - INT return value (typically 0 => no errors)

- local variables
  which were not a concern with CALC
- local labels
  which were also not a concern with CALC

And we need to update and augment the ***assembler.icn*** file to incorporate the following new features:

- if not done so previously, initialize the assembly language code component:
  to identify the dedicated registers
  to create a name generator for all internal identifiers
- the program prologue should now display global database information
  as comment lines in the assembly language code
- the program epilogue should now generate the _DATA_SEGMENT, including both _DEFINE and _RESERVE directives as appropriate
- the main procedure prologue should display local database information
  as comment lines in the assembly language code
- the main procedure prologue and epilogue should
  setup and cleanup the stack activation as previously done

Finally, the source code for the **kize** compiler is going to be significantly larger in size than the **calc** compiler. The context free grammar is larger; the parser was therefore larger; and the compiler will therefore obviously also be larger.

I highly recommend subdividing the compiler organization into three componeents:

- **kizecompiler.icn**: a relatively small driver program which is limited to two similar production rules
- **declarations.icn**: focused solely on production rules for global and local declarations
- **executables.icn**: focused solely on production rules for compiling **kize** executable code

kizecompiler.icn will handle two very similarly structured production rules: **kize_program** and **procedure**. Both productions have parallel syntax but generate information to be stored in different tables in the database.

declarations.icn will be a very large file that will build and maintain the compiler database.

executables.icn will also be a very large file that will retrieve information stored in the compiler database and use it to generate **kcode** instructions.

13.3.2 End-of-Phrase Markers

At this point, review the context free grammar for the **kize** programming language and insert end-of-phrase markers to highlight where information needs to be saved, where information needs to be checked for consistency, and where information needs to be inserted into the database.

Remember we are focusing right now on ***building*** the database and not on any executable code. Consider only the context free grammar elements that pertain to global declarations and local declarations and production rules that build from those two.

Immediately after completing the global declarations for the source code files, you should emit the *program prologue* to display the declarations as comment statements; and similarly, immediately after completing the local declarations for the main procedure, you should emit the *procedure prologue* to display the declarations as comment statements.

kize.grm (phase one)

```
kize_program            PROGRAM IDENTIFIER !save_ident
                        global_declarations
                        BEGIN !emit_program_prologue
                        procedure_list
                        END !emit_program epilogue

procedure               PROCEDURE IDENTIFIER !save_ident
                        !check_declared
                        local_declarations
                        BEGIN !emit_procedure_prologue
                        statement_list
                        END !emit_procedure_epilogue

global_declarations     !initialize_global_symbol_table
                        const_declarations type_declarations
                        var_declarations proc_declarations

local_declarations      !initialize_local_symbol_table
                        label_declarations var_declarations
                        !save_local_storage

const_declarations      CONSTANTS const_list
const_declarations

const_list              IDENTIFIER !check_duplicate :
                        literal ; !insert_const_entry
```

```
                        more_const_list

more_const_list         const_list
more_const_list

literal                 NUMBER
literal                 DECIMAL
literal                 ASTRING

type_declarations       TYPES type_list
type_declarations

type_list               IDENTIFIER !check_duplicate :
                        declaration_type ;
                        !insert_type_entry
                        more_type_list

more_type_list          type_list
more_type_list

declared_type           atomic_type

var_declarations        VARIABLES var_list
var_declarations

var_list                IDENTIFIER !check_duplicate :
                        declared_type ;
                        !insert_variable_entry
                        !insert_static_memory_entry
                        more_var_list

more_var_list           var_list
more_var_list

label_declarations      LABELS label_list
label_declarations

label_list              label !check_duplicate ;
                        !insert_label_entry
                        more_label_list

more_label_list         label_list
more_label_list

label                   NUMBER

proc_declarations       PROCEDURES proc_list
proc_declarations

proc_list               proc_signature more_proc_list

proc_signature          IDENTIFIER !check_duplicate
                        ( formal_arguments )
```

```
                        return_type ; !insert_proc_entry

more_proc_list          proc_list
more_proc_list

formal_arguments        formal_argument
                        more_formal_arguments
formal_arguments

more_formal_arguments   , formal_arguments
more_formal_arguments

formal_argument         call_by IDENTIFIER : defined_type
                        !save_formal_argument

call_by                 VAL
call_by                 VAR
call_by

return_type             : atomic_type !check_type
return_type
```

13.3.3 Testing

Be sure to test your compiler on sample files to check for accuracy of information. I will share two such files (the first a minimal implementation and the second a more complete implementation) and the corresponding output on the pages which immediately follow. Feel free to use them to test your implementation of a **kize** compiler.

After incorporating any new features into ***tables.icn*** and ***assembler.icn***, always be sure to recompile.

$ **icont** kizecompiler [declarations executables] lexeme error tables assembler

$ **./kizescanner** < *source_file*.**kz** | **./kizecompiler** > *assembler_file*.**k**

The last step in the compilation process will be to actually test whether the assembly language executes as intended!

$ **kbox** *assembler_file*.***k***

Certainly executing the assembly language file using **kbox** would be a waste of time! However, reviewing the comment lines describing global and local declarations would be a good first step in ensuring a solid foundation for the balance of the compiler implementation. Also, the _DATA_SEGMENT and the _CODE_SEGMENT should have been properly generated as a containing shell for the balance of the **kcode** executable code.

13.4 Sample Symbol Table Generation

The following pages contain two sample **kize** programs and the resulting output generated by this first phase of our compiler.

13.4.1 phase_01a

phase_01a.kz

```
program phase01a

    constants
      pi          : 3.1416;
      e           : 2.182818;
      uno         : 1;
      duo         : 2;
      message     : "bye!";

    variables
      x           : int;
      y           : real;

begin

  procedure main

      labels
        1;
        2;
        3;

      variables
        a         : int;
        b         : real;
        x         : real;
        y         : int;

  begin

  end

end
```

phase_01a.k

```
# BEGIN PROGRAM: PHASE01A

# GLOBAL SYMBOL TABLE:

# CONSTANTS
#       MESSAGE         STRING          "bye!"
#       E               REAL            2.182818
#       UNO             INT             1
#       PI              REAL            3.1416
#       DUO             INT             2

# TYPES
#       INT             1               ATOM
#       REAL            1               ATOM
#       STRING          1               ATOM

# VARIABLES
#       X               INT             K1
#       Y               REAL            K2

# PROCEDURES
#       MAIN            0               INT


        _CODE_SEGMENT

        _GLOBAL         MAIN


# BEGIN PROCEDURE: MAIN

# LOCAL SYMBOL TABLE:

# LABELS
#       3               K5
#       1               K3
#       2               K4

# VARIABLES
#       A               INT             -1
#       X               REAL            -3
#       B               REAL            -2
#       Y               INT             -4


        _LABEL          MAIN
        push            I14
        push            I13
        mov             I13 I12
        movi            I6 -4
        add             I12 I12 I6
```

```
                _LABEL          EXITMAIN
                mov             I12  I13
                pop             I13
                pop             I14
                ret

# END PROCEDURE: MAIN


                _DATA_SEGMENT

                _RESERVE        K1  1
                _RESERVE        K2  1

# END PROGRAM: PHASE01A
```

13.4.2 phase_01b

phase_01b.kz

```
program phase01b

    constants
      pi          : 3.1416;
      e           : 2.182818;
      uno         : 1;
      duo         : 2;
      message     : "bye!";

    types
      arraytype   : array [7] of int;
      recordtype  : record
                    f : real;
                    g : int;
                    end;
      datatype    : record
                    p : arraytype;
                    q : recordtype;
                    r : real;
                    end;
      junktype    : array [11] of datatype;

    variables
      x           : int;
      y           : real;
      alpha       : arraytype;
      beta        : recordtype;
      gamma       : datatype;

    procedures
      f   (a : int,var b : real) : real;
      sub (var x : real,var y : real);

begin

  procedure main

      labels
        1;
        2;
        3;

      variables
        a         : int;
        b         : real;
        delta     : junktype;

  begin
  end
```

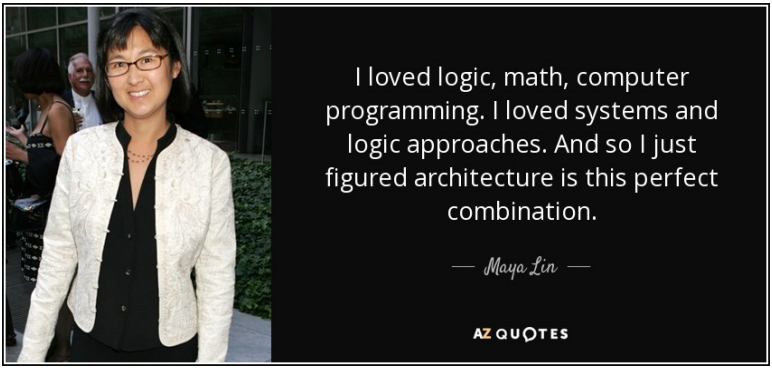
```
  procedure f

      labels
        3;
        4;

      variables
        p          : real;
        q          : int;
        r          : recordtype;

  begin
  end

  procedure sub

      labels
        3;
        4;

      variables
        f          : int;
        g          : int;
        h          : arraytype;

  begin
  end

end
```

phase_01b.k

```
# BEGIN PROGRAM: PHASE01B

# GLOBAL SYMBOL TABLE:

# CONSTANTS
#       MESSAGE         STRING          "bye!"
#       E               REAL            2.182818
#       UNO             INT             1
#       PI              REAL            3.1416
#       DUO             INT             2

# TYPES
#       ARRAYTYPE       7               ARRAY
#                       7       INT
#       RECORDTYPE      2               RECORD
#                       F               REAL            0
#                       G               INT             1
#       DATATYPE        10              RECORD
#                       P               ARRAYTYPE       0
#                       Q               RECORDTYPE      7
#                       R               REAL            9
#       INT             1               ATOM
#       JUNKTYPE        110             ARRAY
#                       11      DATATYPE
#       REAL            1               ATOM
#       STRING          1               ATOM

# VARIABLES
#       BETA            RECORDTYPE      K4
#       ALPHA           ARRAYTYPE       K3
#       X               INT             K1
#       GAMMA           DATATYPE        K5
#       Y               REAL            K2

# PROCEDURES
#       SUB             2               none
#                       X               REAL            var
#                       Y               REAL            var
#       MAIN            0               INT
#       F               2               REAL
#                       A               INT             val
#                       B               REAL            var

        _CODE_SEGMENT

        _GLOBAL         MAIN

# BEGIN PROCEDURE: MAIN

# LOCAL SYMBOL TABLE:
```

```
# LABELS
#       3                   K8
#       1                   K6
#       2                   K7

# VARIABLES
#       DELTA               JUNKTYPE        -112
#       A                   INT             -1
#       B                   REAL            -2

                _LABEL          MAIN
                push            I14
                push            I13
                mov             I13 I12
                movi            I6 -112
                add             I12 I12 I6

                _LABEL          EXITMAIN
                mov             I12 I13
                pop             I13
                pop             I14
                ret

# END PROCEDURE: MAIN

# BEGIN PROCEDURE: F

# LOCAL SYMBOL TABLE:

# LABELS
#       3                   K9
#       4                   K10

# VARIABLES
#       P                   REAL            -1
#       A                   INT             3
#       Q                   INT             -2
#       B                   REAL            2           indirect
#       R                   RECORDTYPE      -4

                _LABEL          F
                push            I14
                push            I13
                mov             I13 I12
                movi            I6 -4
                add             I12 I12 I6

                _LABEL          EXITF
                mov             I12 I13
                pop             I13
                pop             I14
                ret
```

```
# END PROCEDURE: F

# BEGIN PROCEDURE: SUB

# LOCAL SYMBOL TABLE:

# LABELS
#     3                 K11
#     4                 K12

# VARIABLES

#     G                 INT               -2
#     H                 ARRAYTYPE         -9
#     X                 REAL              3           indirect
#     Y                 REAL              2           indirect
#     F                 INT               -1

            _LABEL          SUB
            push            I14
            push            I13
            mov             I13  I12
            movi            I6  -9
            add             I12  I12  I6

            _LABEL          EXITSUB
            mov             I12  I13
            pop             I13
            pop             I14
            ret

# END PROCEDURE: SUB

            _DATA_SEGMENT

            _RESERVE        K1  1
            _RESERVE        K2  1
            _RESERVE        K3  7
            _RESERVE        K4  2
            _RESERVE        K5  10

# END PROGRAM: PHASE01B
```


I loved logic, math, computer programming. I loved systems and logic approaches. And so I just figured architecture is this perfect combination.
— Maya Lin —
AZ QUOTES

Chapter 14

Phase Two: Main Procedure

Everyone Envied Bjorn's Gold Medal in Compiler Construction

As Stephen Covey likes to say

The main thing
is to keep
the main thing
the main thing!

On some transcendental level that probably is relevant to compiler construction. But I just wanted to include the quote somewhere in this text and thought this would be the place to do it!

The title for this chapter – **Main Procedure** – is somewhat misleading. The chapter is really a lot of miscellaneous stuff to get us on our way with the rest of the compiler. And the **main procedure** is where we have to begin generating our executable code. The main procedure will be our first opportunity to actually work with global versus local variables and retrieving L-values versus retrieving R-values,

The topics to be covered in this chapter will include:

- data flow: memory to/from registers to/from the stack
- very simplified READ and WRITE statements
- very simplified ASSIGNMENT statements
- promotion and demotion

The goal of this chapter is to incorporate enough executable code into our main procedure so that we can thoroughly test the accurate flow of data within the various components: input, output, assignment (with promotion / demotion of data values as necessary), and using literal constants. With this arsenal in place we **can not** write very complicated programs, but we **can** be certain that information being processed is both correct and accessible.

The subsequent two chapters – **Phase Three: Simple Statements** and **Phase Four: Control Statements** – can then build upon this solid foundation.

14.1 Data Flow

Recall that every variable has two distinct values associated with it – its data value (called an R-value) and its address value (called an L-value). They are ***the what*** and ***the where*** of the variable. The R-values are required for doing calculations and comparisons; the L-values are required for gaining access to the R-value and making an assignment.

Data flow should be very simple. In principle,

$$\textbf{register} \Longleftrightarrow \textbf{stack}$$

and

$$\textbf{memory} \Longleftrightarrow \textbf{register}$$

Data is moved either ***to the stack*** or ***from the stack*** using the traditional ***push*** or ***pop*** operations. Consider the following code fragments to incorporate into **assembler.icn** to save and to retrieve register contents from the stack.

data flow (registers $\longleftrightarrow$ stack)

```
procedure save_register (reg)
  # emit kbox assembly code to:
  # save the value found in the specified reg
  # to the top of the stack

  emit_line ("push",reg,,,reg||" --> stack")
end

procedure retrieve_register (reg)
  # emit kbox assembly code to:
  # save the value found on the top of the stack
  # to the specified reg

  emit_line ("pop",reg,,,"stack --> "||reg)
end
```

Also, remember that before data can be moved either ***from memory to a register*** or ***to memory from a register*** on a RISC computer, its L-value (address) must be moved into a register (either the ***source address register*** or the ***destination address register***). It is only after this that the R-value (actual data value) may be moved into a register. The following two support procedures would generate the appropriate code.

data flow (memory $\longleftrightarrow$ registers)

```
procedure emit_Lvalue (addr,scope)
  # emit kbox assembly code to:
  # move the address of the identifier
  # into SAR

  if (scope == "global") then
    emit_line ("lda",SAR,"="||addr,,"Lvalue --> SAR")
  else # (scope == "local")
  {
    emit_line ("movi",OR,addr,,"Lvalue --> SAR")
    emit_line ("add",SAR,FP,OR)
  }
end

procedure emit_Rvalue (data_type)
  # emit kbox assembly code to:
  # retrieve the data value found at the memory address
  # currently found in SAR
  # and push the data value onto the top of the stack

  if ((data_type == "INT") |
      (data_type == "STRING") |
      (isPointer(data_type))) then
  {
    emit_line ("ldr",IA,SAR,,"Rvalue --> stack")
    emit_line ("push",IA)
  }
  else if (data_type == "REAL") then
  {
    emit_line ("ldr",FA,SAR,,"Rvalue --> stack")
    emit_line ("push",FA)
  }
  else
  {
    emit_comment ("note: for structured data types, " ||
                  "Rvalue = Lvalue!")
    emit_line ("push",SAR)
  }
end
```

Named constants are handled similarly to global variables; except the data values are immutable – they may not be changed.

Literal constant data can be easily handled using an immediate value within **kcode**.

data flow (literal values)

```
procedure emit_literal (literal_type,literal_value)
  # emit kbox assembly code to:
  # move a constant value to the top of the stack

  if (literal_type == "INT") then
  {
    emit_line ("movi",IA,literal_value,,
               "int literal constant")
    emit_line ("push",IA)
  }
  else if (literal_type == "REAL") then
  {
    emit_line ("movi",FA,literal_value,,
               "real literal constant")
    emit_line ("push",FA)
  }
  else if (literal_type == "STRING") then
  {
    emit_line ("movi",SAR,literal_value)
    emit_line ("push",SAR)
  }
  else # (literal_type == "POINTER")
  {
    emit_line ("movi",IA,0,,"NULL pointer")
    emit_line ("push",IA)
  }
end
```

14.2 Simplified Read and Write Statements

Recall that **kcode** provides four very useful and very basic input / output instructions:

get *fmt*

getln

put *fmt*

putln

But also recall that they are very specific to the top of the system stack!

The **get** instruction transfers a data value from the keyboard to the top of the stack; the ***fmt*** element defines how the data is to be interpreted.

Normally data value transferred to the top of the stack needs to be moved from the stack and stored somewhere – to an address previously moved to the destination address register (DAR) and then saved to the stack.

simplified read

```
procedure emit_read (data_type)
  # emit kbox assembly code to:
  #   - read value from the keyboard
  #   - store value on the top of the stack
  #   - data type determines fmt

  if (\data_type) then
  {
    if (data_type == "INT") then
    {
      fmt := "INT"
      reg := IA
    }
    else
    {
      fmt := "FLT"
      reg := FA
    }
    emit_line ("get",fmt,,,"read")
    emit_line ("pop",reg,,,"store input value")
    emit_line ("pop",DAR)
```

```
    emit_line ("str",reg,DAR)
  }
  else
    emit_line ("getln",,,,"eol")
end
```

Please note within the code above, if the data_type is *not specified* then any remaining characters until an **end-of-line** marker are ignored.

Simarly, the **put** instruction transfers the data value found at the top of the stack to the monitor; the ***fmt*** element defines how the data is to be displayed.

Normally the data value found at the top of the stack is evaluated in the immediately preceding assembly language instructions. Once the data value has been displayed there is no further action required.

simplified write

```
procedure emit_write (data_type)
  # emit kbox assembly code to:
  #   - write value to the monitor
  #   - value is found on the top of the stack
  #   - data type determines fmt

  if (\data_type) then
  {
    if (data_type == "INT") then
      fmt := "INT"
    else if (data_type == "REAL") then
      fmt := "FLT"
    else if (data_type == "STRING") then
      fmt := "STR"
    else if (isPointer(data_type)) then
      fmt := "PTR"
    else
    {
      stop ("invalid write data type ("||data_type||")")
    }
    emit_line ("put",fmt,,,"write")
  }
  else
    emit_line ("putln",,,,"eol")
end
```

Please note within the code above, if the data_type is *not specified* then an **end-of-line** marker is generated.

14.3 Simplified Assignment Statements

At this point in our **kize** compiler, the simple assignment statement will be limited to the following two basic forms:

variable := variable ;

variable := literal_constant ;

The steps for implementing such an assignment statements are quite straight forward:

simplified assignment

```
procedure emit_assign (target_type,expr_type,data_size)
  # emit kbox code to assign data value at top of stack
  #   to target location also found attop of stack
  # target_type specifies the data type for the transfer

  emit_line ("nop",,,,"assignment")
  if (target_type == "INT") then
  {
    if (expr_type == "REAL") then
    {
      emit_line ("pop",FA)
      emit_demote (FA)
    }
    else if (expr_type == "INT") then
      emit_line ("pop",IA)
    else
      stop ("invalid assignment: " ||
            target_type  || " <-> " || expr_type)
    emit_line ("pop",DAR)
    emit_line ("str",IA,DAR)
  }
  else if (target_type == "REAL") then
  {
    if (expr_type == "INT") then
    {
      emit_line ("pop",IA)
      emit_promote (IA)
    }
    else if (expr_type == "REAL") then
      emit_line ("pop",FA)
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
    emit_line ("pop",DAR)
    emit_line ("str",FA,DAR)
  }
end
```

14.4 Promotion and Demotion

All of the above presumes that the L-value and the R-value in the assignments are the same data type: either both INT or both REAL. However, if that is not the case then the compiler must make adjustments so that only compatible assignments are ultimately executed. Certainly INT and REAL data types should be compatible data types! INT can be upgraded (or promoted) to REAL; and REAL can be downgraded (or demoted) to INT – with the obvious side effect of truncating decimal digits. Some programming languages may perform implicit promotion/demotion of data types when necessary; other languages may require explicit **casting** of one data type into another. What is allowable is defined by the given programming languages, as well the steps necessary to perform these actions.

kize will perform promotion / demotion implicitly, but only for the atomic data types INT and REAL. No other castings will be permitted.

To promote an INT type to REAL, the following would be appropriate

promotion

```
procedure emit_promote (reg)
  if (reg[1] == "I") then
  {
    dest := "F" || reg[2:0]
    emit_line ("i2f",dest,reg,,"promote")
  }
  else
    stop ("invalid int register ("||reg||")")
end
```

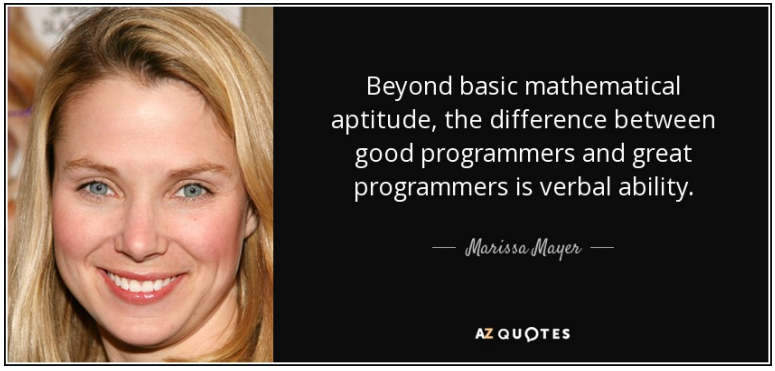
To demote a REAL type to an INT, the following would be appropriate

demotion

```
procedure emit_demote (reg)
  if (reg[1] == "F") then
  {
    dest := "I" || reg[2:0]
    emit_line ("f2i",dest,reg,,"demote")
  }
  else
    stop ("invalid real register ("||reg||")")
end
```

14.5 Phase Two Implementation

- implement data flow
  - memory to/from registers
  - registers to/from stack
  - handling Lvalues
  - handling Rvalues
  - handling literal constants
- simplified read and write statements
- promotion and demotion of data
- simplified assignment statement


Beyond basic mathematical aptitude, the difference between good programmers and great programmers is verbal ability.
— Marissa Mayer —
AZ QUOTES

Chapter 15

Phase Three: Simple Statements

It's Not as Simple as It Looks!

Now that we are assured that we can input and output data correctly and that we can manipulate the data accurately, we are ready to implement computation, comparison, and logic. Our assignment statement will be extended to include the most general forms of expressions, incorporating combinations of arithmetic, comparison, and logical operations. The **kize** compiler will incorporate the standard rules for these operators – standard precedence hierarchy and left-to-right calculation if equal. Logical operations will not require a special data type; INT type will suffice for logical values (with false being represented by zero and true being represented by nonzero).

The concluding sections of this chapter will discuss the full implementation of Input and Output utilizing the simple routines discussed in the previous chapter together with the implementation of the infamous Goto Statement and the seldom-used Empty Statement. These last few items will seem almost trivial after the detail necessary to properly handle the full implementation of Assignment and Expressions.

15.1 Expressions and Assignment

The following page highlights the **kize** grammar as it pertains to expressions. The production rules for expressions in most programming languages tend to look very similar! Typically production rules convey primarily the **syntax** for the language; but quite often the **semantics** for the language can also be incorporated into the production rules as well!

Observe the following two important facts concerning the following **kize** grammar:

- the progression of production rules:
  expression $\longrightarrow$ *simple_expression* $\longrightarrow$ *term* $\longrightarrow$ *factor*
  actually defines the hierarchy (precedence) of operators:
  mulops BEFORE addops BEFORE compops

- the *more_expression* and *more_term* production rules
  imply adjacent **addops** and adjacent **mulops**
  are to be evaluated *left to right*
  i.e., in the order they are encountered

One feature that is not implicit within the grammar is the semantic rule regarding **short-circuit evaluation** of logical operators AND and OR. In a sequence of logical disjunctions (ORs), the first **true** value encountered eliminates the need for any further evaluation; similarly, in a sequence of logical conjunctions (ANDs), the first **false** value encountered eliminates the need for any further evaluation.

Short-circuit evaluation will definitely enhance program execution time, but it also enables simplification of logical expressions, especially ones that potentially might follow a NULL pointer (to be discussed in a later chapter).

KIZE grammar: expressions

```
assignment_statement    named_item := expression ;

expression              simple_expression comparison

comparison              compop simple_expression
comparison

simple_expression       sign term more_terms

more_terms              addop term more_terms
more_terms

term                    factor more_factors

more_factors            mulop factor more_factors
more_factors

factor                  named_item
factor                  ( expression )
factor                  NOT factor
factor                  literal

named_item              IDENTIFIER

addop                   +
addop                   -
addop                   OR

mulop                   *
mulop                   /
mulop                   %
mulop                   NOT

sign                    +
sign                    -
sign

compop                  =
compop                  <>
compop                  >
compop                  >=
compop                  <
compop                  <=
```

15.1.1 Arithmetic

Out of all the standard operations that can be performed by a computer, the arithmetic operations are probably the most familiar and most natural! Since grade school we have been taught (1) multiplication and division before addition and subtraction and (2) if a tie, then left-to-right. We are old enough now that we can refer to these rules as: **precedence / hierarchy of operators** and **left-associativity**!

For readers who are preparing to compete on ***Jeopardy***, **exponentiation** is an example of a **right-associative** operator:

$$4 \wedge 2 \wedge 3 = 4 \wedge (2 \wedge 3) = 4 \wedge 8 = 65{,}536$$

And it has a higher precedence than addition, subtraction, multiplication, and division. At this point you should no longer have the urge to ask the question, "Why does **kize** not have any exponentiation?"

However, arithmetic on the computer does have to deal with at least two types: INT and REAL. And the programmer does have to deal with the limitations of each representation.

INT values have an advantage in precise storage within memory locations, but it is limited in the number of values represented (at most 2^{64} values in 64-bit storage). Hence there is always the possibility of data **overflow** when manipulating data – i.e., the combination of two representable integer values exceeds any 64-=bit representable integers.

REAL values are not precise in their representation; they are the closest approximation using a variation on scientific notation. However, REAL computations not only have the possibility of data **overflow** but also the possibility of data **underflow** where numbers very close to zero can no longer be legitimately represented.

These issues impact any and all programming languages. It is not our responsibility to somehow ***fix*** these problems; but it is our responsibility to ***be aware*** of these issues and the fact that they may occur during program execution.

However, two very specific issues that remain to be addressed – first, mixing modes within a single calculation; and second, questions pertaining to INT division.

mixing modes

All the arithmetic operations (addition, subtraction, multiplication, and division) have this issue. What are the semantics for calculations that potentially have one operand of type INT and the other operand of type REAL? We are fortunate that the **kize** programming language limits itself to these two data types. The larger the number of available data types, the larger the number of combinations for possible mixed modes. For example, we could have several fundamental data types each of which is a proper subset of the subsequent item:

$$\text{integers} \subseteq \text{fractions} \subseteq \text{reals} \subseteq \text{complex numbers}$$

Since we are dealing with only two categories (INT $\subseteq$ REAL) and the ASCII string value "INT" precedes the ASCII string value "REAL" in alphabetical order, determining whether or not a data type needs to be promoted becomes a pretty trivial comparison.

Assuming that all previous temporary calculations have been pushed onto the stack:

When performing a ***binary operation*** the typical sequence of steps is:

- retrieve the second operand from the stack
- retrieve the data type of the second operand
- retrieve the first operand from the stack
- retrieve the data type of the first operand
- promote either the first or second operand (if necessary)
- perform the binary operation with compatible operands
- save the data result on the stack
- return the data type of the result

This sequence for evaluating binary operations is true not just for calculations, but they are also valid for comparisons of numeric values. And they even remain true for the logical operations AND,

OR, and NOT! However, in this last case, all operands **must** be INT.

The immediately following pages contain my additions to **assembler.icn** to perform arithmetic operations. The last procedure **emit_get_operands** performs the first five steps in performing a binary operation.

emit addop

```
procedure emit_addop (op,a_type,b_type,c_type)
  # emit kbox code to add/subtract two data values
  # a_type, b_type, and c_type determine registers to use

  emit_get_operands (a_type,b_type,c_type)
  if (a_type == "INT") then
  {
    if (op == "+") then
      inst := "add"
    else if (op == "-") then
      inst := "sub"
    else
      stop ("invalid INT addop (" || op || ")")
    emit_line (inst,IA,IB,IC," perform addop " || op)
    emit_line ("push",IA)
  }
  else if (a_type == "REAL") then
  {
    if (op == "+") then
      inst := "fadd"
    else if (op == "-") then
      inst := "fsub"
    else
      stop ("invalid REAL addop (" || op || ")")
    emit_line (inst,FA,FB,FC," perform addop " || op)
    emit_line ("push",FA)
  }
  else
    stop ("invalid arithmetic data type (" || a_type || ")")
end
```

emit mulop

```
procedure emit_mulop (op,a_type,b_type,c_type)
  # emit kbox code to multiply/divide two data values
  # a_type, b_type, and c_type determine registers to use

  emit_get_operands (a_type,b_type,c_type)
  if (a_type == "INT") then
  {
```

```
    if (op == "*") then
      inst := "mul"
    else if (op == "/") then
      inst := "div"
    else if (op == "%") then
      inst := "mod"
    else
      stop ("invalid INT mulop (" || op || ")")
    emit_line (inst, IA, IB, IC, "perform mulop " || op)
    emit_line ("push", IA)
  }
  else if (a_type == "REAL") then
  {
    if (op == "*") then
      inst := "fmul"
    else if (op == "/") then
      inst := "fdiv"
    else
      stop ("invalid REAL mulop (" || op || ")")
    emit_line (inst, FA, FB, FC, "perform mulop " || op)
    emit_line ("push", FA)
  }
  else
    stop ("invalid arithmetic data type (" || a_type || ")")
end
```

emit get operands

```
procedure emit_get_operands (a_type, b_type, c_type)
  # emit kbox code to retrieve a two operands
  #   from the top of the stack
  # b_type and c_type determine the registers for data
  #   and a_type determines any promotion

  if ((c_type == "INT") | isPointer(c_type)) then
    emit_line ("pop", IC, , , "get operands")
  else if (c_type == "REAL") then
    emit_line ("pop", FC, , , "get operands")
  else
    stop ("unrecognized data type (" || c_type || ")")
  if ((b_type == "INT") | isPointer(b_type)) then
    emit_line ("pop", IB)
  else if (b_type == "REAL") then
    emit_line ("pop", FB)
  else
    stop ("unrecognized data type (" || b_type || ")")
  if ((b_type == "INT") & (a_type == "REAL")) then
    emit_promote (IB)
  if ((c_type == "INT") & (a_type == "REAL")) then
    emit_promote (IC)
end
```

INT division

INT division does not always yield an INT result; very often there is a non-zero remainder! That is why fractions came into being. REAL division generates another decimal value, so it works fine. Because of this, there are actually TWO kinds of INT division: / and %.

The former performs traditional division and returns the quotient (the number of times the divisor evenly divides the dividend); the latter performs traditional division and returns the remainder (anything left over after performing division). All this is pretty straightforward *as long as* we are working with **positive** integers. Introduce negative integers into division and things can get weird.

Simple truncation division is fairly straight forward to implement:

11/4 = 2 -11/4 = -2 11/-4 = -2 -11/-4 = 2

We just ignore signs, divide, count signs to determine the sign of the result..

The remainder can be retrieved using the formulas:

numerator = denominator x quotient + remainder (or)
remainder = numerator - denominator x quotient

The remainder for the four calculations above would be (in order):

3 -3 3 -3

Observe that moving a minus sign around between the numerator and the denominator **does not** change the value of the quotient, but it **does** change the value of the remainder! So we should probably make a rule that limits the location of any minus sign to the **numerator only** (i.e., the denominator in INT division must be positive). That would seem to solve our little problem!

Or does it?????

For the mathematically inclined reader, the remainder of integer division is sometimes referred to as the **modulus**. For a positive integer n, $n \geqq 2$, **modular arithmetic** defines addition, subtraction, multiplication, and division on positive integers: 0, 1, 2, . . . , n-1. It does so by only retaining the remainder following a division

by the positive integer **n**. However, that remainder is required to be in the range 0, 1, 2, ... , n-1.

So what should the remainder be for -11/4?

Should the value be -3 as calculated above?
Or should the value be adjusted to (-3) + 4 = 1 to be be non-negative?

And if we adjust the remainder, does that not also spill back onto the quotient. Should the quotient also be adjusted to -3 = (-2) - 1, to preserve the fundamental relationships of INT division?

numerator = denominator x quotient + remainder

This brings us to our first implementation option.

We will encounter various implementation options as we discuss the **kize** compiler. These options allow you to choose how complete you want your implementation to be. Some options might be skipped entirely; others might be a choice between several possible candidates. At this time you get to choose how fancy you want INT division to be in your compiler?

Simpler Option:
Implement the operators / and % as originally described:

11/4 = 2 11%4 = 3 -11/4 = -2 -11%4 = -3

Harder Option:
Implement the operators / and % as adjusted for **modular arithmetic**:

11/4 = 2 11%4 = 3 -11/4 = -3 -11%4 = 1

Also remember that this will be **your** compiler and that you can (1) postpone any decision and return to it later or (2) implement a simpler option now and replace it with a more difficult option in the future.

15.1.2 Logic

There is not a LOGICAL or a BOOL data type explicitly defined in the **kize** programming language. But that should not be construed to mean there are no tests and no logical operations. Rather than define a unique BOOL data type, **kize** utilizes the data type INT to represent boolean values.

Specifically, the boolean value **false** is represented by the INT value **zero**, and the boolean value **true** is represented by the INT value **nonzero**.

Short-circuit logic is fairly easy to implement if one recognizes the subtle difference from a normal binary operation.

- a normal binary operation requires that both operands have been evaluated **prior** to determining the value of the result
- a short-circuit operation **does not** evaluate an operand if it has no bearing on the result

For example, to evaluate the expression

$$\text{expression}_a \text{ AND } \text{expression}_b$$

- First evaluate expression_a.
- If the result is **false**, there is no need to evaluate expression_b the result of the full expression is already determined: **false**.
- If the result is **true**
  the result of the full expression will be the value of expression_b.

Consider for a moment the **kize** grammar elements for the mulops * and / and AND:

kize.grm

```
more_factors          mulop factor more_factors
more_factors
```

Incorporating end-of-phrase markers into the grammar would yield the following:

kize.grm (eop-markers)

```
more_factors            mulop !save_op
                        factor !save_operand
                        !emit_operator more_factorss
more_factorss
```

Consider now the modifications necessary for short-circuit AND:

kize.grm (short-circuit logic)

```
more_factors            AND !recognize_operator
                        !cmp_operand !jmp_done
                        factor !save_operand
                        more_factors
more_factors
```

There is a very important but subtle element that comes into play in implementing short-circuit logic. Immediately after recognizing the OR or AND operator, the compiler must generate assembly language code to determine if the subsequent operand(s) is/are necessary to evaluate. Instead of a the very simple pattern

emit_operand emit_operand emit_operator,

where the operator is implemented in a single assembly language instructions, we now have a pattern

emit_operand check_operand emit_operand.

Note: Implementation of short-circuit logic can **not** be performed in a single sequence of assembly language code. It must be split into smaller fragments that will be interlaced with other compiler elements to perform the task. This is the first time we have encountered a grammar element that required such an implementation.

emit_and

```
procedure emit_and_a (label_false)
  # emit kbox code to perform logical AND on INT values
  # note: implements short-circuit logic,
  #     not simple LAND instruction

  emit_line ("pop",IA,,,"short-circuit and")
  emit_line ("cmp",IA,ZR)
  emit_line ("jeq",label_false)
end

procedure emit_and_b (label_false,label_done)

  emit_line ("pop",IA)
  emit_line ("cmp",IA,ZR)
  emit_line ("jeq",label_false)
  emit_line ("movi",IA,1)
  emit_line ("push",IA)
  emit_line ("jmp",label_done)
  emit_label (label_false)
  emit_line ("mov",IA,ZR)
  emit_line ("push",IA)
  emit_label (label_done)
  emit_line ("nop")
end
```

emit_or

```
procedure emit_or_a (label_true)
  # emit kbox code to perform logical OR on INT values
  # note: but implements short-circuit logic
  #     not simple LOR instruction

  emit_line ("pop",IA,,,"short-circuit or")
  emit_line ("cmp",IA,ZR)
  emit_line ("jne",label_true)
end

procedure emit_or_b (label_true,label_done)

  emit_line ("pop",IA)
  emit_line ("cmp",IA,ZR)
  emit_line ("jne",label_true)
  emit_line ("mov",IA,ZR)
  emit_line ("push",IA)
  emit_line ("jmp",label_done)
  emit_label (label_true)
  emit_line ("movi",IA,1)
  emit_line ("push",IA)
  emit_label (label_done)
  emit_line ("nop")
end
```

15.1.3 Comparison

The last category of operations in an expression is the comparison of numeric values. The six operators are equal (=), not equal (<>), greater than (>), greater than or equal (>=), less than (<), and less than or equal (<=). The meaning of the six operators should be familiar to everyone! The only semantics associated with them has to do with mixing modes (INT and REAL). And the resolution of mixed mode is the promotion of INT data to REAL data as with arithmetic calculations.

Note: The equal sign (=) is a symbol that will drive you nuts as you move from one programming language to another. Does = represent assignment or does = represent testing for equality? Or does := represent assignment and == represent testing for equality? And what is preferable ! = or <>?

Note Also: The more types in a programming language the more comparison operators are required. The basic building blocks for representing these operators are the symbols

$$\{ = > < ! \sim \backslash \}$$

combined into longer and longer sequences to differentiate between the underlying types.

The following page contains the **assembler.icn** code to implement the six comparison operators in **kcode**. Because we have now returned to the more normal evaluation of a binary operator, the code can be implemented in a single fragment of assembly language code.

emit_compop

```
procedure emit_compop (op,a_type,b_type,c_type)
  # emit kbox code to compare two data values
  # a_type, b_type, and c_type determine registers to use

  emit_get_operands (a_type,b_type,c_type)
  if ((a_type == "INT") | isPointer(a_type)) then
  {
    inst := "cmp"
    emit_line (inst,IB,IC,,"perform comparison "||op)
  }
  else if (a_type == "REAL") then
  {
    inst := "fcmp"
    emit_line (inst,FB,FC,,"perform comparison "||op)
  }
  else
    stop ("invalid arithmetic data type ("||a_type||")")
  label_true :=  @name_generator
  label_false := @name_generator
  label_done :=  @name_generator
  if (op == "=") then
    emit_line ("jeq",label_true)
  else if (op == "<>") then
    emit_line ("jne",label_true)
  else if (op == ">") then
    emit_line ("jgt",label_true)
  else if (op == "<") then
    emit_line ("jlt",label_true)
  else if (op == ">=") then
    emit_line ("jge",label_true)
  else # (op == "<=")
    emit_line ("jle",label_true)
  emit_label (label_false)
  emit_line ("movi",IA,"0")
  emit_line ("jmp",label_done)
  emit_label (label_true)
  emit_line ("movi",IA,"1")
  emit_label (label_done)
  emit_line ("push",IA)
end
```

15.1.4 Unary Operators

Before we rush to the end of this chapter, it is important to remember two necessary and easily overlooked **unary operators**: arithmetic negation and logical negation.

Arithmetic expressions often incorporate a leading plus or minus sign before the first term. The plus sign is obviously redundant, unnecessary, and could easily be omitted, but the minus sign requires that the numeric value immediately following be changed to the opposite sign. Hence, a compiler must include this unary operation.

Logical operators have three basic flavors: AND, OR, and NOT. Since NOT is a unary operator there is no need for special short-circuit logic! We simply have to reverse the logical value.

The following code fragments from **assembler.icn** would implement these to useful operators.

emit_neg

```
procedure emit_neg (data_type)
  # emit kbox code to negate a single data value

  if (data_type == "INT") then
  {
    emit_line ("pop",IA)
    emit_line ("neg",IA)
    emit_line ("push",IA)
  }
  else if (data_type == "REAL") then
  {
    emit_line ("pop",FA)
    emit_line ("fneg",FA)
    emit_line ("push",FA)
  }
  else
    stop ("invalid arithmetic data type (" || data_type || ")")
end
```

emit_not

```
procedure emit_not (data_type)
  # emit kbox code to perform logical NOT on INT value

   if (data_type == "INT") then
   {
     emit_line ("pop",IA,,"logical not")
     emit_line ("lnot",IA)
     emit_line ("push",IA)
   }
   else
     stop ("invalid logical NOT data type (" ||
           data_type || ")")
end
```

15.1.5 Strings

And now we arrive at the data type STRING, which is **not** really available in the programming language **kize**. The STRING type is available only for use as a constant – either a literal constant or a named constant. They can not be used in any other context.

STRINGs facilitate writing prompts for data input and facilitate formatting the display for data output. **kize** implements the STRING data type the same way as the C programming language. The data value is an address for a sequence of ASCII characters terminated by \0 (end-of-string marker).

The **kize** programming language does not allow for any input of STRING data! All STRING constants are defined either as a ***named constant*** defined in the global declarations or as a ***literal constant*** scattered throughout the executable code. A STRING data value may appear within the evaluation of an expression, but it **must** pass through untouched by any unary or binary operation! In English, that means the expression on the right-hand-side reduces down to a single constant value (either named or literal).

15.1.6 Assignment

This topic is actually a bit of a let-down after adding all the additional features to evaluating expressions and differentiating between normal binary operations and short-circuit logic. Disappointment comes primarily from the fact that the **emit_assign** procedure to generate **kcode** pretty much works as is with no modification!

The only modification that might be necessary at this point would be a test for possible STRING data and a resulting semantics error.

However, be forewarned that we will return to this topic later – when we consider **structured data types** and again when we consider **pointers**. Both topics will require significant additional testing of types and, in the case of structured data types, more complicated **kcode** instructions to perform the actual transferral of data.

15.2 Input and Output

If everything went well in the preceding chapter, the the implementation of input and output for the **kize** programming language should now be very straightforward. In the following description, the ***first line*** represents the basic **kize** input and output statements and the ***second line*** represents the simplified versions discussed in the previous chapter.

- READ[LN] (A, B, C, ...) →

  emit_read (type_a) ; **emit_read** (type_b) ; **emit_read** (type_c) ; ...
  [**emit_read** () ;]

- WRITE[LN] (E_1, E_2, E_3, ...) →

  emit_write (type_{e1}) ; **emit_write** (type_{e2}) ; **emit_write** (type_{e3}) ; ...
  [**emit_write** () ;]

By focusing on our simplified versions of input and output in the previous chapter (i.e., one item at a time), the full implementation of **kize** input and output (i.e., an arbitrary number of items) should be easily implemented.

One cautionary note: emit_read presumes that the destination address has already been pushed onto the stack prior to its activation; **emit_write** presumes that the expression has already been evaluated and the result has been pushed onto the stack prior to its activation. So each of the **emit_read**/**emit_write** statements above are interlaced with appropriate **kcode** instructions to push the appropriate information onto the stack.

15.3 Goto and Empty Statements

The last two items in this chapter are two of the most obvious things to implement!

The first item – the Goto Statement – requires the compiler to identify the external name for the label, retrieve its internal name from the local label table, and then branch to the internal label.

The second item – the Empty Statement – requires the compiler to either do absolutely nothing or to generate the assembly language instruction to do absolutely nothing!

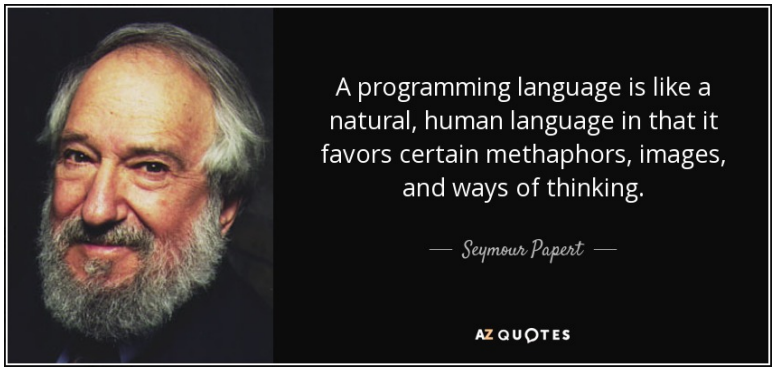
The following code fragments from **assembler.icn** would implement these to useful statements.

emit_goto and emitempty

```
procedure emit_goto (lvalue)
  emit_line ("jmp",lvalue,,,"goto statement")
end

procedure emit_empty ()
  emit_line ("nop",,,,"empty statement")
end
```

15.4 Phase Three Implementation

- full implementation of expression evaluation, including:
  - precedence
  - left-associativity
  - mixed modes and promotion
  - calculation, comparison, and essential logic
  - integer mode division: simple or modulo (*optional feature*)
- representation of BOOL data as INT
- representation of STRING data
- assignment statement (now incorporating expressions)
- input and output statements (as defined in the **kize** grammar)
- goto statement
- empty statement


A programming language is like a natural, human language in that it favors certain methaphors, images, and ways of thinking.
— Seymour Papert —
AZ QUOTES

Chapter 16

Phase Four: Control Statements

So Sweet

But Such a Control Freak!

Now that we finally have implemented calculation, comparison, and essential logic into the **kize** programming language, we are at the point where we can introduce control structures into our algorithms. Right now our compiler generates single-pass, one-and-done, executable code with the exception of the infamous Goto statement. The Goto statement allows us the ability to hop-scotch throughout our code and possibly go into an infinite loop.

But real programming power comes from the ability to make decisions based up the current status of the data within the program and to repeat important steps in a process until we achieve a desired outcomes.

Back in the 1970s a very hot topic was **structured programming**, sometimes euphemistically referred to as **goto-less programming**. E. W. Dijkstra had demonstrated that any algorithm can be implemented using only three basic logical structures.

- sequence
- selection
- iteration

Goto statements were unnecessary! Not only unnecessary but viewed as counterproductive. Readability and maintainability of source code were demonstrated to be **inversely** proportional to the number of goto statements found within.

However, in a simple language like **kize**, a goto statement can be used to solve a number of problems adequately without having to resort to newer features like **break** or **next** and newer concepts like **exception handling**.

As we saw previously with the implementation of short-circuit logic, we may not always be able to implement a certain feature of our compiler within a single contiguous piece of coding. We may need to interlace elements of our control structures among other compiler components.

It is also important to note at the outset of our discussion that control structures typically require branching to specific locations within coding while omitting other areas. We will certainly have to make use of our name generator to create these internal labels.

16.1 Sequence

SEQUENCE is just the ordinary every day garden variety **statement list** given new branding! Most programming languages execute sequentially unless specifically directed to do otherwise.

The reason for the new name and requiring delimiters (DO and END in **kize**) is that other control structures in the programming language are often defined referencing a single statement in their syntax. In order to execute a block of statements as a single item, we need to introduce some form of mandatory grouping – hence the COMPOUND statement or sequencing.

Your implementation of a **kize** compiler **must** include the compound statement!

16.1.1 The DO ... END Statement

Syntax:

DO
< statement_list >
END
;

Compilation:

- recognize the token DO
- compile each individual statement within in the statement list
- recognize the toke END
- recognize the token ;

The keywords DO and END do not even have to generate any assembly language code in and of themselves. If you would prefer to highlight the presence of a control structure within the assembly code your compiler is generating, I would suggest adding the following two elements to **assembler.icn**.

emit do

```
procedure emit_do_a ()
  emit_line ("nop",,,,"do statement")
end

procedure emit_do_b ()
  emit_line ("nop",,,,"end do")
end
```

16.2 Selection

SELECTION simply means choice. Programming languages typically include the following two flavors:

- IF ... THEN ... ELSE ...
- CASE

Your **kize** compiler **must** include the IF ... THEN ... ELSE ... statement; the CASE statement is **optional**. The CASE statement is definitely harder to implement. The simpler IF ... THEN ... ELSE ... statement can essentially be used to mimic any other SELECTION statements, if they are nested correctly. Hence, implement the first statement immediately and return to the second statement in the near future.

16.2.1 The IF ... THEN ... ELSE ... Statement

Syntax:

IF (< test >)
THEN < statement >
[ELSE < statement >]
;

Compilation:

- recognize the token IF
- recognize the token (
- compile the test expression
- recognize the token)
- *determine whether the test expression result is true or false*
  if false (i.e., zero) branch to ***labelfalse***
- recognize the token THEN
- compile statement (true option)
- *branch to* ***labeldone***
  insert ***labelfalse***
- [recognize the token ELSE
  compile statement (false option)]

- *insert* ***labeldone***
- recognize the token ;

Once again, the keywords IF and THEN and ELSE do not generate any assembly language code in and of themselves. As we discussed previously, if you would prefer to highlight the presence of a control structure within the assembly code your compiler is generating, I would suggest adding the following two elements to **assembler.icn**.

emit if

```
procedure emit_if_a ()
  emit_line ("nop",,,,"if statement")
end

procedure emit_if_b ()
  emit_line ("nop",,,,"end if")
end
```

Also, please note that you need to generate **two** labels for the logic in an IF ... THEN ... ELSE ... statement – a ***labelfalse*** and a ***labeldone***. These labels are required to implement the appropriate branching components.

emit test

```
procedure emit_test_a (label_false,label_done)
  emit_line ("pop",IA,,,"test result")
  emit_line ("cmp",IA,ZR)
  emit_line ("jeq",label_false)
  emit_line ("nop",,,,"then clause")
end

procedure emit_test_b (label_false,label_done)
  emit_line ("jmp",label_done)
  emit_label (label_false)
  emit_line ("nop",,,,"else clause")
end

procedure emit_test_c (label_false,label_done)
  emit_label (label_done)
  emit_line ("nop",,,,"end test")
end
```

16.2.2 The CASE Statement

Syntax:

CASE (< expression >) OF
{ NUMBER : < statement > } *
DEFAULT : < statement >

Compilation:

- recognize the token CASE
- recognize the token (
- compile the case expression
- recognize the token)
- *save the case expression value to the stack*
- repeat the following as necessary
  - determine the literal constant NUMBER
  - *retrieve CASE expression value from the stack*
    resave CASE expression value to the stack
    compare NUMBER with CASE expression value
    if not equal branch to ***labelnext***
  - recognize the token :
  - compile the statement
  - *branch to* ***labeldone***
    insert ***labelnext***
- recognize the token DEFAULT
- *throw away the case expression value*
- recognize the token :
- compile the statement
- *insert* ***labeldone***
- recognize the token ;

We must be especially careful to retain the original case expression value throughout the processing of the case list. We do not know which of the following case list items may be a match. As a result, any time we refer to the case expression value to compare it with a test value, we need to restore the case expression value to the top of the stack. It is only when we encounter the DEFAULT clause in the case statement that we realize the case expression value is no longer necessary.

Furthermore, we need to generate as many labels for the CASE statement logic as there are unary numbers to test: ***labelnext*** is a *generic* reference to such labels. On the other hand, ***labeldone*** is a *single* label to exit the CASE statement. Note that ***labeldone*** will be generated first immediately upon recognizing the CASE statement; and that the ***labelnext*** labels will be generated as needed as the compiler processes through the CASE statement.

The final ***labelnext*** and ***labeldone*** will appear consecutively in your generated assembly language code. This may look a bit weird but it is actually correct!

emit case

```
procedure emit_case_a ()
  emit_line ("nop",,,,"case statement")
end

procedure emit_case_b ()
  emit_line ("nop",,,,"end case")
end

procedure emit_case_compare (case_value,label_next)
  emit_line ("movi",IB,case_value,,"case value");
  emit_line ("pop",IA,,,"target value")
  emit_line ("push",IA,,"resave target value")
  emit_line ("cmp",IA,IB)
  emit_line ("jne",label_next)
end

procedure emit_next_case (label_done,label_next)
  if (\label_next) then
  {
    emit_line ("jmp",label_done)
    emit_label (label_next)
  }
  else
    emit_label (label_done)
end
```

16.3 Iteration

ITERATION simply means repetition. Programming languages typically include the following three flavors:

- WHILE
- REPEAT
- FOR

The **kize** compiler **must** implement the WHILE statement; the REPEAT statement and the FOR statement are **optional** features. The FOR statement is a bit more work to implement than either the WHILE or the REPEAT statements. The simple WHILE statement can mimic any of the other ITERATION statements. Hence, the first is mandatory and the second and the third are additional bonus features!

The WHILE statement and the REPEAT statement are very similar to one another. In fact, the two are equivalent in computing power! It is totally a matter of programmer's preference which one to utilize in any given situation.

WHILE statement

- a top-test loop
- the loop is executed as long as the test is TRUE (FALSE exits)

REPEAT statement

- a bottom-tested loop
- the loop is executed as long as the test is FALSE (TRUE exits)

The major distinction between these two looping structures is that the top-tested WHILE statement need **not** necessarily execute the statement list comprising the body of the loop and the REPEAT statement will normally execute the body of the loop *at least once*! The REPEAT statement is perfect for repeatedly displaying menu options until a valid choice has been selected.

The **FOR statement** is your basic **counted** loop; it is ideal for situations where the required number of iterations is known in advance. I borrowed the syntax for **kize** from **Pascal** to indicate increasing or decreasing counters. Hence, only increments of +1 and decrements of -1 are permitted in **kize**.

16.3.1 The WHILE Statement

Syntax:

WHILE (< test >)
< statement >
;

Compilation:

- recognize the token WHILE
- *insert* ***labeltop***
- recognize the token (
- compile the test expression
- recognize the token)
- *determine whether the test expression result is true or false*
  if false (i.e., zero) branch to ***labeldone***
- compile the statement
- *insert* ***labelnext***
  branch to ***labeltop***
  insert ***labeldone***
- recognize the token ;

You need to generate **three** labels for the logic: ***labeltop***, ***labelnext***, and ***labeldone***.

emit while

```
procedure emit_while_a (label_top)
  emit_label (label_top)
  emit_line ("nop",,,,"while statement")
end

procedure emit_while_b (label_done)
  emit_label (label_done)
  emit_line ("nop",,,,"end while")
end
```

16.3.2 The REPEAT ... UNTIL ... Statement

Syntax:

REPEAT
< statement_list >
UNTIL (< test >)
;

Compilation:

- recognize the token REPEAT
- *insert* ***labeltop***
- compile each individual statement in the statement list
- *insert* ***labelnext***
- recognize the token UNTIL
- recognize the token (
- compile the test expresssion
- recognize the token)
- *determine whether the test expression result is true or false if false (i.e., zero) branch to* ***labeltop***
- *insert* ***labeldone***
- recognize the token ;

You need to generate **three** labels for the logic: ***labeltop***, ***labelnext*** and ***labeldone***.

emit repeat

```
procedure emit_repeat_a (label_top)
  emit_label (label_top)
  emit_line ("nop",,,,"repeat statement")
end

procedure emit_repeat_b (label_done)
  emit_label (label_done)
  emit_line ("nop",,,,"end repeat")
end
```

16.3.3 The FOR ... TO / DOWNTO ... Statement

Syntax:

FOR IDENTIFIER := < initial >
{ TO | DOWNTO } < final >
< statement >
;

Compilation:

- recognize the token FOR
- recognize the token IDENTIFIER
- *move its L-value to SAR*
  confirm it is an INT variable
  save L-value to the stack
- recognize the token :=
- compile the start expression
  R-value will be on the stack
- *perform the assignment*
  IDENTIFIER L-value is on the stack
  as is the expression R-value
  save L-value back on the stack
- recognize either TO or DOWNTO token and remember direction
- compile the stop expression
  R-value will be on the stack
- *insert* ***labeltest***
  retrieve stop R-value from the stack to IB
  retrieve counter L-value from the stack
  retrieve counter R-value from memory to IA
  compare registers IA to IB
- if TO *branch greater than to* ***labeldone***
  if DOWNTO *branch less than to* ***labeldone***
- *resave counter L-value to the stack*
  resave stop R-value to the stack
- compile the statement

- *insert* ***labelnext***
  retrieve stop R-value from the stack to ID
  retrieve counter L-value from the stack
  retrieve counter R-value from memory to IA
- if TO *increment IA*
  if DOWNTO *decrement IA*
- *store updated counter R-value back to memory*
  save counter L-value back to the stack
  save stop R-value back to the stack
  branch to ***labeltest***
- *insert* ***labeldone***
  discard stop R-value from the top of the stack
  discard counter L-value from the top of the stack

You need to generate **three** labels for the logic: ***labeltest***, ***labelnext***, and ***labeldone***.

Observe that a counted loop is complicated by the fact that two items must be remembered throughout execution of the assembly language code: the IDENTIFIER and the stop value. We have to initialize the IDENTIFIER with the start value only at the beginning of the loop. But we have to increment or decrement the IDENTIFIER at the end of each iteration, and we have to compare the IDENTIFIER with the stop value ***prior*** to each iteration. Also, recall FOR loops are **top-tested**!.

I have chosen to retain the L-value for the counter and the R-value for the stop value by saving them on the stack! When I increment the counter at the bottom of the loop I store the new counter value at the address popped off the stack; when I test the counter at the top of the loop I compare the counter R-value with the R-value found on the stack.

emit for

```
procedure emit_for_a ()
  emit_line ("nop",,,,"for statement")
end

procedure emit_for_b ()
  emit_line ("nop",,,,"end for")
end

procedure emit_for_setup ()
  emit_line ("pop",IB,,,"set up for loop")
  emit_line ("pop",IA)
  emit_line ("pop",DAR)
  emit_line ("str",IA,DAR)
  emit_line ("push",DAR)
  emit_line ("push",IB)
end

procedure emit_for_test (flag,label_top,label_break)
  emit_label (label_top)
  emit_line ("pop",IB,,,"top test for loop")
  emit_line ("pop",SAR)
  emit_line ("ldr",IA,SAR)
  emit_line ("cmp",IA,IB)
  if (flag == "up") then
    emit_line ("jgt",label_break)
  else
    emit_line ("jlt",label_break)
  emit_line ("push",SAR)
  emit_line ("push",IB)
end

procedure emit_for_increment (flag,label_top,label_next,
                              label_break)
  emit_label (label_next)
  emit_line ("pop",IB,,,"increment for loop")
  emit_line ("pop",DAR)
  emit_line ("ldr",IA,DAR)
  if (flag == "up") then
    emit_line ("inc",IA)
  else
    emit_line ("dec",IA)
  emit_line ("str",IA,DAR)
  emit_line ("push",DAR)
  emit_line ("push",IB)
  emit_line ("jmp",label_top)
  emit_label (label_break)
end
```

16.4 Pseudo Gotos

NEXT statements and BREAK statements are very common in modern programming languages. Personally, I consider them variations on the simpler GOTO statement!

A NEXT statement skips any remaining statements in the current iteration (looping) structure and moves on to the next repetition. In essence it is equivalent to:

goto *labelnext*

A BREAK statement exits the current iteration (looping) structure. In essence it is equivalent to:

goto *labeldone*

Some programming languages incorporate a BREAK statement into the CASE statement as well; program execution will typically continue into subsequent statements without one!. The semantics in **kize** do not allow such *trickle-down* behavior. Hence, a BREAK statement is unnecessary in this context.

However, NEXT and BREAK statements introduce additional book keeping. Since control structures may be nested within others, we need to maintain a list of ***labelnext*** and ***labeldone*** branches, stored in order (most recent loop being first in the list). These requirements describe the behavior of a stack to a tee!

So we will incorporate two stacks into our compiler – the first will be used for NEXT labels (***labelnext***) and the second to be used for BREAK labels (***labeldone***). We need to **push** information onto both stacks at the start of an iteration control structure; we need to **pop** information off from both at the end.

Note: ***As a general rule regarding control structures***, **never** jump *into* the middle of one and **never** jump *out of* the middle of one! The label stacks become corrupted and are no longer usable, if you were to do so!

16.4.1 The NEXT Statement

Syntax:

NEXT ;

Compilation:

- recognize the token NEXT
- *retrieve the label on the top of the* ***labelnext*** *stack*
  a top and ***not*** *a pop operation!*
  branch to ***labelnext***
- recognize the token ;

16.4.2 The BREAK Statement

Syntax:

BREAK ;

Compilation:

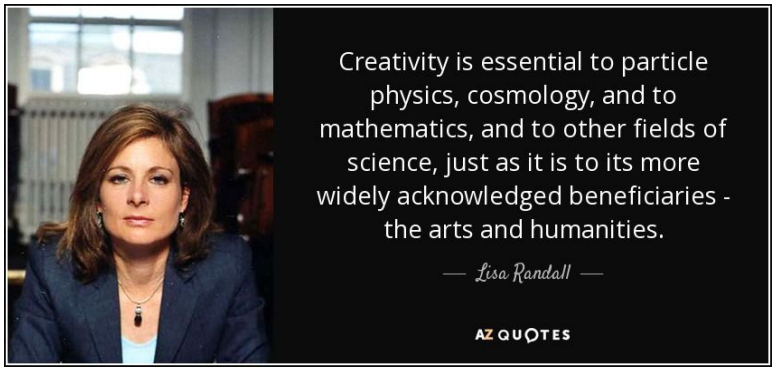
- recognize the token BREAK
- *retrieve the label on the top of the* ***labeldone*** *stack*
  a top and not a pop operation!
  branch to ***labeldone***
- recognize the token ;

Your **kize** compiler **must** implement both the BREAK statement and the NEXT statement. Although they behave in two entirely different ways, their implementations are equivalent in difficulty.

Also, be careful not to **pop** labels off the respective stacks in either BREAK or NEXT! Instead, always *branch* to ***labeldone*** where the first order of business *should be* to remove the obsolete labels from both stacks.

16.5 Phase Four Implementation

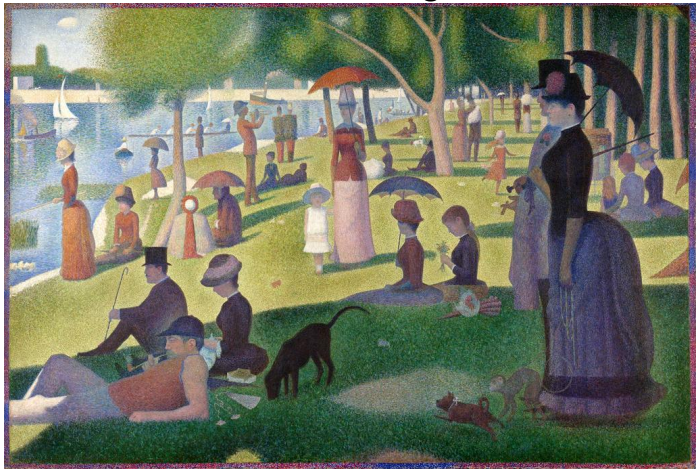
- DO statement
- IF statement
- CASE statement (*optional feature*)
- WHILE statement
- REPEAT statement (*optional feature*)
- FOR TO statement (*optional feature*)
- FOR DOWNTO statement (*optional feature*)
- BREAK statement
- NEXT statement


Creativity is essential to particle physics, cosmology, and to mathematics, and to other fields of science, just as it is to its more widely acknowledged beneficiaries - the arts and humanities.
— Lisa Randall —
AZ QUOTES

Chapter 17

Phase Five: Procedures without Arguments

A World Without Arguments



Subprograms are an especially important topic in computer programming. Our current implementation of **kize** is actually a reasonably complete programming language! Given a specific problem to solve, we have enough tools to implement a solution in the **kize** programming language. Unfortunately, a solution that comes in one humongous program with lots of variables and various specific tasks scattered haphazardly through a single main procedure.

A very powerful strategy that is used in programming is the concept of **top-down design**, also known as **divide and conquer**. A large complicated project is divided into smaller more manageable sub-projects. In turn, each of the sub-projects is analyzed and, if necessary, each is further divided into even smaller more manageable sub-projects. Ultimately, the solution to the original project is implemented in coordinating the activation of these smaller components at the appropriate time and in the appropriate order.

Why should a programmer now take this organized hierarchical structure and mix it all together in a single main procedure? Why not retain these focused blocks of code as cohesive units in their own right? That is the motivation behind the concept of a subprogram or a subalgorithm.

Subprograms within a given programming language may go by several different names! FORTRAN originally distinguished between FUNCTIONs and SUBROUTINEs.

Subprograms also encompass a range of new topics and possible implementation options. We will briefly discuss several of the more common options – primarily to put them in context with the others. But we will also focus on the choices I have selected for the **kize** programming language.

Some of the implementation options include:

- nomenclature:
  functions **versus** subroutines
- programming language syntax:
  functions **and** subroutines **or** functions *only* **or** procedures *only*
- function return types:
  atomic only **or** any defined type

- argument transfer:
  CALL BY VAL **or** CALL BY VAR **or** other transfer mechanism
- storage location:
  global **or** local
- storage allocation:
  static, automatic, **or** dynamic
- scope:
  simple **or** nested

17.1 Terminology

Functions and Procedures

As mentioned previously the programming language FORTRAN distinguished between two types of subalgorithms:

- A **function** performs a set of calculations / operations to ultimately return a *single value*. This definition is obviously modeled on the traditional definition of a mathematical function.
- A **subroutine** performs a set of calculations / operations, but a subroutine does NOT return a *single value*. It might modify nothing at all; or it might modify several values as a result of execution.

One of the interesting aspects of programming language design is the naming conventions for subalgorithms. FORTRAN distinguished functions from subroutines by giving each its own descriptive name. Since then, programming languages have tended to define subalgorithms using a single keyword: either **function** or **procedure**. If a subalgorithm had a return value, the syntax for its declaration provided for an *optional* return type in its syntax.

Which immediately asks the question "What data types are allowable as a return type?" Surely atomic data types must qualify – they are basically built into the language. But what about types the programmer might implement (structured types)? We will discuss this question in more detail later, when we study structured types specifically. But for now and for specifically the **kize** programming language, the simple answer is ***structured data types may not be used as return types***!

Call by Value / Call by Variable

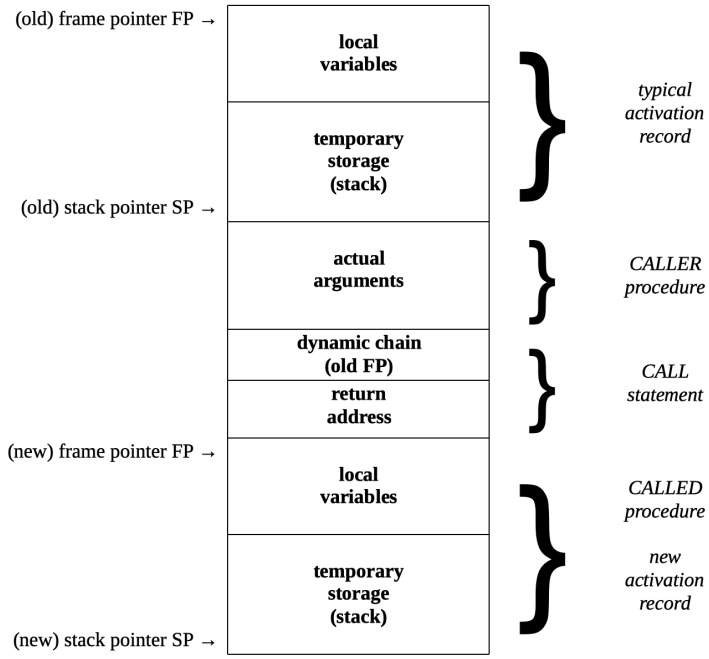
The next obvious questions to consider would be "How do we transfer information into a subalgorithm?" and "Can the subalgorithm modify the information or not?" FORTRAN introduced two data transfer mechanisms: **call by value** and **call by variable**. Other programming languages have suggested others, but these two original techniques seem to be remain dominant.

Call by value transfers a ***copy of the data value*** so no changes can be made to the original; **call by variable** transfers the ***address of the data value*** so any changes immediately propagate back to the original item.

The Activation Stack

Since subalgorithms are activated (CALLed) and concluded (RETURNed), an obvious choice for an appropriate data structure to manage these operations would be a stack. Such a stack would be comprised of individual **activation records**, each one having a similar structure: *storage* for local variables, *temporary storage* for calculations in progress, and *transitions* into the next **activation record** – all this information being stored onto the system stack.

The following diagram illustrates this structure:



Local Versus Global Variables

The difference between **local variables** and **global variables** is much more involved than just the location in the source code where they are declared.

Global variables are known and accessible across the entire program; **local variables** are known and accessible within the specific subprogram in which they are declared. Hence, storage allocation for **global variables** is typically done in the **data** segment, using main memory which can be accessed by direct addressing. Storage allocation for **local variables** is typically done using offset addressing from the frame pointer:

FP + *offset*

This is the first obvious distinction between the two.

These offset values must be calculated while processing both the formal argument list and the local declarations for a procedure. As illustrated in the diagram above, actual arguments will be found ***above*** the frame pointer (FP) and as a result will have **positive offset** values; local variables will be found ***below*** the frame point (FP) and will have **negative offset** values.

Static, Automatic,Dynamic Storage

The second obvious distinction between the two is that global declarations are always there, while local declarations come into existence when the procedure is CALLed and go out of existence when the procedure RETURNs. The storage for global variables is called **static allocation**; the storage for local variables is called **automatic allocation**. A third category for storage is called **dynamic allocation**.

For those of you familiar with pointers and the two fundamental commands: **new** and **dispose**, you already know the power of dynamic structures such as linked lists, binary trees, and graphs. Dynamic allocation requires the use of a memory heap and the implementation of supporting routines to allocate memory from the heap, to return memory to the heap, and to perform *garbage collection* when the heap becomes severely fragmented.

These topics are certainly important, but they may prove to be beyond what can be accomplished in a single introductory text.

We postpone any discussion of dynamic allocation until the very end and as an optional chapter!

Scope for Variables

For the **kize** programming language I have specifically chosen one of the simplest forms of scope for variables.

- A global variable is known throughout the entire program, ***unless*** it is **redeclared** within a given procedure – in which case the local declaration takes precedence.
- A local variable is known only within its procedure; however, the same identifier may be **redeclared** within any other procedure – in which case it will be a completely different entity.

I have chosen to avoid the more complicated form of scope typical of Algol-like languages (**PL/I** and **Pascal**). Such programming languages allow procedures to also be defined within other procedures. Each of these structures have their own scope – hence the term **nested scope**!

The simple scope I have chosen for **kize** is compatible with the activation stack structure described earlier– the frame pointer for the caller procedure and the return address back to the caller are the only additional pieces of information required to be placed on the stack at the time a procedure is called (other than the actual arguments). Nested scope is a bit more complicated to implement in that one additional piece of information must also be saved in order to correctly store and retrieve local variable data values.

Nested scope is discussed in a bit more detail for anyone interested in the concluding section of this chapter.

17.2 Relevant Grammar Elements

The complete description of the syntax for procedures is spread across several components in the **kize** grammar. We will review the basic element here.

17.2.1 Declaration

The global procedure declarations are the very first component that we need to address. We did not deal with this topic earlier because our only procedure was the **main procedure** and it was declared *implicitly*. So we return to book keeping information that pertains to the subalgorithms that comprise our **kize** program.

The purpose of these declarations is very similar to the **Pascal** FORWARD directive – it allows for single-pass compilation and it enables recursive calls within subalgorithms. All procedure signatures must appear in the global declarations of a **kize** program. The only exception to this rule is the **main procedure** which is automatically started upon program execution. These declarations guarantee that all procedures have been clearly described prior to their actual definition and prior to any reference (activation) within any procedure.

A significant portion of time should be spent understanding procedure signatures. They will be the foundation for generating the executable code for these procedures later in this chapter. If not done so already, the global symbol table should be augmented with procedure entries at this point. Just as the local variable entries and the local label entries are essential to generating correct expression evaluation and branching, the global procedure entries are essential to correctly activate and subsequently return from subalgorithms.

Syntax:

IDENTIFIER
(< formal_args >)
[: < return_type >]
;

Observe the amount of detail that is available within a procedure signature. It is a complete description of every element necessary to interact with that subalgorithm: its name; the number, type, and calling mechanism for its arguments; and any return type.

In this chapter we will **not** be discussing any formal arguments. This obviously will feel very strange! But formal arguments introduce a level of complexity that I want to avoid for the time being. Formal arguments will be the sole focus of the next chapter.

But wait!

Shouldn't these subalgorithms work on some kind of data in order to generate results? Or do they just make up results at random?

The answer is: we will use **global variables** to "transfer" data to a subalgorithm! Using global variables, the information is already accessible ***everywhere***.

In concluding this discussion regarding the procedure signature, it is important to highlight the following two points:

- the procedure name is a **global** name
  we must insert a procedure entry into the global symbol table
- the items found in the formal argument list all become **local** variables
  we must insert a variable entry into the local symbol table (next chapter!)

17.2.2 Definition

The procedure list is where the various subalgorithms are defined. The syntax for a procedure mimics the syntax for the overall **kize** program.

Syntax:

PROCEDURE IDENTIFIER
< local_declarations >
BEGIN
< statement_list >
END

The procedure signature has already provided all the vital information concerning the subalgorithm to the compiler.

The good news here is that you are already familiar with implementing the executable code that comprises a procedure definition. There is no difference how you generate executable code between the **main procedure** and ***any other procedure***!

The local declarations are similar in nature to the global declarations, except declarations only have two flavors: variables and labels. These items should be handled in exactly the same fashion as for the **main procedure** with the only requirement being that each procedure has its own symbol table. The global tables remain in existence during the duration of the compiler's operation; the local tables come into existence at the beginning of the procedure definition and go out of existence at the end of the procedure definition.

If a procedure is more specifically a **function** (i.e., returns a value), then the executable code must include the instructions for ultimately moving that calculated value to an appropriate register before exiting: **IA** if the return value is INT or **FA** if the return value is REAL. Also note that a **return** statement may appear anywhere within the procedure code you are compiling. Rather than just generating a simple **ret** instruction in assembly language in place, control must instead be transferred to the procedure epilogue which has the responsibility to properly clean up the stack frame before actually returning from the called procedure. More on this will come shortly!

17.2.3 Activation

The mechanism for activating a subalgorithm depends on whether the procedures is a function or a subroutine!

- **procedure**s are activated using a CALL statement
- **function**s are activated using a function reference within an expression

Note: The above requirements are specific to **kize**. I have tried to simplify your work in implementing the compiler. My grammar may be a bit wordy, but it is intended to facilitate your coding of the compiler. Very often, the CALL keyword is **not** required in a particular language grammar! This implies that an IDENTIFIER appearing at the very beginning of statement could be any of these four possibilities:

- a variable reference (starting an assignment statement)
- an array reference (starting an assignment statement)
- a record reference (starting an assignment statement)
- a procedure activation

At least I eliminated that last possibility! That doesn't sound like much, but every little bit helps!

17.3 Assembly Language Implementation

There is very little new assembly language code necessary to implement procedures in **kcode**:

call ***procedure_label***

and

ret

The **call** instruction is essentially a branch instruction to the ***procedure_label***. But it also stores the ***return address*** in the **rp** register.

The **ret** instruction retrieves the value stored in the **rp** register and transfers control back to that statement.

Together these two instructions provide a nice clean mechanism for *calling* and *returning* to and from subalgorithms – except for fact that the **rp** and the **fp** may be modified within the subalgorithm. Hence, we need to preserve these two vital pieces of information from the beginning to the end of subalgorithm execution. That is why a **procedure prologue** and a **procedure epilogue** have very specific roles!

As is usual with compilers, we use the stack to store and retrieve information that is important and must be preserved for a short period of time. The two registers **fp** and **rp** are especially important for proper maintenance of the activation records and the activation stack.

The following page highlights the important responsibilities that must be performed at the time that control is transferred from the CALLER procedure to the CALLED procedure and then immediately following control being transferred back from the CALLED procedure to the CALLER procedure.

17.3.1 Role of the CALLER

The role of the CALLER procedure is essentially fourfold:

- prior to the actual CALL statement,
  set up the actual arguments on the stack
  which in this chapter means ***do nothing!***
- issue the **kcode** call statement:

  call ***procedure_label***
- if the called procedure is a **function**
  then store the return value (**IA** or **FA**) on the stack
  if the called procedure is a **subroutine**
  then ***do nothing!***
- clean up the actual arguments from the stack
  which in this chapter means ***do nothing!***

17.3.2 Role of the CALLED

The role of the CALLED procedure is essentially fivefold:

- set up a new stack frame for the procedure
  save the **rp**, save the old **fp**, set the new **fp** to the current **sp**
  (this is the **procedure prologue**)
- set up storage and offsets for local variables
  using the information found in the local symbol table
  note that local variables are stored below the **fp** (negative offset)
- execute the procedure assembly language coding
- if the called procedure is a **function**
  then move the return value to **IA** or **FA**
  if the called procedure is a **subroutine**
  then ***do nothing!***
- clean up the now obsolete stack frame for the procedure
  set the **sp** to the current **fp**, retrieve the old **fp**, retrieve the **rp**
  execute the actual **ret** instruction
  (this is the **procedure epilogue**)

Now that we have officially discussed subalgorithms, the very early chapter focusing on the main procedure will make a bit more sense than merely accepting the code as presented on blind faith.

The Program Prologue and Epilogue have responsibility for identifying the bounds of the **code segment** in the assembly language coding.

The Procedure Prologue and Epilogue have responsibility for identifying the bounds of each individual subalgorithm within the program. They do this by initially creating the new activation record for the subalgorithm, and then later dismantling the obsolete activation record when it is no longer necessary.

emit procedure prologue

```
procedure emit_procedure_prologue (proc_name,local_vars)
  # emit kbox assembly code to:
  #   - display the new procedure name as a comment
  #   - set up a new activation stack item
  #     - save return address
  #     - save old frame pointer
  #     - move current stack pointer to new frame pointer

  # remember that the "caller" has the responsibility
  #   - evaluate the actual arguments
  #   - and push appropriate info onto activation stack
  #   - prior to the transfer of control

  emit_blank_line ()
  emit_comment ("BEGIN PROCEDURE: "||proc_name)
  emit_blank_line ()
  display_local_symbol_table ()
  emit_blank_line ()
  emit_label (proc_name)
  emit_line ("push",RP,,,"save the RP to stack")
  emit_line ("push",FP,,,"save the FP to stack")
  emit_line ("mov",FP,SP,,"current SP become the new FP")
  if (\local_vars) then
  {
    emit_line ("movi",OR,local_vars,,
               "space for local variables")
    emit_line ("add",SP,SP,OR)
  }
  emit_blank_line ()
end
```

emit procedure epilogue

```
procedure emit_procedure_epilogue (proc_name)
  # emit kbox assembly code to:
  #   - display the procedure name as a comment
  #   - clean up the obsolete activation stack item
  #     - retrieve old stack pointer from frame pointer
  #     - retrieve old frame pointer
  #     - retrieve return address

  emit_blank_line ()
  emit_label ("EXIT"||proc_name)
  # the following automatically cleans up local variables!
  emit_line ("mov",SP,FP,,
             "retrieve old SP from current FP")
  emit_line ("pop",FP,,,"retrieve FP from stack")
  emit_line ("pop",RP,,,"retrieve RP from stack")
  emit_line ("ret",,,,"return to calling procedure")
  emit_blank_line ()
  emit_comment ("END PROCEDURE: "||proc_name)
  emit_blank_line ()
end
```

Note: The instruction **mov sp,fp** automatically frees local variable storage as a side effect!

17.3.3 Procedure Return

Contrary to the obvious, a **return** statement in the **kize** programming language is **not** translated into the **kcode ret** instruction! In order to properly terminate the subalgorithm, the **return** statement must instead exit through the **procedure epilogue**.

- if the called procedure is a **function**
  then parse the expression and **pop IA** or **pop FA** as appropriate if the called procedure is a **subroutine**
  then ***do nothing!***
- **jmp** ***procedure_exit***

The **kize** return procedure should branch to the procedure epilogue where the actual **kcode ret** instruction will be found!

17.4 Nested Scope

Recall that nesting allows subalgorithms to appear within subalgorithms to create a hierarchical relationship among the components. The concept of nested scope is necessary to understand which variables are available in storage and which variables can actually be accessed.

Definition of Nested Scope: A variable is known and accessible throughout the entirety of its defining block, ***unless*** it is **redeclared** within an interior block.

The difficulty with this definition is **not** its complexity, but its implication. The compiler has to take into consideration not only the local variables within a given procedure but also every local variable found within a containing procedure and ultimately every global variable!

Simple scope only needs to remember the procedure which called it so it can properly return to the CALLER. Hence, simple scope builds a simple linked list of frame pointers that completely describe a backward chain of activation records that ultimately return to the **main procedure**. This simple linked list of frame pointers is commonly referred to as the ***dynamic chain***.

Nested scope not only requires saving a dynamic chain pointer with each procedure call, but also the frame pointer for the procedure frame *in which the CALLED procedure was declared* (or more precisely, the most recent occurrence for a stack frame for the declarer of the CALLED procedure). These frame pointers, called the ***static chain***, provide a backward history of declaration environments going back to the **main procedure**.

Although both chains ultimately go backward through frames on the activation stack, the ***dynamic chain*** returns to the main procedure in the reverse order in which the activation records were created, while the ***static chain*** returns to the main procedure possibly skipping over some activation records. **They need not be the same chains!**

And who figures out the static chain pointer?

the CALLER or the CALLED procedure?

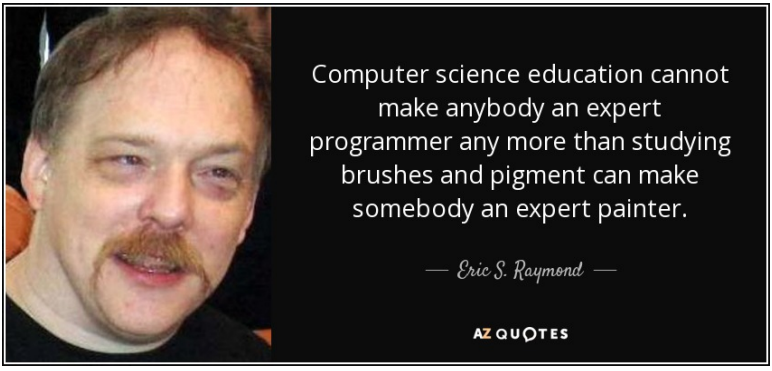
Only the CALLER is capable of determining the static chain pointer. It does so by analyzing the predecessors found on its own static chain.

How does the CALLER remember all these variables that arise using nested scope? By saving the symbol tables for each procedure found on the activation stack on a **stack of symbol tables!**

I think we want to stick with simple scope for the **kize** programming language!

17.5 Phase Five Implementation

- Procedure Signature
  - identifier
  - empty formal argument list (for now)
  - return type
  - implementation of procedure entries and
    - inclusion in the global symbol table
- Activation Stack
  - modifications to Procedure Prologue
  - modifications to Procedure Epilogue
- Function Activation
  - via reference to function in expression
  - return value using register IA or FA
- Procedure Activation
  - via **call** statement
- Function Executable Code
  - evaluation of return value
  - return through procedure epilogue
- Procedure Executable Code
  - return through procedure epilogue


Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter.
— Eric S. Raymond —
AZ QUOTES

Chapter 18

Phase Six: Procedures with Arguments

A World with Arguments

18.1 Procedure Signature Revisited

As previously discussed the **procedure signature** provides a complete description of the procedure's structure: its **name**, its **formal arguments**, and its **return type** (if declaring a function).

This information is obviously necessary in order to properly activate the procedure and provide it with the appropriate information with which to work; this information is also necessary to properly generate the appropriate **kcode** assembly language code especially with regard to manipulating the data transferred via the actual argument list. Hence, with the additional aspect of non-empty formal argument lists, the procedure entries in the **global symbol table**, and the variable entries for formal arguments in the **local symbol table** take on a much more significant role than in the previous chapter.

The **formal argument list** and the corresponding **actual argument list** become the prominent features in this chapter.

18.1.1 Formal Arguments

The **formal argument list** defines both ***what*** and ***how*** information will be transferred from the CALLER to the CALLED procedure. In addition, the **formal argument list** identifies the ***names*** that will be used to access this information from within the CALLED procedure.

Please take note of the following important feature of a procedure signature. The procedure identifier is a *global identifier* and is recognized anywhere within the entire body of source code; the formal argument list identifiers are *local identifiers*, however, and are known only within the body of that specific procedure. It is essential that the procedure entries in the **global symbol table** maintain accurate information regarding the formal argument list in order to correctly transfer information into the procedure and to correctly retrieve that information.

There is also a significant difference between formal argument identifiers and other local variable identifiers as we will learn below.

18.1.2 Actual Arguments

We are now in a position to use the argument lists (both formal and actual) to move data between the various procedures. The formal argument list defines the initial collection of local variables we need to remember within the local symbol table, and the process by which we will transfer the data values.

The actual argument list determines the exact value to be transferred and whether to transfer its R-value (call by value) or its L-value (call by variable). The local symbol table must augment information in the variable entries to distinguish between call by value and call by variable. The activation stack is the vehicle for enabling the data transfer – all data values are pushed onto the stack immediately prior to the CALLER procedure relinquishing control to the CALLED procedure.

Note: As a result, the position for the actual arguments will reside on the activation stack ***above*** the new frame pointer **fp** (i.e., positive offset) for the CALLED procedure! The remaining local variables will reside in their normal position ***below*** the new frame pointer **fp** (i.e., negative offset).

The two most common techniques for transferring data into a procedure are: **call by value** and **call by variable** (also known as **call by reference**).

18.1.3 Call by Value

The simplest and most obvious transfer mechanism is to push the data's R-value (a copy of the actual data) onto the stack. The data is immediately accessible to the CALLED procedure, using the same access methods as any local variable. Clean and simple!

The fly in the ointment is that any changes made to the formal argument variable overwrites its storage location on the stack and is ultimately lost when the CALLED procedure returns to the CALLER procedure.

The original data value at its original memory location remains unchanged by any action performed by the subalgorithm.

This technique is ***uni-directional***. Information comes in, but modifications can never get out!

18.1.4 Call by Variable

A technique which allows ***bi-directional*** communication requires a bit more work. Rather than pushing the data's R-value onto the stack, we instead push the data's L-value (the address where the data may be found) onto the stack. The data is also accessible to the CALLED procedure, but it requires indirect addressing. Rather than finding data immediately on the stack, we have to follow the address on the stack to retrieve the actual data.

We will further discuss call by variable and indirect addressing in each of the remaining chapters – structured data types and pointers. However, it has a significant impact *right now* to the work we have done so far on the **kize** compiler. We have not previously encountered any reason to worry about indirect addressing. All of our variable declarations were either global variables or local variables they have **not** been formal arguments. Local variables that **are** formal arguments have the option ***call by value*** or ***call by variable***. Call by value is the usual access method to data

that we have been using all along; but call by variable is definitely new!

That is why my symbol tables had an extra field for variable entries to distinguish between the two mechanisms. Only at this late date do we realize the purpose for the extra field in the symbol table entry. And definitely pay attention to the following subtle change in accessing the data correctly.

Obviously, **call by value** is the simpler transfer technique. Its implementation is ***mandatory*** in the programming language **kize**.

Call by variable is a bit more challenging and requires some rethinking and rewriting of supporting routines previously developed solely for direct addressing or offset addressing from the frame pointer. However, its implementation is also ***mandatory*** in the programming language **kize**. It is a concept which is absolutely necessary in the field of computer science

Remember we are focusing solely on formal arguments and actual arguments. These are represented as local variables which are accessible via offset from the frame pointer **fp**.

Call by Value

ldr OR,*var_offset*
add SAR,FP,OR
ldr *reg*,[SAR]

Call by Variable

ldr OR,*var_offset*
add SAR,FP,OR
ldr SAR,[SAR] **note: indirect addressing**
ldr *reg*,[SAR]

18.2 Role of the CALLER (redux)

The role of the CALLER procedure is essentially fourfold:

- prior to the actual CALL statement,
  set up the actual arguments on the stack
  which in this chapter means ***do something!***

  do something *means push information onto the stack in the proper order to match the formal argument list*
  make sure data types are consistent with formal arguments
  push data (call by value) or push address (call by variable)

- issue the **kcode** call statement:
  call *procedure_label*

- if the called procedure is a **function**
  then store the return value (**IA** or **FA**) on the stack
  if the called procedure is a **subroutine**
  then ***do nothing!***

- clean up the actual arguments from the stack
  which in this chapter means ***do something!***

  do something *means clean up the stack*
  add sp,sp,#actual_arguments

Note: At this point in time all the actual arguments are either atomic data types or addresses (for arguments using call by variable). Hence, the total size for information pushed onto the stack prior to a procedure call is the actual number of formal arguments (one klunk for each argument).

18.3 Role of the CALLED (redux)

The role of the CALLED procedure is essentially fivefold:

- set up a new stack frame for the procedure
  save the **rp**, save the old **fp**, set the new **fp** to the current **sp**
  (this is the **procedure prologue**)
- set up storage and offsets for local variables
  using the information found in the local symbol table
  note that local variables are stored below the **fp** (negative offset)
- execute the procedure assembly language coding
- if the called procedure is a **function**
  then move the return value to **IA** or **FA**
  if the called procedure is a **subroutine**
  then ***do nothing!***
- clean up the now obsolete stack frame for the procedure
  set the **sp** to the current **fp**, retrieve the old **fp**, retrieve the **rp**
  execute the actual **ret** instruction
  (this is the **procedure epilogue**)

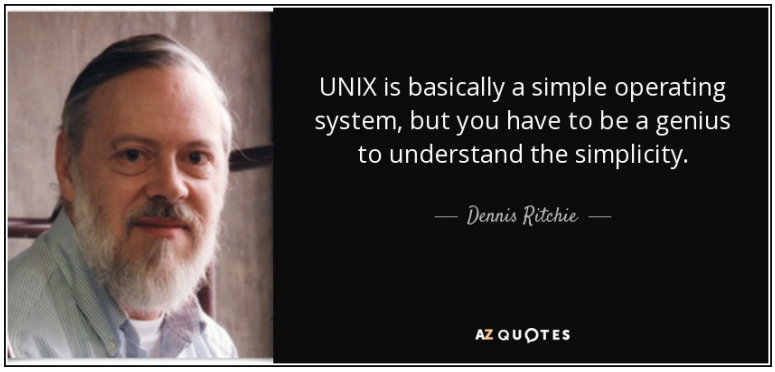
18.4 Other Issues to Consider

In addition to the above specific topics regarding procedures with arguments, the following tangential issues come into play and may require some adjustments to previously developed coding:

- procedure names are global identifiers and must be properly maintained in the global procedure table
  - I chose to incorporate this information into the global symbol table at the time of **procedure declaration**
- formal argument lists are local identifiers and must ***at some point*** be inserted into the local symbol table for that particular procedure
  - I chose to incorporate this information into the local symbol table at the time of **procedure definition**
  - but it does require retrieving formal argument information from the global symbol table
- formal argument lists require quite a bit of book keeping in and of themselves
  - call by value or call by variable (direct or indirect addressing)
  - identifier
  - data type

18.5 Phase Six Implementation

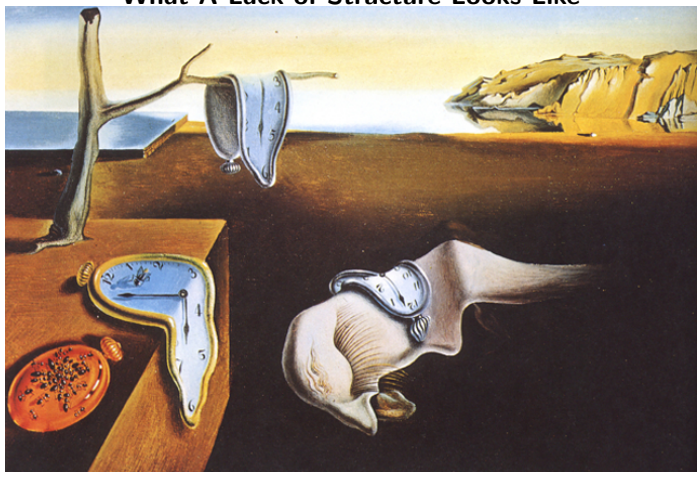
- Implementation of **formal** argument lists
- Implementation of **actual** argument lists
  - Implementation of **call by value**
  - Implementation of **call by variable**
- Appropriate modifications to:
  - CALLER procedure executable code
  - CALLED procedure executable code


UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.
— Dennis Ritchie —
AZ QUOTES

Chapter 19

Phase Seven: Structured Data

What A Lack of Structure Looks Like



Simple data or **atomic data** are the basic building blocks for any programming language. In addition to these atomic types, programming languages provide some capability to build **structured types** (or **aggregate types**) which group items together as a collective unit.

The two most common such structured types are ARRAYs and RECORDs.

ARRAYs have their origin in the mathematical roots of the programming language FORTRAN; RECORDs have their origin in the business-oriented applications supported by the programming language COBOL. Once these two aggregate structures came together in the grammar for the programming language ALGOL, they have become almost universally available in every subsequent programming language!

Records (or their C synonym **struct**s) provide a foundation for the subsequent definition of classes found in more modern object-oriented programming languages.

Just like logical structures may be nested to provide very powerful control structures within a program, structured types may also be nested within one another to create very intricate data storage collections.

Both arrays and records utilize a very simple syntax to retrieve individual data elements from these aggregate collections.

This chapter summarizes the fundamental characteristics for these two structured data types. Please note that type declarations in the **kize** programming language are strictly limited to structured data types. They are not to be used to create synonyms or aliases for atomic data types.

The following page highlights the grammar elements that will be the focus of this chapter. It is essential that the symbol tables be updated a enhanced at this point to accommodate the new concept of user-defined data types.

kize grammar (phase seven)

```
type_declarations       TYPES type_list
type_declarations

type_list               IDENTIFIER : declaration_type ;
                        more_type_list

more_type_list          type_list
more_type_list

declaration_type        array_type
declaration_type        record_type

array_type              ARRAY [ ub ] OF declared_type

ub                      NUMBER
ub                      IDENTIFIER

record_type             RECORD fld_list END

fld_list                IDENTIFIER : declared_type ;
                        more_fld_list

more_fld_list           fld_list
more_fld_list

declared_type           atomic_type
declared_type           IDENTIFIER
declared_type           ^ declared_type

atomic_type             INT
atomic_type             REAL
atomic_type             STRING

qualifier               structure_qualifier

structure_qualifier     array_qualifier
structure_qualifier     record_qualifier
structure_qualifier

array_qualifier         [ expression ]
                        structure_qualifier

record_qualifier        . IDENTIFIER
                        structure_qualifier
```

19.1 Arrays

Definition 19.1.1. An **array** is a linearly ordered data structure comprised of ***homogeneous*** data items which are accessible by referencing its **index** (position) within the data. The index is limited to integer values that are greater than or equal to a specified **lower bound** and are less than or equal to a specified **upper bound**.

In the **kize** programming language, the lower bound for an array will always be the integer 0; the upper bound will always be the specified SIZE for the array minus 1. SIZE must be a constant value – either a literal constant (a *mandatory* featurein your implementation) or a named constant (an *optional* feature in your implementation).

In the **kize** programming language, the data type for an array must either be an atomic type or a *previously declared* structured type.

TYPE declarations

```
TYPES
        ARRAY10       :   ARRAY [10] OF INT;
        ARRAY20X10    :   ARRAY [20] OF ARRAY10;
```

VARIABLE declarations

```
VARIABLES
            A   :   ARRAY10;
            B   :   ARRAY20X10;
```

array references within executable code

```
SUM := 0;
FOR I := 0 TO 9
    SUM := SUM + A[I];
WRITELN (SUM);

FOR I := 0 TO 9
DO
    FOR J := 0 TO 9
        WRITE (B[I][J]," ");
    WRITELN ();
END;
```

As illustrated above, **multidimensional arrays** must be declared one level at a time. Furthermore, references to data within a multidimensional array takes the form B[I][J] rather than the more compact shorthand B [I,J].

You also have the *option* to implement this shorthand form within your **kize** compiler.

19.1.1 array storage

Array storage is a fairly straight-forward topic with few surprises. Suppose an array A has lower bound **lb**, upper bound **ub**, and data size for each element requiring **b** bytes. Then

- total number of elements = **ub - lb + 1 = size**
- total storage = **size * b** bytes
- base address for storage = $\mathbf{a_0}$

As an aid in visualizing storage in memory, I would suggest viewing memory as a sequence of memory locations with smaller address values being toward the bottom and larger address values being toward the top. The base address $\mathbf{a_0}$ is the ***lowest address*** reserved for storage of the array. Elements will be stored in increasing order:

- A[0] will be at address $\mathbf{a_0}$
- A[1] will be at address $\mathbf{a_0 + b}$
- A[2] will be at address $\mathbf{a_0 + 2 * b}$
- etc $\cdots$

Many of you may be yawning right now! But I caution you to remember that global variables are stored in main memory where the addresses **increase** as memory is reserved or allocated while local variables are stored on the stack where the addresses **decrease** as memory is reserved or allocated.

What is important is not whether we fill an array bottom-up or top-down but rather that we are **consistent in storage and retrieval** regardless of where the information is actually stored.

19.1.2 array L-values

Determining L-values for an array is equally straight-forward.

- the L-value for the collective array A
  would be $\mathbf{a_0}$ (the base address)
- the L-value for a single array element A[i]
  would be $\mathbf{a_0} + (\mathbf{i} - \mathbf{lb}) * \mathbf{b}$

Some of you may be asking about weird nested structure types that mix arrays and records. How does one handle these complicated combinations? My answer would be to process these combinations *one at a time*.

What is the structure – an array or a record? Which item do I want – an array element or a record field descriptor? Now, what am I currently looking at – an atomic type or another structure? If it is atomic, then I am done; if not, then start the process anew with the aid of recursion!

Remember, recursion is your friend. Let recursive descent parsing simplify your compiler coding and do the heavy lifting for you.

19.2 Records

Definition 19.2.1. A **record** is a linearly ordered data structure comprised of ***heterogeneous*** data items which are accessible by referencing its **field descriptor** (name) within the data. The field descriptors and their respective types must be clearly defined in the record declaration.

TYPE declarations

```
TYPES
        CARTESIANTYPE   :   RECORD
                            X : REAL;
                            Y : REAL;
                            END;
        POLARTYPE       :   RECORD
                            RADIUS : REAL;
                            THETA : REAL;
                            END;
```

VARIABLE declarations

```
VARIABLES
            CARTESIAN   :   CARTESIANTYPE;
            POLAR       :   POLARTYPE;
```

record references within executable code

```
READLN (CARTESIAN.X,CARTESIAN.Y);
WRITELN (POLAR.RADIUS," ",POLAR.THETA);
```

19.2.1 record storage

Record storage is once again a fairly straight-forward topic with few surprises. Suppose a record R has field descriptors

$$\mathbf{FD_1}, \mathbf{FD_2}, \cdots, \mathbf{FD_n},$$

with data size for each field descriptor requiring

$$\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n}$$

bytes. Then

- total storage = $\Sigma_1^n \mathbf{b_i}$
- offset ($\mathbf{FD_i}$ is defined inductively:
  offset ($\mathbf{FD_1}$) = 0
  offset ($\mathbf{FD_i}$) = offset ($\mathbf{FD_{i-1}}$) + $\mathbf{b_{i-1}}$
- base address for storage = $\mathbf{a_0}$

As an aid in visualizing storage in memory, I would once again suggest viewing memory as a sequence of memory locations with smaller address values being toward the bottom and larger address values being toward the top. The base address $\mathbf{a_0}$ is the ***lowest address*** reserved for storage of the record. Fields will be stored in increasing order:

- R.FD$_1$ will be at address $\mathbf{a_0}$
- R.FD$_2$ will be at address $\mathbf{a_0} + \mathbf{b_1}$
- R.FD$_3$ will be at address $\mathbf{a_0} + \mathbf{b_1} + \mathbf{b_2}$
- etc $\cdots$

Many of you may be experiencing ***deja vu*** all over again! But I caution you once again to remember that global variables are stored in main memory where the addresses **increase** as memory is reserved or allocated while local variables are stored on the stack where the addresses **decrease** as memory is reserved or allocated.

What is important is not whether we fill a record bottom-up or top-down but rather that we are **consistent in storage and retrieval** regardless of where the information is actually stored.

19.2.2 record L-values

Determining L-values for a record is equally straight-forward.

- the L-value for the collective record R
  would be $\mathbf{a_0}$ (the base address)
- the L-value for a single record field descriptor R.FD$_i$
  would be $\mathbf{a_0}$ + offset ($\mathbf{FD_i}$)

Remember, when mixing arrays and records these complicated combination should be processed *one at a time*.

What is the structure – an array or a record? Which item do I want – an array element or a record field descriptor? Now, what am I currently looking at – an atomic type or another structure? If it is atomic, then I am done; if not, then start the process anew with the aid of recursion!

19.3 Implementation: Assignment

Obviously, creating expressions involving structured data types requires the definition and implementation of all operations you intend to allow. That is why I do **not** allow any operations with structures in the **kize** programming language!

The sole exception to this rule is the simple assignment statement

$$A := B$$

i.e., transfer data from one structure to another.

It should be obvious that this is a ***no-brainer***, right? Well, not quite. When are two structured data types considered compatible? This question requires some subtle distinctions. We present two basic definitions:

- **structural equivalence** for two variables
  the definitions of the structured data types are identical
  the number of elements and the base type are identical **(or)**
  the order and the type of field descriptors are identical
- **name equivalence** for two variables
  the names defining the structured data types are identical

Structural equivalence requires recursive processing of the definition of the structured type and its components. As a result, it is potentially time-consuming.

Name equivalence simply requires looking only at the name for the data type. Name equivalence is much simpler to implement and is the type equivalence that **kize** will use.

Once, we have checked for name equivalence, we still are not completely free of complications! If our programming language includes pointers and dynamic allocation of memory (to be discussed next chapter), we still have another issue to consider.

- **shallow copy**
  all data values are copied, as is, from the source to the destination
  i.e., pointer addresses are duplicated in the destination
- **deep copy**
  pointer addresses require construction of a duplicate having its own set of addresses dynamic structure

Shallow copy is obviously much simpler to implement, but it creates multiple pointer references to locations in memory. Multiple pointer references create the potential for dynamic memory errors: losing track of *valid addresses* (**memory leakage**) or referencing *invalid addresses* (**segmentation fault**).

Deep copy may reduce the possibility for such problems, but once again it does so at the cost of time.

We conclude our brief discussion about assignment statements by noting that there is **no difference** between shallow copy and deep copy, **unless** the programming language under consideration supports dynamic allocation of memory and dynamic data structures!

Hence, implementation of simple assignment for structured data types in **klump** becomes a two-step process:

- check for name equivalence
- copy the entire block of data from B to A
  one chunk at a time

emit assignment (structured type)

```
procedure emit_assign (target_type,expr_type,data_size)
  # emit kbox code to assign the data value
  # found at the top of the stack to the target location
  #   also found at the top of the stack
  # target_type specifies the data type for the transfer

  emit_line ("nop",,,,"assignment")
  if (isAtomic(global_symbol_table[target_type])) then
    if (target_type == "INT") then
    {

    }
    else if (target_type == "REAL") then
    {

    }
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
  else if (isArray(global_symbol_table[target_type]) |
           isRecord(global_symbol_table[target_type])) then
    if (target_type == expr_type) then
    {
      label_loop := @name_generator
      emit_line ("pop",SAR)
      emit_line ("pop",DAR)
      emit_line ("movi",IR,data_size)
      emit_label (label_loop)
      emit_line ("ldr",IA,SAR)
      emit_line ("str",IA,DAR)
      emit_line ("inc",SAR)
      emit_line ("inc",DAR)
      emit_line ("dec",IR)
      emit_line ("cmp",IR,ZR)
      emit_line ("jgt",label_loop)
    }
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
  else
    stop ("how on earth did you get here!")
end
```

19.4 Implementation: Procedures

The two obvious considerations with any procedure are: data transfer and the optional return value.

With respect to **data transfer**, the first question would have to once again consider the compatibility of types. Name equivalence, which we found helpful when discussing the assignment statement, also applies here. We need check only that the data types have the same named type. Recall that we have two techniques for data transfer – ***call by value*** and ***call by variables***. The former must copy the data value (R-value) onto the stack; the second must copy the data address (L-value) onto the stack.

Call by value, although relatively simple to implement, requires additional time and memory space to move an entire structure onto the stack for data transfer. For this reason, ***call by variable*** is **mandatory** for structured data in the **kize** programming language! It is important to note that any structured data type is therefore subject to side effects – changes made to a local variable propagating back to the actual parameter in the CALLER procedure.

With respect to the optional **return value**, do we want to allow structured types to be returned from a function as well as atomic types? For **kize** I chose **not** to allow this additional flexibility. The reasoning behind my decision is as follows:

- returning a structured type requires **more** than one register, **IA** or **FA**
- to transfer such data would require
  addition communication between CALLER and CALLED
- if pointers and dynamic allocation are implemented in the language,
  then the issue of **shallow copy/deep copy**
  would arise once again:
  what happens when you return an address
  to a location within the discarded activation record?
- the programmer can accomplish the same end result by simply using ***call by variable*** with a variable in formal argument list!

19.5 Other Issues to Consider

In addition to the above specific topics regarding structured data types, the following tangential issues come into play and may require some adjustments to previously developed coding:

- the global type table may need some modifications to accommodate structured types
- array information is very simple: only the underlying element data type and size are required
- record information is more complicated: **for each field**, its field identifier, its data type, and ultimately its offset
- however, the revenge of the structured data types comes when calculating L-values for individual elements ($\mathbf{a_0} + (\mathbf{i} - \mathbf{lb}) * \mathbf{b}$ or $\mathbf{a_0} + \mathbf{offset}(\mathbf{FD_i})$)
  - it is a simple enough linear function
  - it is not a problem **until** we realize
    it must be evaluated at ***run time***
    and **not** at ***compile time***
- I chose to use the source address register (SAR) as the foundation for building the correct L-values for variables
  - array references use the data size and index value (IR) to calculate the necessary offset (OR)
  - record references get the offset (OR) directly from the record information
  - a new source address is calculated: SAR $\longleftarrow$ SAR + OR
- to safeguard the base address found in the source address register (SAR) when calculating offsets (especially for arrays), I chose to save the SAR on the stack until the offset has been completely determined.

L-value calculation (array)

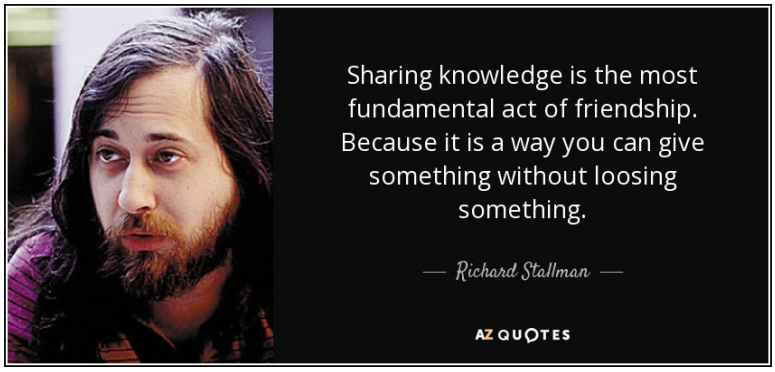
```
procedure calc_array_offset (element_size)
  emit_line ("nop",,,,"array offset")
  retrieve_register (IR)
  emit_line ("movi",OR,element_size)
  emit_line ("mul",OR,IR,OR)
  retrieve_register (SAR)
  emit_line ("add",SAR,SAR,OR)
end
```

L-value calculation (record)

```
procedure calc_record_offset (offset)
  emit_line ("nop",,,,"record offset")
  emit_line ("movi",OR,offset)
  retrieve_register (SAR)
  emit_line ("add",SAR,SAR,OR)
end
```

19.6 Phase Seven Implementation

- Implementation of **either** arrays **or** records
- Implementation of **both** arrays **and** records (*optional*)
- Implementation of nesting for **all** your structured data types
- **call by variable** to transfer structured data
- Implementation of the following two items: (*optional*)
  named constants used to define array size,
  e.g., ARRAY [SIZE] OF INT
  shorthand notation in a multi-dimensional array
  e.g., A[i,j]


Sharing knowledge is the most fundamental act of friendship. Because it is a way you can give something without loosing something.
— Richard Stallman —
AZ QUOTES

Chapter 20

Phase Eight: Pointers

Memory Leakage Will be the Death of Us All!

20.1 Addresses as Data

Until now we have made a clear distinction between L-values and R-values, between addresses for information and the actual data contained at that address. Our first variation from this distinction was when we encountered ***call by value*** and ***call by variable***. What exactly are we transferring when we transfer data into a procedure – its value? or its address? or is there really a difference? Aren't the two just different sides of the **same coin**?

Note: Ray Klump said I wouldn't be able to sneak a numismatist reference into this book. ***I think I just won!***

And if an address can be the data transferred into a procedure, then can it not also be considered as data in its own right ... and even have its own type designation!

PL/I has a catch-all designation for a data type POINTER. Any data type could potentially be found stored at the specified address. The programmer had to keep everything straight in writing his or her code.

C and **C++** require the programmer to specify what data type will be found at a given address. And so does the **kize** programming language.

For any data type which is found in the global type table, there is an implicit ***pointer data type*** which may reference a data item of the specified ***data type***.

sample variable declarations

```
TYPES

    INTARRAY : ARRAY [ 10 ] OF INT;

VARIABLES

    X : INT;
    Y : REAL;

    PX : ∧INT;
    PY : ∧REAL;

    A : INTARRAY;
    PA : ∧INTARRAY;
```

I chose **not** to double the size of the global type table by creating separate entries for pointers. The simple pre-pended caret (∧) identifies a pointer address and its underlying data type immediately follows that single character.

Also note that **all** pointer types, regardless of the underlying data type (atomic or structured) have the same storage requirements – precisely 64 bits. This is the address size (unsigned integer) in **kcode**. This is another reason why there is no need to create entirely new entries in the global type table.

20.2 Three Fundamental Operators

There are three fundamental operators that are appropriate in working with pointers (addresses). We definitely have familiarity with all three from our previous discussions.

The first fundamental operator is **assignment**. How else can we save addresses and move addresses around between variables? However, some care must be taken here. Just as we saw with STRING data type, pointers should not be involved in any complicated expressions. The only permissible operations available with pointers are two comparisons: **equal** and **not equal**.

Manipulating pointers require the implementation of two additional operators that we have previously discussed but not been formally introduced to!

Very early on we considered the distinction between the **L-value** and the **R-value** for a variable. Here we define two operators that parallel those definitions. The operator & is shorthand for ***the address of***, that is, the good old-fashioned L-value. The operator $\wedge$ is shorthand for ***the data at the end of this pointer***, that is, the good old-fashioned R-value.

emit assignment (pointer)

```
procedure emit_assign (target_type,expr_type,data_size)
  # emit kbox code to assign data value at top of stack
  #   to target location also found attop of stack
  # target_type specifies the data type for the transfer

  emit_line ("nop",,,,"assignment")
  if (isAtomic(global_symbol_table[target_type])) then
    if (target_type == "INT") then
    {

    }
    else if (target_type == "REAL") then
    {

    }
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
  else if (isArray(global_symbol_table[target_type]) |
           isRecord(global_symbol_table[target_type])) then
```

```
  {

  }
  else if (isPointer(target_type)) then
    if ((target_type == expr_type) |
        (expr_type == "POINTER")) then
    {
      emit_line ("pop",IA)
      emit_line ("pop",DAR)
      emit_line ("str",IA,DAR)
    }
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
  else
    stop ("how on earth did you get here!")
end
```

examples of the fundamental operators

PX := &X ;

X := PX∧ ;

Note: & is a **prefix** operator – it appears ***before*** the operand; ∧ is a **postfix** operator – it appears ***after*** the operand. Furthermore, in theory, the two are related to one another as follows:

if (PX = PY)
    then PX∧ = PY∧ should also be true!

But the converse is not necessarily true!

if (PX∧ = PY∧)
    then PX = PY need not be true!

The two additional operators are pretty straightforward to implement – we have been using L-values and R-values all along. But the real power inherent in pointers is not so much in the basic concepts and implementation. It comes from the notion that a programmer might request and return memory at will while a program is actually executing.

the null pointer

To provide some minimal level of assistance with pointers, most programming languages provide a single special pointer (typically called **null** or **null pointer**) to initialize pointer variables. Since pointers (addresses) are simply unsigned integers, the data value 0 is a commonly used definition for **null**. I suggest you use that definition for the **kize** programming language as well.

To test for the **null** pointer, we need to allow for two of the six comparison operators (= and <>) to include pointer data types as well. The remaining four comparison operators (>, >=, <, and <=) need not be defined within the context of pointers.

That also applies to ***pointer arithmetic***. Although many programming languages allows such calculations, this is one area where I chose to specifically rein in the **kize** programming language.

20.3 Dynamic Memory Management

We are now ready to discuss a new concept in data storage – one that extends the programmer's options beyond static storage and automatic storage.

Static memory is always fixed in its size and fixed in its location. Why do you think they call it static???

Automatic memory may vary in location as the activation stack grows (with CALL statements) and shrinks (with RETURN statements), but its size and its locations are always a fixed offset from whatever is the current value for the frame pointer which varies over time during program execution.

Dynamic memory is totally at the discretion of the programmer! It comes into existence at the program's request (**allocation**); and it goes out of existence at the program's release (**deallocation**);

During its existence, it is always available $\cdots$ *provided* the program knows where to find it! Vital information must be saved in pointer variables. Incorrect information can wreak havoc:

- if you go to an invalid address, then you experience the wonderful world of ***segmentation faults***
- if you inadvertently lose track of an address, then you experience ***memory leakage***

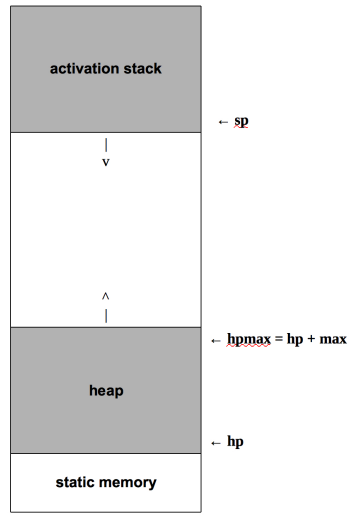
So now we confront the issue of implementing dynamic memory management. We present the basic concepts and some of the underlying principles for this topic. But our discussion is far from compete.

20.3.1 The HEAP

A picture is worth a thousand words, so I am going to show you a lot of pictures in the next few sections!

In compiler construction, a **heap** is a designated area of memory set aside specifically for the implementation of dynamic memory management. The heap typically resides immediately above the static memory reserved for the program. Four our discussion, our heap will have a fixed size (called MAX).

Memory organization for **kize** programming language now looks like:



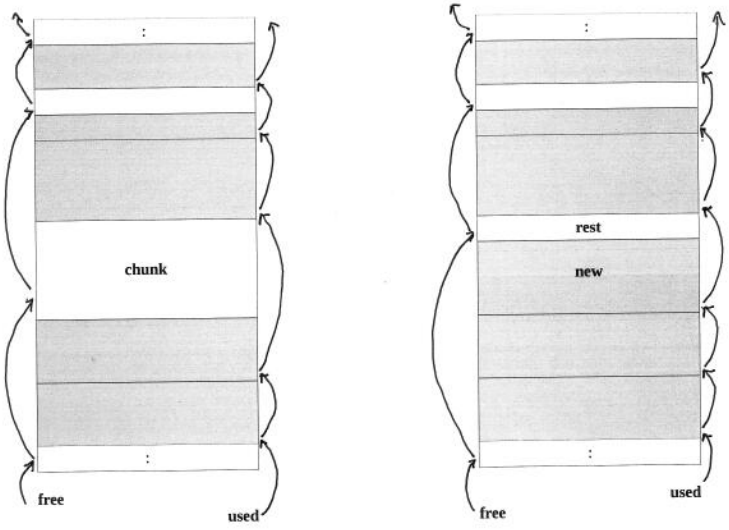
The heap initially is one big chunk of memory which is available to the program, either to request for use or to release back to the heap. Over time our heap's memory will come to resemble *swiss cheese* with used areas and free areas alternating with one another.

To properly maintain this space a decent heap manager must be implemented – requiring two important features: a **free list** of available chunks and a **used list** of chunks that are currently being used.

20.3.2 Allocation

In order to process a request for memory (allocation), the heap manager must perform the following steps:

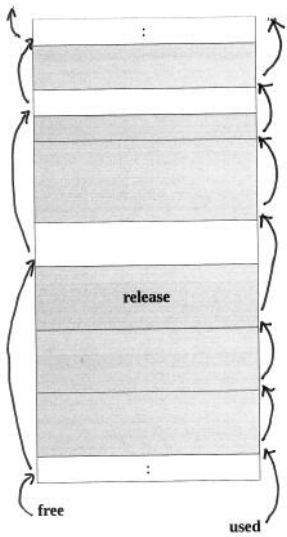
- search the **free list** looking for a chunk of memory large enough to satisfy the request
- split the selected chunk into two pieces:
  allocated portion
  the rest (anything leftover?)
- add the allocated portion to the **used list**
- add the rest to the **free list**
  replacing the selected chunk of memory

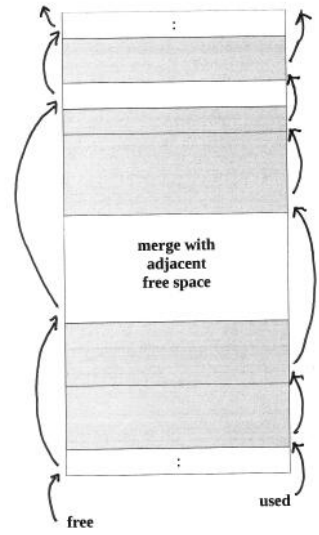


20.3.3 Deallocation

In order to process a release of memory (deallocation), the heap manager must perform the following steps:

- determine the size of the released chunk
- goto that chunk in the **used list**
- move the chunk:
  from the **used list**
  to the **free list**
- *if necessary* merge (coalesce) adjacent chunks
  in the **free list**





20.4 KIZE Dynamic Memory Management

Dynamic memory management is a very important topic and it deserves a thorough presentation. It has implications in several areas of computing – not just in compiler construction but also in operating systems and file management. Selecting the ***best*** chunk of memory available is not as simple as it might initially appear!

There are many competing ideas regarding what is ***best***.

- **first-fit**: this is probably the simplest to understand
  grab the *first* chunk in the free list that works!
- **best-fit**: search the entire free list
  grab the *smallest* chunk that works!
- **worst-fit**: search the entire free list
  grab the *biggest* chunk that works!

All three techniques have their unique advantages and disadvantages; each one can out perform the others in certain situations. And all require time and thought to implement.

The dynamic memory management I propose for the **kize** programming language is the **worst** – not worst-fit, just the worst ... the pits ... a disgrace ... ! It should more properly be called ***dynamic memory mismanagement***.

This simplest description I can think of for its operation is how a two-year old would give out paper plates at a family picnic:

- the kid starts out with a small stack of paper plates
- every time he gets a request, he gives out a plate
- every time he gets a dirty plate returned, he throws it on the ground
- and so on, and so on, ...
- until he gets a request and no plates remain
  (except for those strewn all across the picnic area)
  so he throws a tantrum and storms off

20.4.1 NEW directive

The **new** directive should be implemented as follows:

- the **new** directive requests storage for **b** bytes
- the **IB** register contains the block size
- the MALLOC instruction provides the base address for the new memory block in the **IA** register
- the address found in the **IA** register should be pushed onto the stack

new

```
procedure NewRequest ()
# new_request --> NEW defined_type

   must_match (current,"NEW") &
     current := read_lexeme ()
   new_type := DeclaredType ()
   new_size := global_symbol_table[new_type].type_size
   emit_line ("movi",IB,new_size,,"new request")
   emit_line ("malloc",IA,IB)
   emit_line ("push",IA)
   return "^"||new_type
end
```

The containing pointer **assignment** statement should now be completed!

20.4.2 DISPOSE directive

The **dispose** directive would be implemented as follows:

- the **dispose** directive returns a block of storage to the heap
- the **SAR** register contains the base address
- the **IA** registers contains the size **b** of the block
- the DALLOC instruction would normally perform the heavy lifting for this task!

dispose

```
procedure DisposeRequest ()
# dispose_request --> DISPOSE named_item

  must_match (current,"DISPOSE") &
    current := read_lexeme ()
  named_type := NamedItem ()
  if (type(named_type) == "variable_location") then
  {
    dispose_type := named_type.data_type
    dispose_size := 0
    emit_line ("mov",SAR,IA,,"dispose request")
    emit_line ("dalloc",IA,IB)
  }
  else
    SemanticsError (current.line_no,
      "invalid item to get address of (" ||
      type(named__type) || ")")
end
```

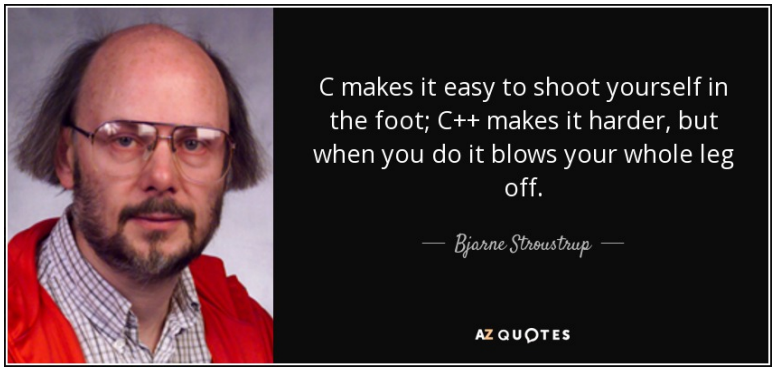
But it is important to recall that way back in the first part of this text when we were discussing assembly language, that we chose not to fully implement dynamic memory management. The memory managements technique we defined there was the **worst**!

The **kize** heap manager gives out the next available address within the heap area until it runs out of addresses. This really is not much of a memory management system! But it actually works for small scale projects. I promise you will feel a sense of accomplishment when you successfully compile a **kize** program building a *singly-linked list* or a *doubly-linked list*.

If anyone really feels cheated by not implementing a real heap manager, you can ... add it ... later ... on your own ...

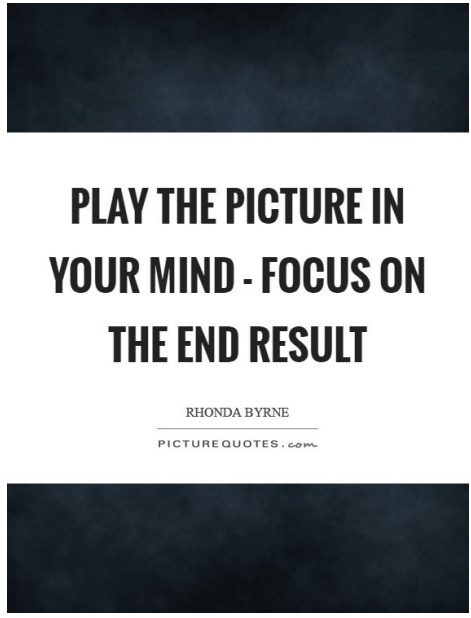
20.5 Phase Eight Implementation

- Implementation of **pointer data types** (*optional*)
  - including **assignment** statement assigning pointer values
  - including two **specialized operators**: & and ∧
- Implementation of **dynamic memory management** (*optional*)
  - define **heap** memory area
  - including implementation of **new**
  - including implementation of **dispose**


C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.
— Bjarne Stroustrup —
AZ QUOTES

Chapter 21

The End Result



21.1 kizecompiler: the main program

kizecompiler.icn

```
# ---------------------------------------------------------- #

# kizecompiler.icn
#
# this program is a recursive descent compiler
# for the programming language kize
#
# input file will be an ASCII text file of triplets
#     - line number within the original input file
#     - token type  for a given lexeme
#     - token value for a given lexeme
#
# to generate the executable code:
#     $ icont kizecompiler lexeme error tables assembler
#
# to execute the code, use IO redirection and/or piping:
# for a previously scanned ac source file
#     $ ./kizecompiler < scanned_file > output_file
# for an unscanned ac source file
#     $ ./kizescanner < input_file | \
#         ./kizecompiler > output_file

# ---------------------------------------------------------- #

global          control_stack
global          current

global          ATOMIC
global          LITERALS
global          COMPOPS
global          ADDOPS
global          MULOPS
global          QUALIFIERS

# ---------------------------------------------------------- #

procedure main ()
  ATOMIC        := set ([ "INT", "REAL" ])
  LITERALS      := set ([ "NUMBER", "DECIMAL",
                          "ASTRING", "NULL" ])
  COMPOPS       := set ([ "=", "<>", ">", "<",
                          ">=", "<=" ])
  ADDOPS        := set ([ "+", "-", "OR" ])
  MULOPS        := set ([ "*", "/", "%", "AND" ])
  QUALIFIERS    := set ([ "[", ".", "^" ])

  initialize_assembler ()
  control_stack := []
  current := read_lexeme ()
```

```
   KizeProgram ()
   return
end

# ------------------------------------------------------------ #

procedure KizeProgram ()
# kize_program --> PROGRAM IDENTIFIER
#                  !save_ident global_declarations
#                  BEGIN !emit_program_prologue
#                  procedure_list
#                  END !emit_program epilogue

   must_match (current,"PROGRAM") &
     current := read_lexeme ()
   must_match (current,"IDENTIFIER")
   # !save_ident
   program_name := current.tok_value
   current := read_lexeme ()
   GlobalDeclarations ()

   must_match (current,"BEGIN") &
     current := read_lexeme ()
   # !emit_program_prologue
   emit_program_prologue (program_name)

   ProcedureList ()

   must_match (current,"END") &
     current := read_lexeme ()
   # !emit_program_epilogue
   emit_program_epilogue (program_name)
end

# ------------------------------------------------------------ #

procedure Procedure ()
# procedure --> PROCEDURE IDENTIFIER !save_ident
#                  !check_declared
#                  local_declarations
#                  BEGIN !emit_procedure_prologue
#                  statement_list
#                  END !emit_procedure_epilogue

   must_match (current,"PROCEDURE") &
     current := read_lexeme ()
   must_match (current,"IDENTIFIER")
   # !save_ident
   procedure_name := current.tok_value
   current := read_lexeme ()
   # !check_declared
   if (not (isProcedure(isGlobal(procedure_name)))) then
     SemanticsError (current.line_no,
```

```
      "undeclared procedure (" || procedure_name || ")")
  formal_argument_list :=
    global_symbol_table[procedure_name].formal_arguments
  local_storage :=
    LocalDeclarations (formal_argument_list)
  must_match (current,"BEGIN") &
    current := read_lexeme ()
  # !emit_procedure_prologue
  emit_procedure_prologue (procedure_name,local_storage)
  StatementList ()
  must_match (current,"END") &
    current := read_lexeme ()
  # !emit_procedure_epilogue
  emit_procedure_epilogue (procedure_name)
end

$include "declarations.icn"
$include "executables.icn"
```

21.1.1 declarations: global and local

declarations.icn

```
# ---------------------------------------------------------- #

procedure GlobalDeclarations ()
# global_declarations --> !initialize_global_symbol_table
#                         const_declarations
#                         type_declarations
#                         var_declarations
#                         proc_declarations

   initialize_global_symbol_table ()

   ConstDeclarations ()
   TypeDeclarations ()
   VarDeclarations ()
   ProcDeclarations ()
end

procedure LocalDeclarations (formal_argument_list)
# local_declarations --> !initialize_local_symbol_table
#                        label_declarations
#                        var_declarations
#                        !save_local_storage

   initialize_local_symbol_table (formal_argument_list)

   LabelDeclarations ()
   local_storage := VarDeclarations (0)
   # !save_local_storage
   return local_storage
end

# ---------------------------------------------------------- #

procedure ConstDeclarations ()
# const_declarations --> CONSTANTS const_list
# const_declarations -->

   if (does_match (current,"CONSTANTS")) then
   {
     current := read_lexeme ()
     ConstList ()
   }
   else
     return
end

procedure ConstList ()
# const_list --> IDENTIFIER !check_duplicate :
#                literal ; !insert_const_entry
```

```
#                    more_const_list

  must_match (current,"IDENTIFIER")
  ident := current.tok_value
  # !check_duplicate
  if (isGlobal(ident)) then
    SemanticsError (current.line_no,
      "duplicate_global_identifier_("||ident||")")
  current := read_lexeme ()
  must_match (current,":") &
    current := read_lexeme ()
  literal := Literal ()
  must_match (current,";") &
    current := read_lexeme ()
  # !insert_const_entry
  constant_type  := literal[1]
  constant_value := literal[2]
  if (constant_type == "POINTER") then
    SemanticsError (current.line_no,
      "creating_an_alias_for_NULL_POINTER_("||ident||")")
  global_symbol_table[ident] :=
    constant_entry (ident,constant_type,constant_value)
  MoreConstList ()
end

procedure MoreConstList ()
# more_const_list --> const_list
# more_const_list -->

  if (does_match (current,"IDENTIFIER")) then
    ConstList ()
  else
    return
end

procedure Literal ()
# literal --> NUMBER
# literal --> DECIMAL
# literal --> ASTRING
# literal --> NULL

  if (member (LITERALS,current.tok_type)) then
  {
    if (current.tok_type == "NUMBER") then
      literal_type := "INT"
    else if (current.tok_type == "DECIMAL") then
      literal_type := "REAL"
    else if (current.tok_type == "ASTRING") then
      literal_type := "STRING"
    else # (current.tok_type == "NULL")
      literal_type := "POINTER"
    if (literal_type == "POINTER") then
      literal_value := 0
```

```
    else
      literal_value := current.tok_value
    current := read_lexeme ()
    return [literal_type,literal_value]
  }
  else
    SyntaxError (current.line_no,
                 "CONSTANT",
                 current.tok_type)
end

# ------------------------------------------------------------ #

procedure TypeDeclarations ()
# type_declarations --> TYPES type_list
# type_declarations -->

  if (does_match (current,"TYPES")) then
  {
    current := read_lexeme ()
    TypeList ()
  }
  else return
end

procedure TypeList ()
# type_list --> IDENTIFIER !check_duplicate
#               : declaration_type ; !insert_type_entry
#               more_type_list

  must_match (current,"IDENTIFIER")
  ident := current.tok_value
  if (isGlobal(ident)) then
    SemanticsError (current.line_no,
      "duplicate_type_identifier_("||ident||")")
  current := read_lexeme ()
  must_match (current,":") &
    current := read_lexeme ()

  # the following entry allows for POINTERS to a node
  # within a node definition
  #
  global_symbol_table[ident] :=
    type_entry(ident,&null,&null)
  #
  # global symbol table entry must be corrected
  # when node definition is complete

  type_info := DeclarationType ()
  type_size := type_info[1]
  type_detail := type_info[2]
  must_match (current,";") &
    current := read_lexeme ()
```

```
  # !insert_type_entry
  global_symbol_table[ident] :=
    type_entry(ident,type_size,type_detail)
  MoreTypeList ()
end

procedure MoreTypeList ()
# more_type_list --> type_list
# more_type_list -->

  if (does_match (current,"IDENTIFIER")) then
    TypeList ()
  else
    return
end

# ------------------------------------------------------------ #

procedure DeclarationType ()
# declare_type --> array_type
# declare_type --> record_type

  if (does_match (current,"ARRAY")) then
    return ArrayType ()
  else if (does_match (current,"RECORD")) then
    return RecordType ()
  else
    SyntaxError (current.line_no,
                 "STRUCTURE",
                 current.tok_value)
end

procedure ArrayType ()
# array_type --> ARRAY [ ub !check_no_elements ]
#                OF declared_type !save_array_info

  current := read_lexeme () # previously matched "ARRAY"!
  must_match (current,"[") &
    current := read_lexeme ()
  # !check_no_elements
  no_elements := UB ()
  must_match (current,"]") &
    current := read_lexeme ()
  must_match (current,"OF") &
    current := read_lexeme ()
  # !save_array_info
  element_type := DeclaredType ()
  if (element_type[1] == "^") then
    element_size := 1
  else
    (element_size :=
      global_symbol_table[element_type].type_size) |
      SemanticsError (current.line_no,
```

```
        "circular reference "||element_type||")")
  array_size := no_elements * element_size
  return [array_size,array_info(no_elements,element_type)]
end

procedure UB ()
# ub --> NUMBER
# ub --> IDENTIFIER !check_int_constant

  if (does_match (current,"NUMBER")) then
  {
    ub := current.tok_value
    current := read_lexeme ()
    return ub
  }
  else if (does_match (current,"IDENTIFIER")) then
  {
    ident := current.tok_value
    current := read_lexeme ()
    (entry := isGlobal(ident)) |
      SemanticsError (current.line_no,
        "undeclared identifier ("||ident||")")
    (isConstant(entry)) |
      SemanticsError (current.line_no,
        "identifier must be named constant ("||ident||")")
    (entry.const_type == "INT") |
      SemanticsError (current.line_no,
        "UB must be type INT")
    return entry.const_value
  }
  else
    SyntaxError (current.line_no,"UB",current.tok_value)
end

procedure RecordType ()
# record_type --> RECORD fld_list END

  current := read_lexeme () # previously matched "RECORD"!
  fields := FldList (0,[])
  record_size := fields[1]
  field_descriptors := fields[2]
  must_match (current,"END") &
    current := read_lexeme ()
  return [record_size,record_info(field_descriptors)]
end

procedure FldList (offset,previous)
# fld_list --> IDENTIFIER !check_duplicate : declared_type ;
#              !insert_field_descriptor
#              more_fld_list

  must_match (current,"IDENTIFIER")
  field_name := current.tok_value
```

```
   current := read_lexeme ()
   # !check_duplicate
   every i := 1 to *previous do
     if (previous[i].identifier == field_name) then
       SemanticsError (current.line_no,
         "duplicate field name ("||field_name||")")
   must_match (current,":") &
     current := read_lexeme ()
   field_type := DeclaredType ()
   if (field_type[1] == "^") then
     field_size := 1
   else
     (field_size :=
       \global_symbol_table[field_type].type_size) |
         SemanticsError (current.line_no,
           "circular reference "||field_type||")")
   must_match (current,";") &
     current := read_lexeme ()
   # !insert_field_descriptor
   put (previous,
        field_descriptor(field_name,field_type,offset))
   offset +:= field_size
   return MoreFldList (offset,previous)
end

procedure MoreFldList (offset,previous)
# more_fld_list --> fld_list
# more_fld_list -->

   if (does_match (current,"IDENTIFIER")) then
     return FldList (offset,previous)
   else
     return [offset,previous]
end

procedure DeclaredType ()
# declared_type --> atomic_type
# declared_type --> IDENTIFIER !check_declared
# declared_type --> ^ declared_type

   if (member (ATOMIC,current.tok_type)) then
   {
     data_type := current.tok_type
     current := read_lexeme ()
     return data_type
   }
   else if (does_match(current,"IDENTIFIER")) then
   {
     data_type := current.tok_value
     current := read_lexeme ()
     # !check_declared
     if (isType(isGlobal(data_type))) then
       return data_type
```

```
    else
      SemanticsError (current.line_no,
        "undeclared_data_type_("||data_type||")")
  }
  else if (does_match (current,"^")) then
  {
    current := read_lexeme ()
    return "^"||DeclaredType()
  }
  else
    SemanticsError (current.line_no,
      "undeclared_data_type_("||current.tok_type||")");
end

#------------------------------------------------------------#

procedure VarDeclarations (offset)
# var_declarations --> VARIABLES var_list
# var_declarations -->

  # offset value indicates declaration of local variables
  # global variable has absolute address and a data size
  # local variable has offset address and a data size
  # data size increments the offset for NEXT local variable
  if (does_match (current,"VARIABLES")) then
  {
    current := read_lexeme ()
    return VarList (offset)
  }
  else
    return offset
end

procedure VarList (offset)
# var_list --> IDENTIFIER !check_duplicate :
#              declared_type ;
#              !insert_variable_entry
#              !insert_static_memory_entry
#              more_var_list

  if (/offset) then
    scope := "global"
  else
    scope := "local"
  must_match (current,"IDENTIFIER")
  ident := current.tok_value
  # !check_duplicate
  if ((scope == "global") & (isGlobal(ident))) then
    SemanticsError (current.line_no,
                    "duplicate_global_identifier_(" ||
                    ident || ")")
  if ((scope == "local") & (isLocal(ident))) then
    SemanticsError (current.line_no,
```

```
                    "duplicate local identifier (" ||
                    ident || ")")
  current := read_lexeme ()
  must_match (current,":") &
    current := read_lexeme ()
  variable_type := DeclaredType ()
  if (variable_type[1] == "^") then
    variable_size := 1
  else
    variable_size :=
      global_symbol_table[variable_type].type_size
  must_match (current,";") &
    current := read_lexeme ()
  # !insert_variable_entry
  if (scope == "global") then
  {
    variable_address := @name_generator
    global_symbol_table[ident] :=
      variable_entry (ident,variable_type,variable_address)
    # !insert_static_memory_entry
    put (static_memory,
         reserve(variable_address,variable_size))
  }
  else
  {
    offset := offset - variable_size
    local_symbol_table[ident] :=
      variable_entry (ident,variable_type,offset)
  }
  return MoreVarList (offset)
end

procedure MoreVarList (offset)
# more_var_list --> var_list
# more_var_list -->

  if (does_match (current,"IDENTIFIER")) then
    return VarList (offset)
  else
    return offset
end

# ------------------------------------------------------------ #

procedure LabelDeclarations ()
# label_declarations --> LABELS label_list
# label_declarations -->

  if (does_match (current,"LABELS")) then
  {
    current := read_lexeme ()
    LabelList ()
  }
```

```
  else
    return
end

procedure LabelList ()
# label_list --> label !check_duplicate ;
#                !insert_label_entry
#                more_label_list

  lnumber := Label ()
  # !check_duplicate
  if (isLocal(lnumber)) then
    SemanticsError (current.line_no,
      "duplicate label encountered ("||lnumber||")")
  must_match (current,";") &
    current := read_lexeme ()
  # !insert_label_entry
  lvalue := @name_generator
  local_symbol_table[lnumber] :=
    label_entry(lnumber,lvalue)
  MoreLabelList ()
end

procedure MoreLabelList ()
# more_label_list --> label_list
# more_label_list -->

  if (does_match (current,"NUMBER")) then
    LabelList ()
  else
    return
end

procedure Label ()
# label --> NUMBER

  must_match (current,"NUMBER")
  lnumber := current.tok_value
  current := read_lexeme ()
  return lnumber
end

# ------------------------------------------------------------ #

procedure ProcDeclarations ()
# proc_declarations --> PROCEDURES proc_list
# proc_declarations -->

  if (does_match (current,"PROCEDURES")) then
  {
    current := read_lexeme ()
    ProcList ()
  }
```

```
  else
    return
end

procedure ProcList ()
# proc_list --> proc_signature more_proc_list

  ProcSignature ()
  MoreProcList ()
end

procedure ProcSignature ()
# proc_signature --> IDENTIFIER !check_duplicate
#                    ( formal_arguments )
#                    return_type ; !insert_proc_entry

  must_match (current,"IDENTIFIER")
  ident := current.tok_value
  current := read_lexeme ()
  # !check_duplicate
  if (isGlobal(ident)) then
    SemanticsError (current.line_no,
      "duplicate_global_identifier_("||ident||")")
  must_match (current,"(") &
    current := read_lexeme ()
  arguments := FormalArguments ([])
  must_match (current,")") &
    current := read_lexeme ()
  return_type := ReturnType ()
  # !insert_proc_entry
  global_symbol_table[ident] :=
    procedure_entry(ident,arguments,return_type)
  must_match (current,";") &
    current := read_lexeme ()
end

procedure MoreProcList ()
# more_proc_list --> proc_list
# more_proc_list -->

  if (not does_match (current,"BEGIN")) then
    ProcList ()
  else
    return
end

procedure FormalArguments (previous)
# formal_arguments --> formal_argument
#                      more_formal_arguments
# formal_arguments -->

  if (does_match (current,")")) then
    return previous
```

```
  else
  {
    put (previous,FormalArgument ())
    return MoreFormalArguments (previous)
  }
end

procedure MoreFormalArguments (previous)
# more_formal_arguments --> , formal_arguments
# more_formal_arguments -->

  if (does_match (current,","")) then
  {
    current := read_lexeme ()
    return FormalArguments(previous)
  }
  else
    return previous
end

procedure FormalArgument ()
# formal_argument --> call_by IDENTIFIER : defined_type
#                     !save_formal_argument

  callby :=CallBy ()
  must_match (current,"IDENTIFIER")
  ident := current.tok_value
  current := read_lexeme ()
  must_match (current,":") &
    current := read_lexeme ()
  arg_type := DeclaredType ()
  if ((isArray(isGlobal(arg_type))) |
      (isRecord(isGlobal(arg_type)))) then
    callby := "var"
  # !save_formal_argument
  return argument_entry(ident,arg_type,callby)
end

procedure CallBy ()
# call_by --> VAL
# call_by --> VAR
# call_by -->

  if (does_match (current,"VAR")) then
  {
    current := read_lexeme ()
    return "var"
  }
  else if (does_match (current,"VAL")) then
  {
    current := read_lexeme ()
    return "val"
  }
```

```
  else
    return "val"
end

procedure ReturnType ()
# return_type --> : atomic_type !check_type
# return_type -->

  if (does_match (current,";")) then
    return &null
  must_match (current,":") &
    current := read_lexeme ()
  # !check_type
  if (member (ATOMIC, current.tok_type)) then
  {
    return_type := current.tok_value
    current := read_lexeme ()
    return return_type
  }
  else
    SyntaxError (current.line_no,
                 "return_type",
                 current.tok_type)
end

procedure ActualArguments  (caller,formal_args)
# actual_arguments --> !check_arg_list_sizes
#                      actual_argument
#                      more_actual_arguments
# actual_arguments -->

  if ((caller == "read") | (caller == "write")) then
  {
    # !check_arg_list_sizes
    # read and write have arbitrary actual argument lists!
    # arg_list_sizes will automatically match
    if (does_match(current,")")) then
      return
    ActualArgument (caller,formal_args)
    MoreActualArguments (caller,formal_args)
  }
  else
  {
    # !check_arg_list_sizes
    # procedures must have a formal argument list
    #   even if empty
    no_args := *formal_args |
      SemanticsError (current.line_no,
        "procedure_(" || caller ||
        ")_must_have_formal_argument_list")
    if ((does_match (current,")")) & (no_args = 0)) then
      # actual arguments match formal arguments
      return
```

```
    if ((does_match (current,")")) & (no_args > 0)) then
      # actual arguments fall short of formal arguments
      SemanticsError (current.line_no,
        "procedure (" || caller ||
        ") has too few actual arguments")
    if (not (does_match (current,")")) &
        (no_args = 0)) then
      # actual arguments exceed formal arguments
      SemanticsError (current.line_no,
        "procedure (" || caller ||
        ") has too many actual arguments")
    ActualArgument (caller,formal_args)
    MoreActualArguments (caller,formal_args)
  }
end

procedure MoreActualArguments (caller,formal_args)
# more_actual_arguments --> , actual_arguments
# more_actual_arguments -->

  if (does_match (current,",")) then
  {
    current := read_lexeme ()
    formal_args := formal_args[2:0]
    return ActualArguments(caller,formal_args)
  }
  else
  {
    return
  }
end

procedure ActualArgument (caller,formal_args)
# actual_argument --> !determine_caller
#                     !determine_call_by
#                     expression (if by val)
#                     | named_item (if by_var)

  # !determine_caller
  if (caller == "read") then
  {
    # !determine_call_by
    named_type := NamedItem ()          # call by var
    if (type(named_type) == "variable_location") then
      emit_line ("push",SAR)
    else
      SemanticsError (current.line_no,
        "invalid item to read ("||type(named_type)||")")
    emit_read (named_type.data_type)
  }
  else if (caller == "write") then
  {
    # !determine_call_by
```

```
    write_type := Expression ()          # call by val
    emit_write (write_type)
  }
  else # (caller is a procedure)
  {
    formal_arg := formal_args[1]
    # !determine_call_by
    callby := formal_arg.call_by
    datatype := formal_arg.data_type
    if (callby == "val") then
      arg_type := Expression ()          # value top stack
    else # (callby == "var") then
    {
      named_type := NamedItem ()         # lvalue to SAR
      if (type(named_type) == "variable_location") then
      {
        emit_line ("push",SAR)           # SAR top stack
        arg_type := named_type.data_type
      }
      else
        SemanticsError (current.line_no,
          "invalid item to read (" || type(named_type) || ")")
    }
  }
end

# ------------------------------------------------------------ #
```

21.1.2 executables: compiling executable code

executables.icn

```
# ------------------------------------------------------------ #

procedure ProcedureList ()
# procedure_list --> procedure more_procedure_list

   Procedure ()
   MoreProcedureList ()
end

procedure MoreProcedureList ()
# more_procedure_list  procedure_list
# more_procedure_list -->

   if (does_match (current,"PROCEDURE")) then
     ProcedureList ()
   else
     return
end

# ------------------------------------------------------------ #

procedure StatementList ()
# statement_list --> statement statement_list
# statement_list -->

   if ((not does_match (current,"END")) &
       (not does_match (current,"UNTIL"))) then
   {
     Statement ()
     StatementList ()
   }
   else
     return
end

procedure Statement ()
# statement --> opt_label executable_statement

   OptLabel ()
   ExecutableStatement ()
end

procedure OptLabel ()
# opt_label --> label !check_label !emit_label
# opt_label -->

   if (does_match (current,"NUMBER")) then
   {
     lnumber := current.tok_value
```

```
    current := read_lexeme ()
    # !check_label
    if (not (isLabel(isLocal(lnumber)))) then
      SemanticsError (current.line_no,
        "undeclared label ("||lnumber||")")
    # !emit_label
    emit_label (local_symbol_table[lnumber].lvalue)
  }
end

procedure ExecutableStatement ()
# executable_statement --> read_statement
# executable_statement --> write_statement
# executable_statement --> assignment_statement
# executable_statement --> goto_statement
# executable_statement --> empty_statement
# executable_statement --> compound_statement
# executable_statement --> if_statement
# executable_statement --> while_statement
# executable_statement --> repeat_statement
# executable_statement --> case_statement
# executable_statement --> for_statement
# executable_statement --> next_statement
# executable_statement --> break_statement
# executable_statement --> call_statement
# executable_statement --> return_statement
# executable_statement --> dispose_request

  if ((does_match (current,"READ")) |
      (does_match (current,"READLN"))) then
    ReadStatement ()
  else if ((does_match (current,"WRITE")) |
           (does_match (current,"WRITELN"))) then
    WriteStatement ()
  else if (does_match (current,"IDENTIFIER")) then
    AssignmentStatement ()
  else if (does_match (current,"GOTO")) then
    GotoStatement ()
  else if (does_match (current,";")) then
    EmptyStatement ()
  else if (does_match (current,"DO")) then
    CompoundStatement ()
  else if (does_match (current,"IF")) then
    IfStatement ()
  else if (does_match (current,"WHILE")) then
    WhileStatement ()
  else if (does_match (current,"REPEAT")) then
    RepeatStatement ()
  else if (does_match (current,"CASE")) then
    CaseStatement ()
  else if (does_match (current,"FOR")) then
    ForStatement ()
  else if (does_match (current,"NEXT")) then
```

```
    NextStatement ()
  else if (does_match (current,"BREAK")) then
    BreakStatement ()
  else if (does_match (current,"CALL")) then
    CallStatement ()
  else if (does_match (current,"RETURN")) then
    ReturnStatement ()
  else if (does_match (current,"DISPOSE")) then
    DisposeRequest ()
  else
    SyntaxError (current.line_no,
                 "EXECUTABLE_STATEMENT",
                 current.tok_value)
end

# ------------------------------------------------------------ #

procedure ReadStatement ()
# read_statement --> READ   ( actual_arguments ) ;
# read_statement --> READLN ( actual_arguments ) ;

  if (does_match (current,"READLN")) then
    eol := "true"
  else
    eol := &null
  current := read_lexeme ()
  must_match (current,"(") &
    current := read_lexeme ()
  # read behaves very much like a called procedure
  # but read does not have a formal argument list
  ActualArguments ("read",[])
  must_match (current,")") &
    current := read_lexeme ()
  must_match (current,";") &
    current := read_lexeme ()
  if (\eol) then
    emit_read ()
end

procedure WriteStatement ()
# write_statement --> WRITE   ( actual_arguments ) ;
# write_statement --> WRITELN ( actual_arguments ) ;

  if (does_match (current,"WRITELN")) then
    eol := "true"
  else
    eol := &null
  current := read_lexeme ()
  must_match (current,"(") &
    current := read_lexeme ()
  # write behaves very much like a called procedure
  # but write does not have a formal argument list
  ActualArguments ("write",[])
```

```
  must_match (current,")") &
    current := read_lexeme ()
  must_match (current,";") &
    current := read_lexeme ()
  if (\eol) then
    emit_write ()
end

#------------------------------------------------------------#

procedure AssignmentStatement ()
# assignment_statement --> named_item !save_type
#                          !save_address
#                          := two_options !save_type
#                          !emit_assign;

  # !save_type
  named_type := NamedItem ()
  # !save_address
  if (type(named_type) == "variable_location") then
    save_register (SAR)
  else
    SemanticsError (current.line_no,
      "invalid item to assign ("||type(named_type)||")")
  must_match (current,":=") &
    current := read_lexeme ()
  # !save_type
  value_type := TwoOptions ()
  if (isPointer(value_type)) then
    value_size := 1
  else
    value_size := global_symbol_table[value_type].type_size
  # !emit_assign
  emit_assign (named_type.data_type,value_type,value_size)
  must_match (current,";") &
    current := read_lexeme ()
end

procedure TwoOptions ()
# two_options --> expression !return_type
# two_options --> new_request !return_type

  if (does_match (current,"NEW")) then
    return NewRequest ()
  else
    return Expression ()
end

#------------------------------------------------------------#

procedure Expression ()
# expression --> simple_expression !save_type comparison !return_type
```

```
  # !save_type
  a_type := SimpleExpression ()
  # !return_type
  return Comparison (a_type)
end

procedure Comparison (a_type)
# comparison --> compop !save_op simple_expression
#                !save_type
#                !resolve_types !emit_comparison
#                !return_type
# comparison --> !return_type

  if (member (COMPOPS, current.tok_type)) then
  {
    # !save_op
    op := Compop ()
    # !save_type
    b_type := SimpleExpression ()
    # !resolve_types
    result_type := resolve_types (a_type, b_type)
    # !emit_comparison
    emit_compop (op, result_type, a_type, b_type)
    return "INT"                  # comparison is BOOL (INT)
  }
  else
    return a_type
end

procedure SimpleExpression ()
# simple_expression --> sign !save_sign term !save_type
#                       !emit_neg more_terms !save_type
#                       !return_type

  # !save_sign
  neg := Sign ()
  # !save_type
  a_type := Term ()
  # !emit_neg
  if (\neg) then
    emit_neg (a_type)
  # !save_type
  result_type := MoreTerms (a_type)
  # !return_type
  return result_type
end

procedure MoreTerms (a_type)
# more_terms --> addop !save_op !check_or
#                term !save_type
#                !resolve_types !emit_mulop
#                more_factors
# more_terms --> addop !save_op !is_or
```

```
#                  !check_type
#                  !emit_or_a
#                  term !save_type
#                  !check_type !emit_or_b
#                  more_terms
# more_terms --> !return_type

  if (member (ADDOPS, current.tok_type)) then
  {
    # !save_op
    op := Addop ()
    # !check_or
    if (op == "OR") then
    {
      label_true := @name_generator
      label_done := @name_generator
      # !check_type
      if (not (a_type == "INT")) then
        SemanticsError (current.line_no,
          "logic operators must be INT (not " || a_type || ")")
      # !emit_or_a
      emit_or_a (label_true)
      # !save_type
      b_type := Term ()
      if (not (b_type == "INT")) then
        SemanticsError (current.line_no,
          "logic operators must be INT (not " || b_type || ")")
      emit_or_b (label_true, label_done)
      return MoreTerms ("INT")
    }
    else
    {
      # !save_type
      b_type := Term ()
      result_type := resolve_types (a_type, b_type)
      emit_addop (op, result_type, a_type, b_type)
      return MoreTerms (result_type)
    }
  }
  else
    return a_type
end

procedure Term ()
# term --> factor !save_type more_factors !save_type
#          !return_result

  # !save_type
  a_type := Factor ()
  # !save_type
  result_type := MoreFactors (a_type)
  # !return_result
  return result_type
```

```
end

procedure MoreFactors (a_type)
# more_factors --> mulop !save_op !not_and
#                  factor !save_type
#                  !resolve_types !emit_mulop
#                  more_factors_a
# more_factors --> mulop !save_op !is_and
#                  !check_type
#                  !emit_and_a
#                  factor !save_type
#                  !check_type !emit_and_b
#                  more_factors
# more_factors --> !return_type

   if (member (MULOPS, current.tok_type)) then
   {
     # !save_op
     op := Mulop ()
     # !check_and
     if (op == "AND") then
     {
       label_false := @name_generator
       label_done := @name_generator
       # !check_type
       if (not (a_type == "INT")) then
         SemanticsError (current.line_no,
           "logic operators must be INT (not " || a_type || ")")
       # !emit_and_a
       emit_and_a (label_false)
       # !save_type
       b_type := Factor ()
       if (not (b_type == "INT")) then
         SemanticsError (current.line_no,
           "logic operators must be INT (not " || b_type || ")")
       emit_and_b (label_false, label_done)
       return MoreFactors ("INT")
     }
     else
     {
       # !save_type
       b_type := Factor ()
       result_type := resolve_types (a_type, b_type)
       emit_mulop (op, result_type, a_type, b_type)
       return MoreFactors (result_type)
     }
   }
   else
     return a_type
end

procedure Factor ()
# factor --> named_item !emit_rvalue !return_type
```

```
# factor --> ( expression !save_type ) !return_type
# factor --> NOT factor !save_type !emit_not !return_type
# factor --> const !save_type !return_type
# factor --> & named_item !save_type !return_type

  if (does_match (current,"IDENTIFIER")) then
  {
    if (isConstant(isGlobal(current.tok_value))) then
    {
      entry := global_symbol_table[current.tok_value]
      current := read_lexeme ()
      const_type := entry.const_type
      const_value := entry.const_value
      emit_literal (const_type,const_value)
      # !return_type
      return const_type
    }
    else
    {
      # !save_type
      named_type := NamedItem ()
      # !emit_rvalue
      if (type(named_type) == "variable_location") then
        emit_Rvalue (named_type.data_type)
      # !return_type
      result := named_type.data_type
      return result
    }
  }
  else if (does_match (current,"(")) then
  {
    current := read_lexeme ()
    # !save_type
    data_type := Expression ()
    must_match (current,")") & current := read_lexeme ()
    # !return_type
    return data_type
  }
  else if (does_match (current,"NOT")) then
  {
    current := read_lexeme ()
    # !save_type
    data_type := Factor ()
    # !emit_not
    if (data_type == "INT") then
      emit_not (data_type)
    else
      SemanticsError (current.line_no,
        "incorrect data (" || data_type || ") to NOT")
    # !return_type
    return data_type
  }
  else if (does_match (current,"&")) then
```

```
  {
    current := read_lexeme ()
    # !save_type
    named_type := NamedItem ()
    if (type(named_type) == "variable_location") then
      emit_line ("push",SAR)
    else
      SemanticsError (current.line_no,
        "invalid_item_to_get_address_of_("||
        type(named__type)||")")
    return "^"||named_type.data_type
  }
  else # factor is literal constant
  {
    # !save_type
    literal := Literal ()
    literal_type  := literal[1]
    literal_value := literal[2]
    emit_literal (literal_type,literal_value)
    # !return_type
    return literal_type
  }
end

procedure NamedItem (called)
# named_item --> IDENTIFIER !save_ident !retrieve_info
#                qualifier !result_type

# named item returns one of two possible results
#   variable_location = address for the named item
#   procedure_evaluation = result
#      is found in appropriate register (IA or FA)

  must_match (current,"IDENTIFIER")
  # !save_ident
  ident := current.tok_value
  if (isLocal(ident)) then
  {
    entry := local_symbol_table[ident]
    if (isVariable(entry)) then
    {
      if (\called) then
        SemanticsError (current.line_no,
          "only_subroutines_are_called_("||ident||")")
      current := read_lexeme ()
      scope := "local"
      emit_Lvalue (entry.variable_address,scope)
      if (\entry.indirect) then
        emit_Indirect ()
      # !result_type
      return variable_location(
        StructureQualifier (entry.variable_type))
    }
```

```
    else
      SemanticsError (current.line_no,
        "local label (" || ident ||
        ") is not a valid named item")
  }
  else if (isGlobal(ident)) then
  {
    entry := global_symbol_table[ident]
    if (isVariable(entry)) then
    {
      if (\called) then
        SemanticsError (current.line_no,
          "only subroutines are called ("||ident||")")
      current := read_lexeme ()
      scope := "global"
      emit_Lvalue (entry.variable_address,scope)
      if (\entry.indirect) then
        emit_Indirect ()
      # !result_type
      return variable_location(
        StructureQualifier (entry.variable_type))
    }
    else if (isProcedure(entry)) then
    {
      current := read_lexeme ()
      procedure_name := entry.identifier
      formal_arguments := entry.formal_arguments
      return_type := entry.return_type
      if (\called & \return_type) then
        SemanticsError (current.line_no,
          "functions (" || procedure_name ||
          ") must not be called!")
      if (/called & /return_type) then
        SemanticsError (current.line_no,
          "subroutines (" || procedure_name ||
          ") must be called!")
      ProcedureQualifier (procedure_name,formal_arguments)
      emit_call (procedure_name,return_type)
      return procedure_evaluation(entry.return_type)
    }
    else
      SemanticsError (current.line_no,
        "global identifier (" || ident ||
        ") is not a proper named item")
  }
  else
    SemanticsError (current.line_no,
      "inappropriate named item ("||ident||")")
end

procedure Addop ()
# addop --> +
# addop --> -
```

```
# addop --> OR

   if (member (ADDOPS, current.tok_type)) then
   {
     op := current.tok_value
     current := read_lexeme ()
     return op
   }
   else
     fail
end

procedure Mulop ()
# mulop --> *
# mulop --> /
# mulop --> %
# mulop --> AND

   if (member (MULOPS, current.tok_type)) then
   {
     op := current.tok_value
     current := read_lexeme ()
     return op
   }
   else
     fail
end

procedure Sign ()
# sign --> + !return_sign
# sign --> - !return_sign
# sign --> !return_sign

   if (does_match (current,"+")) then
   {
     current := read_lexeme ()
     fail
   }
   else if (does_match (current,"-")) then
   {
     current := read_lexeme ()
     return "true"
   }
   else
     fail
end

procedure Compop ()
# compop --> =
# compop --> <>
# compop --> >
# compop --> >=
# compop --> <
```

```
# compop --> <=

   if (member (COMPOPS, current.tok_type)) then
   {
     op := current.tok_value
     current := read_lexeme ()
     return op
   }
   else
     fail
end

# ------------------------------------------------------------ #

procedure GotoStatement ()
# goto_statement --> GOTO label !check_label !emit_goto ;

   must_match (current,"GOTO") & current := read_lexeme ()
   lnumber := Label ()
   # !check_label
   if (not (isLabel(isLocal(lnumber)))) then
     SemanticsError (current.line_no,
       "undefined label ("||lnumber||")")
   # !emit_goto
   lvalue := local_symbol_table[lnumber].lvalue
   emit_goto (lvalue)
   must_match (current,";") & current := read_lexeme ()
end

procedure EmptyStatement ()
# empty_statement --> !emit_empty ;

   emit_empty ()
   must_match (current,";") &
     current := read_lexeme ()
end

# ------------------------------------------------------------ #

procedure CompoundStatement ()
# compound_statement --> DO !emit_do_a
#                          statement_list
#                        END !emit_do_b ;

   must_match (current,"DO") &
     current := read_lexeme ()
   emit_do_a ()
   StatementList ()
   must_match (current,"END") &
     current := read_lexeme ()
   emit_do_b ()
   must_match (current,";") &
     current := read_lexeme ()
```

```
end

# ------------------------------------------------------------ #

procedure IfStatement ()
# if_statement --> IF !emit_if_a ( expression !check_type )
#                  !emit_test_a then_clause
#                  !emit_test_b else_clause
#                  !emit_test_c !emit_if_b

   must_match (current,"IF") &
     current := read_lexeme ()
   # !emit_if_a
   emit_if_a ()
   must_match (current,"(") &
     current := read_lexeme ()
   # !check_type
   data_type := Expression ()
   if (data_type ~== "INT") then
     SemanticsError (current.line_no,
       "test expression must be of type INT")
   must_match (current,")") &
     current := read_lexeme ()
   # !emit_test_a
   label_false := @name_generator
   label_done := @name_generator
   emit_test_a (label_false,label_done)
   ThenClause ()
   emit_test_b (label_false,label_done)
   ElseClause ()
   emit_test_c (label_false,label_done)
   emit_if_b ()
end

procedure ThenClause ()
# then_clause --> THEN statement

   must_match (current,"THEN") &
     current := read_lexeme ()
   Statement ()
end

procedure ElseClause ()
# else_clause --> ELSE statement
# else_clause -->

   if (does_match (current,"ELSE")) then
   {
     current := read_lexeme ()
     Statement ()
   }
   else
     return
```

```
end

procedure CaseStatement ()
# case_statement --> CASE !emit_case_a
#                    ( expression !check_type ) OF
#                    case_list !emit_case_b

  must_match (current,"CASE") &
    current := read_lexeme ()
  # !emit_case_a
  emit_case_a ()
  must_match (current,"(") &
    current := read_lexeme ()
  # !check_type
  target_type := Expression ()
  if (target_type ~== "INT") then
    SemanticsError (current.line_no,
      "case expression must be of type INT")
  must_match (current,")") &
    current := read_lexeme ()
  must_match (current,"OF") &
    current := read_lexeme ()
  label_exit := @name_generator
  CaseList (label_exit)
  # !emit_case_b
  emit_case_b ()
end

procedure CaseList (label_exit)
# case_list --> NUMBER !emit_case_compare : statement
#               !emit_next_case case_list
# case_list --> DEFAULT !discard_target : statement
#               !exit_next_case

  if (does_match (current,"NUMBER")) then
  {
    case_value := current.tok_value
    current := read_lexeme ()
    must_match (current,":") &
      current := read_lexeme ()
    # !emit_case_compare
    label_next := @name_generator
    emit_case_compare (case_value,label_next)
    Statement ()
    # !emit_next_case
    emit_next_case (label_exit,label_next)
    CaseList ()
  }
  else if (does_match (current,"DEFAULT")) then
  {
    current := read_lexeme ()
    must_match (current,":") &
      current := read_lexeme ()
```

```
    # !discard_target
    emit_line ("pop",IA,,,"discard target value")
    Statement ()
    # !emit_next_case
    emit_next_case (label_exit)
  }
  else
    SyntaxError (current.line_no,
                 "CASE entry",
                 current.tok_type)
end

# ------------------------------------------------------------ #

procedure WhileStatement ()
# while_statement --> WHILE !emit_while_a
#                     ( expression !check_type )
#                     !emit_test_a statement
#                     !emit_test_b !break
#                     !emit_test_c !next
#                     !emit_while_b

  must_match (current,"WHILE") &
    current := read_lexeme ()
  # !emit_while_a
  label_top := label_next := @name_generator
  label_bottom := label_break := @name_generator
  push (control_stack,[label_next,label_break])
  emit_while_a (label_top)
  must_match (current,"(") &
    current := read_lexeme ()
  data_type := Expression ()
  if (data_type ~== "INT") then
    SemanticsError (current.line_no,
      "test expression must be of type INT")
  must_match (current,")") &
    current := read_lexeme ()
  # !emit_test_a
  label_false := @name_generator
  label_done := @name_generator
  emit_test_a (label_false,label_done)
  Statement ()
  emit_test_b (label_false,label_done)
  # !break
  emit_line ("jmp",label_break)
  emit_test_c (label_false,label_done)
  # !next
  emit_line ("jmp",label_next)
  # !emit_while_b
  emit_while_b (label_bottom)
  pop (control_stack)
end
```

```
procedure RepeatStatement ()
# repeat_statement --> REPEAT !emit_repeat_a
#                      statement_list
#                      UNTIL ( expression !check_type )
#                      !emit_test_a !break
#                      !emit_test_b !nop
#                      !emit_test_c !top
#                      !emit_repeat_b ;

  must_match (current,"REPEAT") &
    current := read_lexeme ()
  # !emit_repeat_a
  label_top := @name_generator
  label_next := @name_generator
  label_bottom := label_break := @name_generator
  push (control_stack,[label_next,label_break])
  emit_repeat_a (label_top)
  StatementList ()
  emit_label (label_next)
  must_match (current,"UNTIL") &
    current := read_lexeme ()
  must_match (current,"(") &
    current := read_lexeme ()
  data_type := Expression ()
  if (data_type ~== "INT") then
    SemanticsError (current.line_no,
      "test expression must be of type INT")
  must_match (current,")") &
    current := read_lexeme ()
  # !emit_test_a
  label_false := @name_generator
  label_done := @name_generator
  emit_test_a (label_false,label_done)
  # !break
  emit_line ("jmp",label_break)
  emit_test_b (label_false,label_done)
  # !nop
  emit_line ("nop")
  emit_test_c (label_false,label_done)
  # !top
  emit_line ("jmp",label_top)
  must_match (current,";") & current := read_lexeme ()
  # !emit_repeat_b
  emit_repeat_b (label_bottom)
  pop (control_stack)
end

procedure ForStatement ()
# for_statement --> FOR !emit_for_a IDENTIFIER !check_type :=
#                   initial direction final
#                   !emit_for_setup !emit_for_test
#                   statement
#                   !emit_for_increment
```

```
  must_match (current,"FOR") &
    current := read_lexeme ()
  # !emit_for_a
  emit_for_a ()
  must_match (current,"IDENTIFIER")
  counter := current.tok_value
  current := read_lexeme ()
  # !check_type
  if (\local_symbol_table[counter]) then
  {
    counter_info := local_symbol_table[counter]
    counter_type := counter_info.variable_type
    counter_address := counter_info.variable_address
    scope := "local"
  }
  else if (\global_symbol_table[counter]) then
  {
    counter_info := global_symbol_table[counter]
    counter_type := counter_info.variable_type
    counter_address := counter_info.variable_address
    scope := "global"
  }
  else
    SemanticsError (current.line_no,
      "for␣loop␣counter␣must␣be␣declared␣("||counter||")")
  if (counter_type ~== "INT") then
    SemanticsError (current.line_no,
      "for␣loop␣counter␣must␣be␣INT")
  emit_Lvalue (counter_address,scope)
  emit_line ("push",SAR)
  must_match (current,":=") &
    current := read_lexeme ()
  Initial ()
  flag := Direction ()
  Final ()
  label_top := @name_generator
  label_next := @name_generator
  label_break := @name_generator
  push (control_stack,[label_next,label_break])
  # !for_setup
  emit_for_setup ()
  # !for_test
  emit_for_test (flag,label_top,label_break)
  Statement ()
  # !for_increment
  emit_for_increment (flag,label_top,
                      label_next,label_break)
  pop (control_stack)
end

procedure Initial ()
# initial --> expression
```

```
   initial_type := Expression ()
   if (initial_type ~== "INT") then
     SemanticsError (current.line_no,
       "for loop initial value must be INT")
end

procedure Final ()
# final --> expression

   final_type := Expression ()
   if (final_type ~== "INT") then
     SemanticsError (current.line_no,
       "for loop final value must be INT")
end

procedure Direction ()
# direction --> UPTO
# direction --> DOWNTO

   if (does_match (current,"UPTO")) then
   {
     current := read_lexeme ()
     return "up"
   }
   else if (does_match (current,"DOWNTO")) then
   {
     current := read_lexeme ()
     return "down"
   }
   else
     SyntaxError (current.line_no,
                  "FOR direction",
                  current.tok_type)
end

# ------------------------------------------------------------ #

procedure NextStatement ()
# next_statement --> NEXT ; !get_next !go_there

   must_match (current,"NEXT") &
     current := read_lexeme ()
   must_match (current,";") &
     curent := read_lexeme ()
   # !get_next
   stack_info := control_stack[1]
   next_label := stack_info[1]
   # !go_there
   emit_line ("jmp",next_label,,,"next statement")
end

procedure BreakStatement ()
```

```
# break_statement --> BREAK ; !get_break !go_there

   must_match (current,"BREAK") &
     current := read_lexeme ()
   must_match (current,";") &
     current := read_lexeme ()
   # !get_break
   stack_info := control_stack[1]
   break_label := stack_info[2]
   # !go_there
   emit_line ("jmp",break_label,,,"break_statement")
end

# ------------------------------------------------------------ #

procedure CallStatement ()
# call_statement --> CALL named_item ;

   must_match (current,"CALL") &
     current := read_lexeme ()
   named_type := NamedItem ("called")
   must_match (current,";") &
     current := read_lexeme ()
end

procedure ReturnStatement ()
# return_statement --> RETURN opt_value ;

   must_match (current,"RETURN") &
     current := read_lexeme ()
   opt_type := OptValue ()
   must_match (current,";") &
     current := read_lexeme ()
   return opt_type
end

procedure OptValue ()
# opt_value --> expression
# opt_value -->

   if (not does_match (current,";")) then
     return Expression ()
   else
     return &null
end

# ------------------------------------------------------------ #

procedure Qualifier (info)
# qualifier --> procedure_qualifier !return_result_type
# qualifier --> structure_qualifier !return_result_type

# this procedure is now obsolete!
```

```
# NamedItem performs these test and directs control
# to the appropriate qualifier.
end

procedure ProcedureQualifier (proc_name,formal_arguments)
# procedure_qualifier --> ( actual_arguments )

  must_match (current,"(") &
    current := read_lexeme ()
  ActualArguments (proc_name,formal_arguments)
  must_match (current,")") &
    current := read_lexeme ()
end

procedure StructureQualifier (data_type)
# structure_qualifier --> array_qualifier
#                         !return_data_type
# structure_qualifier --> record_qualifier
#                         !return_data_type
# structure_qualifier --> pointer_qualifier
#                         !return_data_type
# structure_qualifier --> !return_data_type

  if (does_match (current,"[")) then
    return ArrayQualifier (data_type)
  else if (does_match (current,".")) then
    return RecordQualifier (data_type)
  else if (does_match (current,"^")) then
    return PointerQualifier (data_type)
  else
    return data_type
end

procedure ArrayQualifier (data_type)
# array_qualifier --> !check_array !save_SAR
#                     [ expression !check_index ]
#                     !calc_array_offset
#                     structure_qualifier !return_type

  # !check_array
  entry := global_symbol_table[data_type]
  if (not (type(entry.type_info) == "array_info")) then
    SemanticsError (current.line_no,
      "attempting␣array␣qualification␣on␣non-array␣element")
  else
  {
    # !save_SAR
    save_register (SAR)
    array_structure := entry.type_info
    no_elements := array_structure.no_elements
    element_type := array_structure.data_type
    element_size :=
      \global_symbol_table[element_type].type_size
```

```
    must_match (current,"[") &
      current := read_lexeme ()
    index_type := Expression ()
    # !check_index
    if (not (index_type == "INT")) then
      SemanticsError (current.line_no,
        "array index must be INT")

#   possibly add this later??
#   it requires run-time testing
#     within assembly language code
#   if (not (0 <= index <= no_elements-1) then
#     SemanticsError (current.line_no,
#       "array index out of bounds")

    must_match (current,"]") &
      current := read_lexeme ()
    # !calc_array_offset
    calc_array_offset (element_size)
    # !return_type
    return StructureQualifier (element_type)
  }
end

procedure RecordQualifier (data_type)
# record_qualifier --> !check_record !save_SAR . IDENTIFIER
#                      !calc_record_offset
#                      structure_qualifier !return_type

  # !check_record
  entry := global_symbol_table[data_type]
  if (not (type(entry.type_info) == "record_info")) then
    SemanticsError (current.line_no,
      "attempting record qualification " ||
      "on non-record element")
  else
  {
    # !save_SAR
    save_register (SAR)
    record_structure := entry.type_info
    must_match (current,".") &
      current := read_lexeme ()
    must_match (current,"IDENTIFIER")
    ident := current.tok_value
    current := read_lexeme ()
    fields := record_structure.field_descriptors
    loc := -1
    every (i := 0 to *fields) do
      if (ident == fields[i].identifier) then
        loc := i
    if (loc < 0) then
      SemanticsError (current.line_no,
        "unrecognized field name (" || ident || ")")
```

```
    field_type := fields[loc].field_type
    field_offset := fields[loc].field_offset
    !calc_record_offset (field_offset)
    # !return_type
    return StructureQualifier (field_type)
  }
end

procedure PointerQualifier (data_type)
# pointer_qualifier --> ^ structure_qualifier

  must_match (current,"^") &
    current := read_lexeme ()
  if (data_type == "POINTER") then
    SemanticsError (current.line_no,
      "reference through the NULL pointer")
  else if (not (data_type[1] == "^")) then
    SemanticsError (current.line_no,
      "reference through a non-pointer (" || data_type || ")")
  else
  {
    emit_line ("ldr",SAR,SAR,,"pointer qualifier")
    return StructureQualifier (data_type[2:0])
  }
end

# ------------------------------------------------------------ #

procedure NewRequest ()
# new_request --> NEW defined_type

  must_match (current,"NEW") &
    current := read_lexeme ()
  new_type := DeclaredType ()
  new_size := global_symbol_table[new_type].type_size
  emit_line ("movi",IB,new_size,,"new request")
  emit_line ("malloc",IA,IB)
  emit_line ("push",IA)
  return "^" || new_type
end

procedure DisposeRequest ()
# dispose_request --> DISPOSE named_item

  must_match (current,"DISPOSE") &
    current := read_lexeme ()
  named_type := NamedItem ()
  if (type(named_type) == "variable_location") then
  {
    dispose_type := named_type.data_type
    dispose_size := 0
    emit_line ("mov",SAR,IA,,"dispose request")
    emit_line ("dalloc",IA,IB)
```

```
  }
  else
    SemanticsError (current.line_no,
      "invalid item to get address of (" ||
      type(named_type) || ")")
end

# ------------------------------------------------------------ #
```

21.2 tables: symbol table

tables.icn

```
# ---------------------------------------------------------- #

# tables.icn

# defines the data structures to support the kize compiler
# - global_symbol_table
# - local_symbol_table
# - static_memory

# support routines for the data structures
# - initialize_global_symbol_table
# - display_global_symbol_table
# - initialize_local_symbol_table
# - display_local_symbol_table
# - setup_static_memory

# ---------------------------------------------------------- #

global global_symbol_table
global local_symbol_table

record constant_entry   (identifier,const_type,const_value)
record type_entry       (identifier,type_size,type_info)
record variable_entry   (identifier,variable_type,
                         variable_address,indirect)
record procedure_entry  (identifier,formal_arguments,
                         return_type)
record label_entry      (lnumber,lvalue)

# variable_information is location where data may be found
# the location stored in source address register (SAR)
# the variable may be atomic type, or structured type,
#    or structure-qualified type, or pointer-qualified type
record variable_location        (data_type)

# procedure_evaluation yields at most one atomic data value
# however, the evaluated result found at top of stack
record procedure_evaluation     (data_type)

# formal_arguments is a list of
record argument_entry   (identifier,data_type,call_by)

# type_info is one of two categories:
record array_info       (no_elements,data_type)
record record_info      (field_descriptors)

# field descriptors is a list of
record field_descriptor (identifier,field_type,field_offset)
```

```
# ---------------------------------------------------------- #

global static_memory

# static_memory is a list of memory allocations, either:
record define                   (identifier,data_value)
record reserve                  (identifier,data_size)

procedure initialize_global_symbol_table ()
  global_symbol_table :=
    table ()
  global_symbol_table ["INT"] :=
    type_entry ("INT",1,&null)
  global_symbol_table ["REAL"] :=
    type_entry ("REAL",1,&null)
  global_symbol_table ["STRING"] :=
    type_entry ("STRING",1,&null)
  global_symbol_table ["MAIN"] :=
    procedure_entry ("MAIN",[],"INT")
  static_memory :=
    []
end

procedure initialize_local_symbol_table (formal_arg_list)
  local_symbol_table            := table ()
  offset := *formal_arg_list + 2
  every (item := !formal_arg_list) do
  {
    offset := offset - 1
    if (item.call_by == "var") then
      indirect := "indirect"
    else
      indirect := &null
    local_symbol_table [item.identifier] :=
      variable_entry (item.identifier,item.data_type,
      offset,indirect)
  }
end

procedure display_global_symbol_table ()
  dashes := "----------------------------------------------------"

  emit_comment (dashes)
  emit_blank_line ()
  emit_comment ("GLOBAL_SYMBOL_TABLE:")
  emit_blank_line ()

  emit_comment ("CONSTANTS")
  every (entry := !global_symbol_table) do
    if (isConstant(entry)) then
      write (left ("#",5),
             left (entry.identifier,15),
             left (entry.const_type,15),
```

```
            entry.const_value)
  emit_blank_line ()

  emit_comment ("TYPES")
  every (entry := !global_symbol_table) do
    if (isType(entry)) then
    {
      writes (left ("#",5),
              left (entry.identifier,15),
              left (entry.type_size,15))
      if (isAtomic(entry)) then
        write ("ATOM")
      else if (isArray(entry)) then
      # arrays are discussed in a later chapter
      {
        write ("ARRAY")
        write (left ("#",15),
               left (entry.type_info.no_elements,5),
               entry.type_info.data_type)
      }
      else if (isRecord(entry)) then
      # records are discussed in a later chapter
      {
        write ("RECORD")
        every (item :=
               !(entry.type_info.field_descriptors)) do
          write (left ("#",15),
                 left (item.identifier,15),
                 left (item.field_type,15),
                 item.field_offset)
      }
      else
        stop ("unexpected item in global type table")
    }
  emit_blank_line ()

  emit_comment ("VARIABLES")
  every (entry := !global_symbol_table) do
    if (isVariable(entry)) then
      write (left ("#",5),
             left (entry.identifier,15),
             left (entry.variable_type,15),
             left (entry.variable_address,10),
             (\entry.indirect | ""))
  emit_blank_line ()

  emit_comment ("PROCEDURES")
  # procedures and argument lists are in later chapter
  every (entry := !global_symbol_table) do
    if (isProcedure(entry)) then
    {
      write (left ("#",5),
             left (entry.identifier,15),
```

```
            left (*entry.formal_arguments,15),
            \entry.return_type | "none")
      if (*entry.formal_arguments > 0) then
        every (arg := !entry.formal_arguments) do
          write (left ("#",15),
                 left (arg.identifier,15),
                 left (arg.data_type,15),
                 arg.call_by)
    }
  emit_blank_line ()
  emit_comment (dashes)
  emit_blank_line ()
end

procedure display_local_symbol_table ()
  dashes := "------------------------------------------------------------"

  emit_comment (dashes)
  emit_blank_line ()
  emit_comment ("LOCAL SYMBOL TABLE:")
  emit_blank_line ()

  emit_comment ("LABELS")
  every (entry := !local_symbol_table) do
    if (isLabel(entry)) then
      write (left ("#",5),
             left (entry.lnumber,15),
             entry.lvalue)
  emit_blank_line ()

  emit_comment ("VARIABLES")
  every (entry := !local_symbol_table) do
    if (isVariable(entry)) then
      write (left ("#",5),
             left (entry.identifier,15),
             left (entry.variable_type,15),
             left (entry.variable_address,10),
             (\entry.indirect | ""))
  emit_blank_line ()
  emit_comment (dashes)
  emit_blank_line ()
end

#------------------------------------------------------------#

procedure setup_static_memory ()
  every entry := !static_memory do
    if (type(entry) == "define") then
      emit_line ("_DEFINE",entry[1],entry[2])
    else # (type(entry) == "reserve")
      emit_line ("_RESERVE",entry[1],entry[2])
end
```

```
# ------------------------------------------------------------ #

procedure isGlobal (ident)
  return \global_symbol_table[ident]
end

procedure isConstant (entry)
  return (type(entry) == "constant_entry")
end

procedure isType (entry)
  return type(entry) == "type_entry"
end

procedure isVariable (entry)
  return (type(entry) == "variable_entry")
end

procedure isProcedure (entry)
  return type(entry) == "procedure_entry"
end

procedure isLocal (ident)
  return \local_symbol_table[ident]
end

procedure isLabel (entry)
  return type(entry) == "label_entry"
end

# procedure isVariable (entry)
#   defined earler also applies to local identifiers
# end

procedure isAtomic (entry)
  return isType(entry) &
    (type(entry.type_info) == "null")
end

procedure isArray (entry)
  # arrays are discussed in a later chapter
  return isType(entry) &
    (type(entry.type_info) == "array_info")
end

procedure isRecord (entry)
  # records are discussed in a later chapter
  return isType(entry) &
    (type(entry.type_info) == "record_info")
end

procedure isPointer (data_type)
  # pointers are discussed in a later chapter
```

```
  if (data_type == "POINTER") then
    return data_type
  else if (data_type[1] == "^") then
  {
    target_type := data_type [2:0]
    if (\global_symbol_table [target_type]) then
      return "^" || target_type
    else
      return "^" || isPointer (target_type)
  }
  else
    fail
end

#---------------------------------------------------------------#

procedure resolve_types (a_type,b_type)
  if ((a_type == "STRING") |
      (b_type == "STRING")) then
    stop ("binary operations on STRINGs not permitted!")
  if ((member(ATOMIC, a_type)) &
      (member(ATOMIC, b_type))) then
    if ((a_type == "REAL") |
        (b_type == "REAL")) then
      return "REAL"
    else
      return "INT"
  # structured types and pointers discussed in later chapter
  else if (isType(isGlobal(a_type)) &
           isType(isGlobal(b_type))) then
    if (a_type == b_type) then
      return a_type
    else
      stop ("incompatible types (" ||
            a_type || "," || b_type || ")")
  else if (isPointer(a_type) &
           isPointer(b_type)) then
    if (a_type == b_type) then
      return a_type
    else if (a_type == "POINTER") then
      return b_type
    else if (b_type == "POINTER") then
      return a_type
    else
      stop ("incompatible pointer types ("  ||
            a_type || "," || b_type || ")")
  else
    stop ("incompatible types (" ||
          a_type || "," || b_type || ")")
end

#---------------------------------------------------------------#
```

21.3 assembler: code generation

assembler.icn

```
# ---------------------------------------------------------- #

# assembler.icn

# a collection of procedures to aid
# in generating kcode assembly language

# ---------------------------------------------------------- #

global  IA,IB,IC,ID,SAR,DAR,OR,IR,ZR,SP,FP,RP,HP
global  FA,FB,FC,FD
global  name_generator

# ---------------------------------------------------------- #

procedure initialize_assembler ()
  # procedure to initialize register sets
  # and to create the name generator

  IA        := "I0"         # integer registers
  IB        := "I1"
  IC        := "I2"
  ID        := "I3"

  SAR       := "I4"         # source      address register
  DAR       := "I5"         # destination address register
  OR        := "I6"         # offset register
  IR        := "I7"         # index  register

  ZR        := "I11"        # zero   register
  SP        := "I12"        # frame  pointer
  FP        := "I13"        # stack  pointer
  RP        := "I14"        # return pointer
  HP        := "I15"        # heap   pointer

  FA        := "F0"         # floating point registers
  FB        := "F1"
  FC        := "F2"
  FD        := "F3"

  name_generator          := create ("K"||seq())
end

# ---------------------------------------------------------- #

procedure emit_blank_line ()
  # generate a blank line within kbox assembly code

  write ()
```

```
end

procedure emit_comment (message)
  # insert a single comment line in kbox assembly code

  write ("# ",message)
end

procedure emit_label (label)
  # insert an internal label name into kbox assembly code

  emit_line ("_LABEL",label)
end

procedure emit_line (opcode,oper1,oper2,oper3,comment)
  # generate a single line of code in kbox assembly code

  writes (left ("",10))
  writes (left (opcode,15))
  if (\oper1) then
  {
    ops := oper1
    if (\oper2) then
    {
      ops := ops||" "||oper2
      if (\oper3) then
        ops := ops||" "||oper3
    }
  }
  else
    ops := ""
  if ((opcode == "_DEFINE")  |
      (opcode == "_RESERVE") |
      (*ops > 25)) then
  {
    write (ops)
    return
  }
  writes (left (ops,25))
  if (/comment) then
    write ()
  else
    write ("# "||comment)
end

# ---------------------------------------------------------- #

procedure emit_program_prologue (prog_name)
  # emit kbox assembly code to:
  #   - define the text section
  #   - identify entry point "main" for kbox assembly code

  emit_comment ("BEGIN_PROGRAM: "||prog_name)
```

```
  emit_blank_line ()
  display_global_symbol_table ()
  emit_blank_line ()
  emit_line ("_CODE_SEGMENT")
  emit_blank_line ()
  emit_line ("_GLOBAL","MAIN")
  emit_blank_line ()
end

procedure emit_program_epilogue (prog_name)
  # emit kbox assembly code to:
  #   - define the data section
  #   - display
  #     . symbol table
  #     . static memory

  emit_blank_line ()
  emit_line ("_DATA_SEGMENT")
  emit_blank_line ()
  setup_static_memory ()
  emit_blank_line ()
  emit_comment ("END_PROGRAM: " || prog_name)
  emit_blank_line ()
end

# ------------------------------------------------------------ #

procedure emit_procedure_prologue (proc_name,local_vars)
  # emit kbox assembly code to:
  #   - display the new procedure name as a comment
  #   - set up a new activation stack item
  #     - save return address
  #     - save old frame pointer
  #     - move current stack pointer to new frame pointer

  # remember that the "caller" has the responsibility
  #   - evaluate the actual arguments
  #   - and push appropriate info onto activation stack
  #   - prior to the transfer of control

  emit_blank_line ()
  emit_comment ("BEGIN_PROCEDURE: " || proc_name)
  emit_blank_line ()
  display_local_symbol_table ()
  emit_blank_line ()
  emit_label (proc_name)
  emit_line ("push",RP,,,"save_the_RP_to_stack")
  emit_line ("push",FP,,,"save_the_FP_to_stack")
  emit_line ("mov",FP,SP,,"current_SP_become_the_new_FP")
  if (\local_vars) then
  {
    emit_line ("movi",OR,local_vars,,
               "space_for_local_variables")
```

```
    emit_line ("add",SP,SP,OR)
  }
  emit_blank_line ()
end

procedure emit_procedure_epilogue (proc_name)
  # emit kbox assembly code to:
  #   - display the procedure name as a comment
  #   - clean up the obsolete activation stack item
  #     - retrieve old stack pointer from frame pointer
  #     - retrieve old frame pointer
  #     - retrieve return address

  emit_blank_line ()
  emit_label ("EXIT"||proc_name)
  # the following automatically cleans up local variables!
  emit_line ("mov",SP,FP,,
             "retrieve old SP from current FP")
  emit_line ("pop",FP,,,"retrieve FP from stack")
  emit_line ("pop",RP,,,"retrieve RP from stack")
  emit_line ("ret",,,,"return to calling procedure")
  emit_blank_line ()
  emit_comment ("END PROCEDURE: "||proc_name)
  emit_blank_line ()
end

#------------------------------------------------------------#

procedure save_register (reg)
  # emit kbox assembly code to:
  # save the value found in the specified reg
  # to the top of the stack

  emit_line ("push",reg,,,reg||" --> stack")
end

procedure retrieve_register (reg)
  # emit kbox assembly code to:
  # save the value found on the top of the stack
  # to the specified reg

  emit_line ("pop",reg,,,"stack --> "||reg)
end

#------------------------------------------------------------#

procedure emit_Lvalue (addr,scope)
  # emit kbox assembly code to:
  # move the address of the identifier
  # into SAR

  if (scope == "global") then
    emit_line ("lda",SAR,"="||addr,,"Lvalue --> SAR")
```

```
  else # (scope == "local")
  {
    emit_line ("movi",OR,addr,,"Lvalue --> SAR")
    emit_line ("add",SAR,FP,OR)
  }
end

procedure emit_Indirect ()
  # emit kbox assembly code to:
  # replace the address in the SAR register
  # with the actual address for the data

  emit_line ("ldr",SAR,SAR,,"indirect / call by var")
end

procedure emit_Rvalue (data_type)
  # emit kbox assembly code to:
  # retrieve the data value found at the memory address
  # currently found in SAR
  # and push the data value onto the top of the stack

  if ((data_type == "INT") |
      (data_type == "STRING") |
      (isPointer(data_type))) then
  {
    emit_line ("ldr",IA,SAR,,"Rvalue --> stack")
    emit_line ("push",IA)
  }
  else if (data_type == "REAL") then
  {
    emit_line ("ldr",FA,SAR,,"Rvalue --> stack")
    emit_line ("push",FA)
  }
  else
  {
    emit_comment ("note: for structured data types, " ||
                  "Rvalue = Lvalue!")
    emit_line ("push",SAR)
  }
end

procedure emit_literal (literal_type,literal_value)
  # emit kbox assembly code to:
  # move a constant value to the top of the stack

  if (literal_type == "INT") then
  {
    emit_line ("movi",IA,literal_value,,
               "int literal constant")
    emit_line ("push",IA)
  }
  else if (literal_type == "REAL") then
  {
```

```
    emit_line ("movi",FA,literal_value,,
               "real literal constant")
    emit_line ("push",FA)
  }
  else if (literal_type == "STRING") then
  {
    emit_line ("movi",SAR,literal_value)
    emit_line ("push",SAR)
  }
  else # (literal_type == "POINTER")
  {
    emit_line ("movi",IA,0,,"NULL pointer")
    emit_line ("push",IA)
  }
end

#-------------------------------------------------------------------#

procedure emit_promote (reg)
  if (reg[1] == "I") then
  {
    dest := "F" || reg[2:0]
    emit_line ("i2f",dest,reg,,"promote")
  }
  else
    stop ("invalid int register (" || reg || ")")
end

procedure emit_demote (reg)
  if (reg[1] == "F") then
  {
    dest := "I" || reg[2:0]
    emit_line ("f2i",dest,reg,,"demote")
  }
  else
    stop ("invalid real register (" || reg || ")")
end

#-------------------------------------------------------------------#

procedure emit_read (data_type)
  # emit kbox assembly code to:
  #   - read value from the keyboard
  #   - store value on the top of the stack
  #   - data type determines fmt

  if (\data_type) then
  {
    if (data_type == "INT") then
    {
      fmt := "INT"
      reg := IA
    }
```

```
    else
    {
      fmt := "FLT"
      reg := FA
    }
    emit_line ("get",fmt,,,"read")
    emit_line ("pop",reg,,,"store input value")
    emit_line ("pop",DAR)
    emit_line ("str",reg,DAR)
  }
  else
    emit_line ("getln",,,,"eol")
end

#-------------------------------------------------------------#

procedure emit_write (data_type)
  # emit kbox assembly code to:
  #   - write value to the monitor
  #   - value is found on the top of the stack
  #   - data type determines fmt

  if (\data_type) then
  {
    if (data_type == "INT") then
      fmt := "INT"
    else if (data_type == "REAL") then
      fmt := "FLT"
    else if (data_type == "STRING") then
      fmt := "STR"
    else if (isPointer(data_type)) then
      fmt := "PTR"
    else
    {
      stop ("invalid write data type (" || data_type || ")")
    }
    emit_line ("put",fmt,,,"write")
  }
  else
    emit_line ("putln",,,,"eol")
end

#-------------------------------------------------------------#

procedure emit_assign (target_type,expr_type,data_size)
  # emit kbox code to assign data value at top of stack
  #   to target location also found attop of stack
  # target_type specifies the data type for the transfer

  emit_line ("nop",,,,"assignment")
  if (isAtomic(global_symbol_table[target_type])) then
    if (target_type == "INT") then
    {
```

```
      if (expr_type == "REAL") then
      {
        emit_line ("pop",FA)
        emit_demote (FA)
      }
      else if (expr_type == "INT") then
        emit_line ("pop",IA)
      else
        stop ("invalid assignment: " ||
              target_type  || " <-> " || expr_type)
      emit_line ("pop",DAR)
      emit_line ("str",IA,DAR)
    }
    else if (target_type == "REAL") then
    {
      if (expr_type == "INT") then
      {
        emit_line ("pop",IA)
        emit_promote (IA)
      }
      else if (expr_type == "REAL") then
        emit_line ("pop",FA)
      else
        stop ("invalid assignment: " ||
              target_type || " <-> " || expr_type)
      emit_line ("pop",DAR)
      emit_line ("str",FA,DAR)
    }
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
  else if (isArray(global_symbol_table[target_type]) |
           isRecord(global_symbol_table[target_type])) then
    if (target_type == expr_type) then
    {
      label_loop := @name_generator
      emit_line ("pop",SAR)
      emit_line ("pop",DAR)
      emit_line ("movi",IR,data_size)
      emit_label (label_loop)
      emit_line ("ldr",IA,SAR)
      emit_line ("str",IA,DAR)
      emit_line ("inc",SAR)
      emit_line ("inc",DAR)
      emit_line ("dec",IR)
      emit_line ("cmp",IR,ZR)
      emit_line ("jgt",label_loop)
    }
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
  else if (isPointer(target_type)) then
    if ((target_type == expr_type) |
```

```
        (expr_type == "POINTER")) then
    {
      emit_line ("pop",IA)
      emit_line ("pop",DAR)
      emit_line ("str",IA,DAR)
    }
    else
      stop ("invalid assignment: " ||
            target_type || " <-> " || expr_type)
  else
    stop ("how on earth did you get here!")
end

#-------------------------------------------------------------#

procedure emit_get_operands (a_type,b_type,c_type)
  # emit kbox code to retrieve a two operands
  #   from the top of the stack
  # b_type and c_type determine the registers for data
  #   and a_type determines any promotion

  if ((c_type == "INT") | isPointer(c_type)) then
    emit_line ("pop",IC,,,"get operands")
  else if (c_type == "REAL") then
    emit_line ("pop",FC,,,"get operands")
  else
    stop ("unrecognized data type (" || c_type || ")")
  if ((b_type == "INT") | isPointer(b_type)) then
    emit_line ("pop",IB)
  else if (b_type == "REAL") then
    emit_line ("pop",FB)
  else
    stop ("unrecognized data type (" || b_type || ")")
  if ((b_type == "INT") & (a_type == "REAL")) then
    emit_promote (IB)
  if ((c_type == "INT") & (a_type == "REAL")) then
    emit_promote (IC)
end

#-------------------------------------------------------------#

procedure emit_compop (op,a_type,b_type,c_type)
  # emit kbox code to compare two data values
  # a_type, b_type, and c_type determine registers to use

  emit_get_operands (a_type,b_type,c_type)
  if ((a_type == "INT") | isPointer(a_type)) then
  {
    inst := "cmp"
    emit_line (inst,IB,IC,,"perform comparison "||op)
  }
  else if (a_type == "REAL") then
  {
```

```
    inst := "fcmp"
    emit_line (inst,FB,FC,,"perform comparison "||op)
  }
  else
    stop ("invalid arithmetic data type ("||a_type||")")
  label_true := @name_generator
  label_false := @name_generator
  label_done := @name_generator
  if (op == "=") then
    emit_line ("jeq",label_true)
  else if (op == "<>") then
    emit_line ("jne",label_true)
  else if (op == ">") then
    emit_line ("jgt",label_true)
  else if (op == "<") then
    emit_line ("jlt",label_true)
  else if (op == ">=") then
    emit_line ("jge",label_true)
  else # (op == "<=")
    emit_line ("jle",label_true)
  emit_label (label_false)
  emit_line ("movi",IA,"0")
  emit_line ("jmp",label_done)
  emit_label (label_true)
  emit_line ("movi",IA,"1")
  emit_label (label_done)
  emit_line ("push",IA)
end

procedure emit_addop (op,a_type,b_type,c_type)
  # emit kbox code to add/subtract two data values
  # a_type, b_type, and c_type determine registers to use

  emit_get_operands (a_type,b_type,c_type)
  if (a_type == "INT") then
  {
    if (op == "+") then
      inst := "add"
    else if (op == "-") then
      inst := "sub"
    else
      stop ("invalid INT addop ("||op||")")
    emit_line (inst,IA,IB,IC,"perform addop "||op)
    emit_line ("push",IA)
  }
  else if (a_type == "REAL") then
  {
    if (op == "+") then
      inst := "fadd"
    else if (op == "-") then
      inst := "fsub"
    else
      stop ("invalid REAL addop ("||op||")")
```

```
    emit_line (inst,FA,FB,FC,"perform addop "||op)
    emit_line ("push",FA)
  }
  else
    stop ("invalid arithmetic data type ("||a_type||")")
end

procedure emit_mulop (op,a_type,b_type,c_type)
  # emit kbox code to multiply/divide two data values
  # a_type, b_type, and c_type determine registers to use

  emit_get_operands (a_type,b_type,c_type)
  if (a_type == "INT") then
  {
    if (op == "*") then
      inst := "mul"
    else if (op == "/") then
      inst := "div"
    else if (op == "%") then
      inst := "mod"
    else
      stop ("invalid INT mulop ("||op||")")
    emit_line (inst,IA,IB,IC,"perform mulop "||op)
    emit_line ("push",IA)
  }
  else if (a_type == "REAL") then
  {
    if (op == "*") then
      inst := "fmul"
    else if (op == "/") then
      inst := "fdiv"
    else
      stop ("invalid REAL mulop ("||op||")")
    emit_line (inst,FA,FB,FC,"perform mulop "||op)
    emit_line ("push",FA)
  }
  else
    stop ("invalid arithmetic data type ("||a_type||")")
end

#-------------------------------------------------------------#

procedure emit_neg (data_type)
  # emit kbox code to negate a single data value

  if (data_type == "INT") then
  {
    emit_line ("pop",IA)
    emit_line ("neg",IA)
    emit_line ("push",IA)
  }
  else if (data_type == "REAL") then
  {
```

```
    emit_line ("pop",FA)
    emit_line ("fneg",FA)
    emit_line ("push",FA)
  }
  else
    stop ("invalid arithmetic data type ("||data_type||")")
end

#------------------------------------------------------------#

procedure emit_and_a (label_false)
  # emit kbox code to perform logical AND on INT values
  # note: implements short-circuit logic,
  #     not simple LAND instruction

  emit_line ("pop",IA,,,"short-circuit and")
  emit_line ("cmp",IA,ZR)
  emit_line ("jeq",label_false)
end

procedure emit_and_b (label_false,label_done)

  emit_line ("pop",IA)
  emit_line ("cmp",IA,ZR)
  emit_line ("jeq",label_false)
  emit_line ("movi",IA,1)
  emit_line ("push",IA)
  emit_line ("jmp",label_done)
  emit_label (label_false)
  emit_line ("mov",IA,ZR)
  emit_line ("push",IA)
  emit_label (label_done)
  emit_line ("nop")
end

procedure emit_or_a (label_true)
  # emit kbox code to perform logical OR on INT values
  # note: but implements short-circuit logic
  #     not simple LOR instruction

  emit_line ("pop",IA,,,"short-circuit or")
  emit_line ("cmp",IA,ZR)
  emit_line ("jne",label_true)
end

procedure emit_or_b (label_true,label_done)

  emit_line ("pop",IA)
  emit_line ("cmp",IA,ZR)
  emit_line ("jne",label_true)
  emit_line ("mov",IA,ZR)
  emit_line ("push",IA)
  emit_line ("jmp",label_done)
```

```
  emit_label (label_true)
  emit_line ("movi",IA,1)
  emit_line ("push",IA)
  emit_label (label_done)
  emit_line ("nop")
end

procedure emit_not (data_type)
  # emit kbox code to perform logical NOT on INT value

  if (data_type == "INT") then
  {
    emit_line ("pop",IA,,"logical not")
    emit_line ("lnot",IA)
    emit_line ("push",IA)
  }
  else
    stop ("invalid logical NOT data type (" ||
          data_type || ")")
end

# ------------------------------------------------------------ #

procedure calc_array_offset (element_size)
  emit_line ("nop",,,,"array offset")
  retrieve_register (IR)
  emit_line ("movi",OR,element_size)
  emit_line ("mul",OR,IR,OR)
  retrieve_register (SAR)
  emit_line ("add",SAR,SAR,OR)
end

procedure calc_record_offset (offset)
  emit_line ("nop",,,,"record offset")
  emit_line ("movi",OR,offset)
  retrieve_register (SAR)
  emit_line ("add",SAR,SAR,OR)
end

# ------------------------------------------------------------ #

procedure emit_empty ()
  emit_line ("nop",,,,"empty statement")
end

procedure emit_goto (lvalue)
  emit_line ("jmp",lvalue,,,"goto statement")
end

# ------------------------------------------------------------ #

procedure emit_do_a ()
  emit_line ("nop",,,,"do statement")
```

```
end

procedure emit_do_b ()
  emit_line ("nop",,,,"end do")
end

# ---------------------------------------------------------- #

procedure emit_if_a ()
  emit_line ("nop",,,,"if statement")
end

procedure emit_if_b ()
  emit_line ("nop",,,,"end if")
end

procedure emit_test_a (label_false,label_done)
  emit_line ("pop",IA,,,"test result")
  emit_line ("cmp",IA,ZR)
  emit_line ("jeq",label_false)
  emit_line ("nop",,,,"then clause")
end

procedure emit_test_b (label_false,label_done)
  emit_line ("jmp",label_done)
  emit_label (label_false)
  emit_line ("nop",,,,"else clause")
end

procedure emit_test_c (label_false,label_done)
  emit_label (label_done)
  emit_line ("nop",,,,"end test")
end

procedure emit_case_a ()
  emit_line ("nop",,,,"case statement")
end

procedure emit_case_b ()
  emit_line ("nop",,,,"end case")
end

procedure emit_case_compare (case_value,label_next)
  emit_line ("movi",IB,case_value,,"case value");
  emit_line ("pop",IA,,,"target value")
  emit_line ("push",IA,,"resave target value")
  emit_line ("cmp",IA,IB)
  emit_line ("jne",label_next)
end

procedure emit_next_case (label_done,label_next)
  if (\label_next) then
  {
```

```
    emit_line ("jmp",label_done)
    emit_label (label_next)
  }
  else
    emit_label (label_done)
end

# ------------------------------------------------------------ #

procedure emit_while_a (label_top)
  emit_label (label_top)
  emit_line ("nop",,,,"while statement")
end

procedure emit_while_b (label_done)
  emit_label (label_done)
  emit_line ("nop",,,,"end while")
end

procedure emit_repeat_a (label_top)
  emit_label (label_top)
  emit_line ("nop",,,,"repeat statement")
end

procedure emit_repeat_b (label_done)
  emit_label (label_done)
  emit_line ("nop",,,,"end repeat")
end

procedure emit_for_a ()
  emit_line ("nop",,,,"for statement")
end

procedure emit_for_b ()
  emit_line ("nop",,,,"end for")
end

procedure emit_for_setup ()
  emit_line ("pop",IB,,,"set up for loop")
  emit_line ("pop",IA)
  emit_line ("pop",DAR)
  emit_line ("str",IA,DAR)
  emit_line ("push",DAR)
  emit_line ("push",IB)
end

procedure emit_for_test (flag,label_top,label_break)
  emit_label (label_top)
  emit_line ("pop",IB,,,"top test for loop")
  emit_line ("pop",SAR)
  emit_line ("ldr",IA,SAR)
  emit_line ("cmp",IA,IB)
  if (flag == "up") then
```

```
    emit_line ("jgt",label_break)
  else
    emit_line ("jlt",label_break)
  emit_line ("push",SAR)
  emit_line ("push",IB)
end

procedure emit_for_increment (flag,label_top,label_next,
                              label_break)
  emit_label (label_next)
  emit_line ("pop",IB,,,"increment_for_loop")
  emit_line ("pop",DAR)
  emit_line ("ldr",IA,DAR)
  if (flag == "up") then
    emit_line ("inc",IA)
  else
    emit_line ("dec",IA)
  emit_line ("str",IA,DAR)
  emit_line ("push",DAR)
  emit_line ("push",IB)
  emit_line ("jmp",label_top)
  emit_label (label_break)
end

# ------------------------------------------------------------ #

procedure emit_call (procedure_name,return_type)
  emit_line ("call",procedure_name);
  no_actual_args :=
    *global_symbol_table[procedure_name].formal_arguments
  emit_line ("movi",OR,no_actual_args)
  emit_line ("add",SP,SP,OR)
  if (\return_type) then
    if (return_type == "REAL") then
      emit_line ("push",FA)
    else
      emit_line ("push",IA)
end

# ------------------------------------------------------------ #
```

21.4 lexeme: basic building blocks

lexeme.icn

```
# ---------------------------------------------------------- #

# lexeme.icn

# read an input file generated by the scanner
# - line number
# - token type
# - token value

# support routines
# - read_lexeme: primary component of this file!
# - display_lexeme: useful for debugging (if necessary)
# - does_match: does token type match a given target

# ---------------------------------------------------------- #

record lexeme (line_no,tok_type,tok_value)

# ---------------------------------------------------------- #

procedure read_lexeme ()
  line := read () | fail
  line ?
  {
    line_no   := tab (upto ('␣'))
    tab (many ('␣'))
    tok_type  := tab (upto ('␣'))
    tab (many ('␣'))
    tok_value := tab (0)
  }
  result := lexeme (line_no,tok_type,tok_value)

  # uncomment the following line to facilitate debugging
  # display_lexeme (result)

  return result
end

procedure display_lexeme (current)
  write (left (current.line_no,7),
         left (current.tok_type,12),
         current.tok_value)
end

# ---------------------------------------------------------- #

procedure does_match (lex,target)
  return (lex.tok_type == target)
end
```

```
procedure must_match (lex,target)
  if (lex.tok_type == target) then
    return
  else
    SyntaxError (current.line_no,target,lex.tok_type)
end

#---------------------------------------------------------- #
```

21.5 error: error handling

error.icn

```
#---------------------------------------------------------- #

# error.icn

# simple error handler ("one and done")
# - syntax error is a mismatch
#     between the current token and what is expected in the grammar
# - semantics error is an inability to implement
#     the desired operation as defined by the language

#---------------------------------------------------------- #

procedure SyntaxError (line_no,expected,found)
  stop ("Syntax Error (line " || line_no || "): " ||
        "expected (" || expected || "), found (" || found || ")")
end

procedure SemanticsError (line_no,description)
  stop ("Semantics Error (line " || line_no || "): " ||
        description)
end

#---------------------------------------------------------- #
```

21.5.1 buildit script

buildit

```
#! /bin/bash

icont kizecompiler \
      lexeme error tables assembler
```

21.6 Sample Programs

21.6.1 Sample 01

The first sample program is a basic array of real numbers, which is to be sorted into increasing order.

simple sorting array of reals

```
program sorting

  types
    vector  :  array [100] of real;

  procedures
    sort (x : vector,size : int);
    swap (var a : real,var b : real);

begin

  procedure main

    variables
      a     :  vector;
      sz    :  int;
      i     :  int;

  begin
    write ("enter number of elements: ");
    readln (sz);
    for i := 0 upto sz-1
    do
      write ("enter real: ");
      readln (a[i]);
    end;
    call sort (a,sz);
    writeln ("the sorted vector is:");
    for i := 0 upto sz-1
      writeln (a[i]);
    writeln ("the end");
  end

  procedure sort

    variables
      i    :  int;
      j    :  int;
      loc  :  int;

  begin
```

```
    for i := 0 upto size-2
    do
      loc := i;
      for j := i+1 upto size-1
        if (x[j] < x[loc])
          then loc := j;
      if (loc <> i)
        then call swap (x[i],x[loc]);
    end;
  end

  procedure swap

    variables

      temp  :  real;

  begin
    temp := a;
    a := b;
    b := temp;
  end

end
```

21.6.2 Sample 02

The second sample program is a compound interest calculation, based on an annual deposit, the annual interest rate, and the number of years to be considered.

compound interest calculation

```
program interest

  constants
    monthsinyear : 12;

  procedures
    compound (amount : real,intrate : real) : real;

begin

  procedure main

    variables
      balance  : real;
      deposit  : real;
      intrate  : real;
      noyears  : int;
      counter  : int;

  begin

    balance := 0.0;
    write ("enter annual deposit: ");
    readln (deposit);
    write ("enter interest rate: ");
    readln (intrate);
    write ("enter number of years to project: ");
    readln (noyears);
    for counter := 0 upto noyears
    do
      balance := balance + deposit;
      balance := compound (balance,intrate);
      writeln ("year ",counter," bank balance: ",balance);
    end;

  end

  procedure compound

    variables
      monthlyrate : real;
      counter     : int;
```

```
  begin

    monthlyrate := intrate / monthsinyear;
    for counter := monthsinyear downto 1
      amount := amount*(1.0+monthlyrate);
    return amount;

  end

end
```

21.6.3 Sample 03

The third sample program is basic statistical analysis for a collection of floating ponit data.

statistical analysis of floating point data

```
program stats

  constants
    maxsize    : 100;
    epsilon    : 0.00001;

  types
    arraytype : array [maxsize] of real;

  procedures
    max        (var a : arraytype,n : int) : real;
    min        (a : arraytype,n : int) : real;
    average    (a : arraytype,n : int) : real;
    variance   (a : arraytype,n : int,mean : real) : real;
    abs        (x : real) : real;
    square     (x : real) : real;
    sqrt       (x : real) : real;

begin

  procedure main

    labels
      1;

    variables
      x          : arraytype;
      n          : int;
      i          : int;
      highx      : real;
      lowx       : real;
      rangex     : real;
      meanx      : real;
      varx       : real;
      stdx       : real;

  begin
    write ("enter the number of items: ");
    readln (n);
    if (n = 0)
      then goto 1;
    writeln ();
    for i := 1 upto n
    do
```

```
    write ("item ",i,": ");
    readln (x[i-1]);
  end;
  writeln ();
  highx := max (x,n);
  lowx := min (x,n);
  rangex := highx - lowx;
  meanx := average (x,n);
  varx := variance (x,n,meanx);
  stdx := sqrt (varx);
  writeln ("the max is:                 ",highx);
  writeln ("the min is:                 ",lowx);
  writeln ("the range is:               ",rangex);
  writeln ("the average is:             ",meanx);
  writeln ("the variance is:            ",varx);
  writeln ("the standard deviation is: ",stdx);
  1;
end

procedure max

  variables
    loc       : int;
    i         : int;

begin
  loc := 0;
  for i := 1 upto n-1
    if (a[i] > a[loc])
      then loc := i;
  return a[loc];
end

procedure min

  variables
    loc       : int;
    i         : int;

begin
  loc := 0;
  for i := 1 upto n-1
    if (a[i] < a[loc])
      then loc := i;
  return a[loc];
end

procedure average

  variables
    sum : real;
    i : int;
```

```
begin
  sum := 0.0;
  for i := 0 upto n-1
    sum := sum + a[i];
  return sum / n;
end

procedure variance

  variables
    sum        : real;
    i          : int;

begin
  sum := 0.0;
  for i := 0 upto n-1
    sum := sum + square(a[i]-mean);
  return sum / n;
end

procedure abs

begin
  if (x >= 0)
    then return x;
    else return -x;
end

procedure square

begin
  return x*x;
end

procedure sqrt

  variables
    guess      : real;
    newguess   : real;

begin
  if (x = 0.0)
    then return x;
  guess := 1.0;
  newguess := 0.5 * (guess + x/guess);
  while (abs(newguess - guess) > epsilon)
  do
    guess := newguess;
    newguess := 0.5 * (guess + x/guess);
  end;
  return newguess;
end
```

```
end
```

21.6.4 Sample 04

The fourth sample program is a collection of recursive recursive functions: factorial, fibonacci, and one whose name I always forget!

recursive functions

```
program recursive

  procedures
    factorial (n : int         ) : int;
    fibonacci (n : int         ) : int;
    strange   (n : int,c : int);

begin

  procedure main

    variables
      i     : int;
      start : int;
      count : int;

  begin
    for i := 1 upto 20
      writeln (i," factorial is: ",factorial (i));
    writeln ();
    for i := 0 upto 20
      writeln (i," fibonacci is: ",fibonacci (i));
    writeln ();
    write ("enter a positive integer: ");
    readln (start);
    count := 1;
    call strange (start,count);
  end

  procedure factorial

  begin
    if (n = 0)
      then return 1;
      else return n * factorial (n-1);
  end

  procedure fibonacci

  begin
    if ((n = 0) or (n = 1))
      then return 1;
      else return fibonacci (n-1) + fibonacci (n-2);
```

```
  end

  procedure strange

  begin
    if (c = 20)
      then return;
    if (n = 1)
      then do
           writeln (n);
           return;
           end;
      else do
           write (n," ");
           if (n % 2 = 0)
             then call strange (n/2,c+1);
             else call strange (3*n + 1,c+1);
           end;
  end

end
```

21.6.5 Sample 05

The fifth sample program implements complex arithmetic – addition, subtraction, multiplication, division, conjugate, and modulus.

complex arithmentic

```
program complex

  constants
    i         : "i";
    prompt    : "enter a complex number!";
    sum       : "the sum is:        ";
    diff      : "the difference is: ";
    prod      : "the product is:    ";
    quot      : "the quotient is:   ";
    conj      : "the conjugate is:  ";
    mods      : "the modulus is:    ";
    epsilon   : 0.00001;

  types
    complex   : record first : real; second : real; end;

  procedures
    get       (x : complex);
    put       (x : complex);
    add       (x : complex,y : complex,z : complex);
    sub       (x : complex,y : complex,z : complex);
    mul       (x : complex,y : complex,z : complex);
    div       (x : complex,y : complex,z : complex);
    conjugate (x : complex,y : complex);
    modulus   (x : complex) : real;
    sqrt      (x : real) : real;
    abs       (x : real) : real;

begin

  procedure main

      variables
        a         : complex;
        b         : complex;
        c         : complex;

  begin
    writeln (prompt);
    call get (a);
    writeln (prompt);
    call get (b);
    call add (a,b,c);
```

```
    write (sum);
    call put (c);
    call sub (a,b,c);
    write (diff);
    call put (c);
    call mul (a,b,c);
    write (prod);
    call put (c);
    call div (a,b,c);
    write (quot);
    call put (c);
    call conjugate (a,c);
    write (conj);
    call put (c);
    call conjugate (b,c);
    write (conj);
    call put (c);
    writeln (mods,modulus(a));
    writeln (mods,modulus(b));
  end

  procedure get

  begin
    write ("enter the real part: ");
    readln (x.first);
    writeln (x.first);
    write ("enter the imaginary part: ");
    readln (x.second);
    writeln (x.second);
  end

  procedure put

  begin
    writeln ("the complex number is (",x.first,") + (",x.second,") ",i)
  end

  procedure add

  begin
    z.first := x.first + y.first;
    z.second := x.second + y.second;
  end

  procedure sub

  begin
    z.first := x.first - y.first;
    z.second := x.second - y.second;
  end

  procedure mul
```

```
begin
  z.first := x.first*y.first - x.second*y.second;
  z.second := x.first*y.second + x.second*y.first;
end

procedure div

  variables
    m          : real;
    scale      : real;
    temp       : complex;

begin
  m := modulus(y);
  if (m > 0.0)
    then do
         scale := 1.0/(m*m);
         call conjugate (y,temp);
         call mul (x,temp,z);
         z.first := scale*z.first;
         z.second := scale*z.second;
         end;
end

procedure conjugate

begin
  y.first := x.first;
  y.second := - x.second;
end

procedure modulus

begin
  return sqrt (x.first*x.first + x.second*x.second);
end

procedure sqrt

  variables
    guess      : real;
    newguess   : real;

begin
  if (x = 0.0)
    then return x;
  guess := 1.0;
  newguess := 0.5 * (guess + x/guess);
  while (abs(newguess - guess) > epsilon)
  do
    guess := newguess;
    newguess := 0.5 * (guess + x/guess);
```

```
    end;
    return newguess;
  end

  procedure abs

  begin
    if (x >= 0)
      then return x;
      else return -x;
  end

end
```

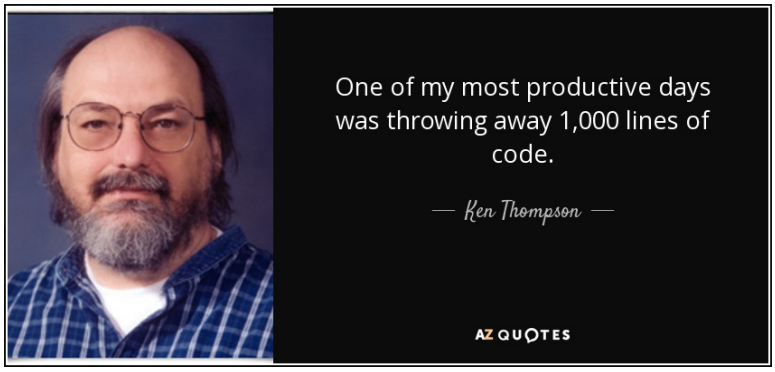
21.6.6 Sample 06

The sixth (and final) sample program implements a simple forward linked list in the **kize** programming language.

```
program linkedlist

    types

      node       : record
                   data : int;
                   point : ^node;
                   end;

    procedures
      initialize (var header : ^node);
      insert (var header : ^node,value : int);
      display (header : ^node);
      reversedisplay (header : ^node);
      total (header : ^node) : int;
      cleanup (var header : ^node);

begin

  procedure main

      variables

        head : ^node;
        n    : int;

  begin
    call initialize (head);
    write ("enter positive integer: ");
    readln (n);
    while (n > 0)
    do
      call insert (head,n);
      write ("enter positive integer: ");
      readln (n);
    end;
    writeln ();
    call display (head);
    writeln ();
    call reversedisplay (head);
    writeln ();
    writeln ("the sum of all the integers is: ",total(head));
  end

  procedure initialize
```

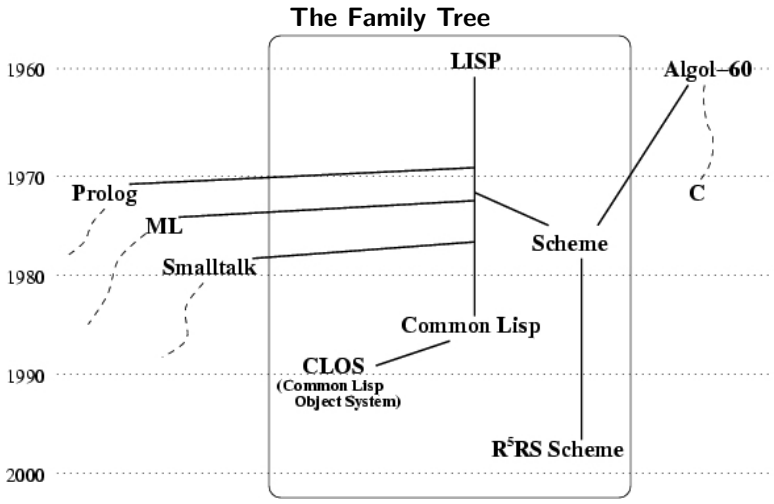
```
begin
  header := null;
end

procedure insert

    variables

      p : ^node;

begin
  p := new node;
  p^.data := value;
  p^.point := header;
  header := p;
end

procedure display

begin
  if (header <> null)
    then do
         writeln (header^.data);
         call display (header^.point);
         end;
end

procedure reversedisplay

begin
  if (header <> null)
    then do
         call reversedisplay (header^.point);
         writeln (header^.data);
         end;
end

procedure total

begin
  if (header = null)
    then return 0;
    else return header^.data + total (header^.point);
end

procedure cleanup

begin
  if (header = null)
    then return;
    else do
         call cleanup (header^.point);
         dispose header;
```

```
            header := null;
            end;
  end

end
```


One of my most productive days was throwing away 1,000 lines of code.
— Ken Thompson —
AZ QUOTES

Chapter 22

Introduction to Functional Programming



22.1 Historical Roots

The infancy of computation had obvious roots in theoretical mathematics.

Hilbert's Problems, posed at the turn of the twentieth century, spurred on a burst of mathematical research particularly in the areas of logic and the foundations of mathematics. By the early 1930s mathematicians had focused their attention on the famous Entscheidungsproblem. Alan Turing and Alonzo Church, working independently, both proved the Entscheidungsproblem to be false and by doing so became two of the founding fathers for computer science.

Alan Turing's solution to the problem was based on simple mechanical machines, called Turing Machines. He subsequently demonstrated that such a machine would then be capable of simulating the behavior of any specific Turing Machine. He had conceived of a Universal Turing Machine – which is the theoretical precursor for the digital computer.

Alonzo Church, on the other hand, focused on how mathematical functions work – their essential elements. Stripped of all non-essential elements, Church considered functions to be two parts: first, a variable representing the item to be manipulated; and second, a rule for manipulating that item. For example, the standard identity function f(x) = x is represented in Church's notation as

$$\lambda x.x$$

Using this minimalist notation, referred to as the **lambda calculus**, Church was also able to develop a proof for the Entscheidungsproblem.

Although the two techniques seem on face value miles apart, Turing proved in his seminal paper on computation that, in fact, the two techniques used to solve Hilbert's problem were equivalent to one another. Interestingly, Alan Turing subsequently studied under Alonzo Church at Princeton where he received his doctorate.

22.1.1 John McCarthy and LISP

John McCarthy is a legend in the field of computer science, particularly in artificial intelligence (AI). Primarily known as the creator of one of the longest-lived computer languages in use —LISP (in 1958), McCarthy was also one of the first people interested in AI and coined that particular term in 1955. In the late fifties and early sixties, he also developed the concept of timesharing. He is recognized for his substantial contribution to the theory of computation and knowledge representation.

In 1958, McCarthy specified LISP the second-oldest high-level programming language still in widespread use today (only FORTRAN is older, by one year). Both languages have evolved over the past half century and a number of enhancements and variations have been suggested and incorporated. Today, the most widely known general-purpose LISP dialects are Common Lisp, Scheme, and Clojure.

Originally created as a practical mathematical notation for computer programs, LISP is based on the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for AI research. As one of the earliest programming languages, LISP pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, and the self-hosting compiler.

In John McCarthy's own words:

> *As a programming language, LISPis characterized by the following ideas:*
>
> - *computing with symbolic expressions rather than numbers,*
> - *representation of symbolic expressions and other information by list structure in the memory of a computer,*
> - *representation of information in external media mostly by multi-level lists and sometimes by S-expressions,*
> - *a small set of selector and constructor operations expressed as functions,*

- *composition of functions as a tool for forming more complex functions,*
- *the use of conditional expressions for getting branching into function definitions,*
- *the recursive use of conditional expressions as a sufficient tool for building computable functions,*
- *the use of λ-expressions for naming functions,*
- *the representation of LISP programs as LISP data,*
- *the conditional expression interpretation of Boolean connectives,*
- *the LISP function eval that serves both as a formal definition of the language and as an interpreter,*
- *and garbage collection as a means of handling the erasure problem.*

LISP statements are also used as a command language when LISP is used in a time-sharing environment.

Some of these ideas were taken from other languages, but most were new.

Towards the end of the initial period, it became clear that this combination of ideas made an elegant mathematical system as well as a practical programming language. Then mathematical neatness became a goal and led to pruning some features from the core of the language. This was partly motivated by esthetic reasons and partly by the belief that it would be easier to devise techniques for proving programs correct if the semantics were compact and without exceptions. The results of [Cartright and McCarthy during the late 1970s], which show that LISP programs can be interpreted as sentences and schemata of first order logic, provide new confirmation of the original intuition that logical neatness would pay off.

22.1.2 Dialects

The word LISP is not really a designation for a specific programming language, but rather a generic reference to a family of programming languages that trace their lineage back to John McCarthy's original vision of LISP.

Common Lisp and MIT Scheme are two branches within this LISP family tree that have large followings and very specific standards regarding syntax and implementation.

Lisp implementations include Steel Bank Common Lisp (SBCL), Clozure CL, CLISP, and LispWorks. **Lisp** even has an object-oriented version Common Lisp Object System (CLOS).

Scheme implementations include MIT/GNU Scheme and Racket. Both **Scheme** implementations are freely available.

In essence, **Scheme** is a preorder representation of an operation. That is, the operator is identified first immediately followed in order by a list of its arguments.

In **C++** or in **Java** we might write the following code:

add (1 , 2 , 3 , 4 , 5)

In **Scheme** (or **Lisp**) we would write the following instead:

(add 1 2 3 4 5)

At first glance, everything looks pretty trivial! We move the parentheses to the outside and we throw away the commas! But recall in actual argument lists within a procedural language, the number, the type, and the order is vitally important. In the LISP family, argument list size and types are not an issue!

Here is another, only slightly more complicated example:

(add (mul 2 3 4) (sub 10 5) 3 7)

That is basically LISP syntax in a nutshell! You can now read just about any **Lisp** or **Scheme** program and understand what it is essentially trying to accomplish.

Next step is to better understand exactly what it is doing and how it is doing it.

22.2 Scheme Building Blocks

For this text I have chosen to focus on the more modern **scheme** programming language than the more traditional **lisp** programming language.

At this juncture I highly recommend that you obtain a **scheme** interpreter for your computer system. There are many very good open source versions easily available – among them I would suggest either **Dr. Racket** or **GNU MIT-Scheme**.

Common Lisp is also readily available. But it is important to remember that although **lisp** and **scheme** appear in the same family tree, the two languages are based upon different definitions of scope. As a result, as we move deeper into the syntax and semantics for **scheme** it will eventually part ways with **lisp** and will create some confusion.

So, pick either **scheme** or **lisp**, preferably **scheme**. But, do not play with both at the same time!

In the words of the Okinawan karate master Mister Miyagi:

walk on road ...
walk right side, safe ...
walk left side, safe ...
walk middle, ...
sooner or later, ...
squished like grape!

So let us begin ...

atoms

These are the most basic of the building blocks that we work with in **scheme**. They come in essentially two distinct flavor: literal data and symbolic names.

Literal data types include the following:

- integer: 123, -45, 542
- real: 123.45, -23 54, 11.47e2, -15.6,e-3, 0E0
- character: #\a, #\b, #\A, #\5

- string: "paul" "" "x" "professor"
- boolean: #t and #f
- symbol: x, x23, interest-rate, list− >vector, +, >=

Integers and reals may begin with an optional plus or minus sign.

Single character values must be preceded by the two character sequence #\.
String values begin and end with a double quote (").

There is no escape sequence for ASCII characters (e.g., \ in C or C++).

Only two boolean literal values #t (true) and #f (false);

Symbols are sequences of characters comprised of the following: any ASCII character, except for white space (blank, tab, carriage return) and these special characters ( ) { } [ ] . , ; ′ " ‘| # . Many items that are operators in other programming languages become symbols in Scheme.

I personally prefer to consider an operator, like +, to represent the symbol **add**. During lexical analysis, when reading an input stream, I will actually make this conversion explicit.

dotted pairs

The first data structure in LISP family of languages is the concept of a **dotted pair**. A dotted pair takes two elements from within the language and combines them using the ***cons*** procedure.

The following **scheme** statement

(***cons*** 123 #t)

will generate a dotted pair which will appear (when printed) as follows

[123 . #t]

It is important to note that ***cons*** can be used to combine *any* two Scheme objects into a dotted pair and *not just* two atoms! ***cons*** is essential to the two data structures which immediately follow.

lists

Lists are the cornerstone for the LISP family of languages. Its very name is derived from the phrase **LIS**t **P**rocessing.

In its simplest incarnation, a list is a sequence of items enclosed between parentheses.

(1 2 3 4 5 6 7)

However, this representation (for easy display) must be converted to an internal representation for storage and/or manipulation. The dotted pair constructor ***cons*** is used repeatedly.

(***cons*** 1 (***cons*** 2 (***cons*** 3 (***cons*** 4 (***cons*** 5 (***cons*** 6 (***cons*** 7 ***nil***)))))))

A list is represented as a sequence of *nested* dotted pairs! The first component of the dotted pair is the first item in the list; the second component of the dotted pair is the *rest of the list*!

The last item in the list is also a dotted pair! Notice above the use of a special symbol **nil**, which represents the *empty list* ( ).

It is important to remember the relationship and the distinction between lists and dotted pairs! A list is always comprised of nested dotted pairs. However, dotted pairs do **not** mean it is a list!

(***cons*** 123 #t) is just a *dotted pair* of two atoms;
(***cons*** 123 (***cons*** #t ***nil***)) is a *list* of two atoms.

There is a significant difference!

A list, in its most general form, is a sequence of either atoms or other lists separated by blanks and enclosed within parentheses.

Other examples of lists include:

(1 2 3 4)
(bob ted carol alice)
(a (b c) (d ( ) (e f)))
( )
(123 () #f (#t #\x "bob") ((x y z) "alice"))

Note that lists may be elements of lists. This nesting may be arbitrarily deep and allows us to create symbol structures of any desired form and complexity.

symbolic-expressions

Symbolic-expressions (or **s-expressions**) are a close relative to lists!

An ***s-expression*** may be an atom, or a list in the previous sense, or a list of elements which are, in turn, either atoms or s-expressions.

The following would be examples of s-expressions:

123
"alice"
bob
(add (mul 2 3 4) (sub 11 7))
(define circumference (* pi radius radius))
(cdr (cons 1 (cons 2 (cons 3 ()))))
((figure-out-function item1 item2) some-data 27 (add 3 5))

We will shortly be discussing the scheme interpreter, comprised of a fundamental read-eval-print loop.

The first component in this triumvirate is a reader which must convert an input stream of characters into an internal representation that can be handled by the second component, an evaluator!

The third component is a printer which must convert results obtained from the evaluator into a human-readable form.

Symbolic expressions are essentially a description of the desired human-readable form used for input and most output. Dotted pairs are the internal representation which the interpreter actually manipulates; lists are much easier on the eyes to understand!

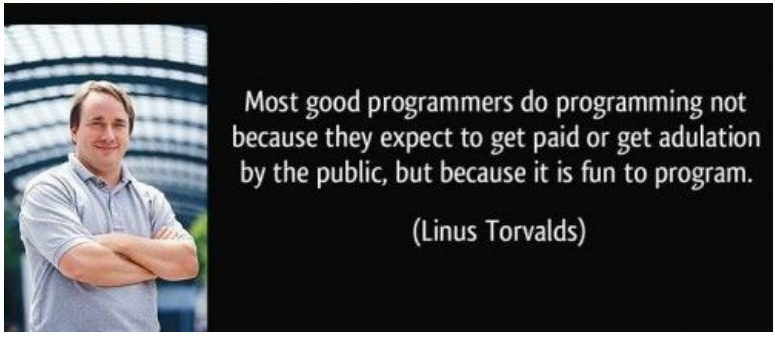
comments

Comments are also an essential component within any programming language to clarify what the author is trying to accomplish within his or her source code file. The semicolon (;) serves as the comment indicator in **scheme**.

A frequently used convention is that one semicolon is used for a short comment following a single line of code, two semicolons for a comment within a function but on its own line, and three semicolons for an introductory or global comment.

Below is an illustrative example:

```
;;; this is an artificial illustration of commenting
;;; as a result, it is trivial and obvious

(define square (lambda (x)

;; the square function
;; input single value x
;; result single value (* x x)

(* x x) ))        ; wow, i squared it!
```


Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.
(Linus Torvalds)

Chapter 23

Core Implementation

Honey! Have you noticed that ...

... our table seems a foot taller!

In this chapter we begin to implement the most basic elements of the **scheme** programming language. We know very little regarding the language except for its basic building blocks and most rudimentary functions. The focus of this chapter is to extend and this rudimentary foundation.

We want to implement these core items in a programming language of our choice. We want to simulate the behavior of **scheme** at the lowest level, using the tools available within our chosen language.

I have own again chosen to work with the **Icon** programming language. I certainly like its flexibility with lexical analysis, which will be essential in implementing a **scheme reader** for the input of information to our **skeme** interpreter. I have chosen to call the minimal version we implement **skeme** to continue my proclivity for using the letter **K** – **kbox**, **kcode**, **kize**, and now **skeme**!

A second reason is that, by now, the reader should have some familiarity with the **Icon** language and the advantage of a typeless language where a variable is capable of holding any type of value. This is ideal for a LISP-like language.

And a third reason might be **Icon**'s facility to implement and work with list structures – an obvious necessity for a LISP-like language.

So, let us begin this new project by taking what little information we do know regarding **scheme** and implementing these features using **Icon** procedures.

Remember, this core implementation will be an **Icon** source file, ***core.icn***, simulating **scheme** building blocks. Core procedures in this source file will have an underscore character (_) prepended to their **scheme** name in order to highlight the fact that these are implemented at the lowest level. As we move to higher levels in our implementation, our **Icon** source files will assume a more **scheme**-like flavor.

Note: The artwork for this chapter is the cover art for **Structure and Interpretation of Computer Programs**, by Abelson and Sussman (2nd ed), MIT Press. It is a very good book, with a very unique cover!

23.1 Data Types and Symbols

We start with how to represent the various types of **atoms** within our **skeme** interpreter.

Consider the various data types available.

integers

Integers are simple regular-expressions that we have previously discussed. Our upcoming **skeme read** function will process the input stream and capture an ASCII string representing the integer value. The final step in the process will be to *recast* the **Icon** string as an **Icon** *integer* value.

return integer (result)

reals

Reals are also simple regular-expressions that we have also previously discussed. Our upcoming **skeme read** function will process the input stream and capture an ASCII string representing the real value. The final step in the process will be to *recast* the **Icon** string as an **Icon** *real* value.

return real (result)

characters

A single ASCII character is represented in **Scheme** as the following three-character sequence $\#\backslash < char >$.This is a trivial regular-expression. It will be easy to recognize in our **skeme read** function.

The internal representation is to ignore the first two characters and simply return an **Icon** character set (*cset*) literal.

return '$< char >$'

strings

Icon has a very useful lexical analysis feature – the *balance* function, which simplifies locating balanced expressions within a string. Balance can have several meanings: balanced parentheses, balanced brackets, or in this specific case balanced (double) quote marks!

The resulting string of characters is returned an **Icon** *string*.

return result

booleans

Remember that a boolean value is represented in **Scheme** as a two-character sequence #t for TRUE and #f for FALSE. These are both trivial regular-expressions and easy to recognize in our **skeme read** function.

However, instead of casting it into an existing **Icon** data type, the final step is to return an **Icon** *record* ***boolean*** containing the appropriate string value as its contents.

return boolean(result)

symbols

All of the previous items are considered explicit data values in **Scheme**. They are often referred to as ***self-evaluating atoms*** because they identify exactly what they represent.

The final category of atoms is that of a symbol, very much like the concept of a variable in a procedural language. Symbols can be subsequently defined to represent something specific.

But, for now, we only need to know how to recognize them. Symbols may be any sequence comprised of **any** ASCII characters, excluding white space and ten specific taboo characters. The **taboo ten** will be defined a bit later!

Symbols are trivial regular-expressions and easy to recognize. As with boolean values, the final step is to return an **Icon** *record* ***symbol*** containing the appropriate string value as its contents.

return symbol(result)

These descriptions and definitions are contained in the documentation for the source file **core.icn** which may be read in **Chapter 29: The End Result**.

The implementation of our **skeme reader** will be found in the next chapter. The source file **repl.icn** will contain the details of the **skeme reader** and **skeme printer** which serve repectively as the front-end lexical analyzer for the input stream and as the back-end display for resulting values.

23.2 Dotted Pairs and Lists

Right here, right now. We have to make a critical decision which will impact everything we do subsequently. How do we choose to represent non-atomic structures in our **skeme** interpreter?

We have at least two basic techniques available in **Icon**:

- record structure: a dotted pair could be represented as an **Icon** record ***dotted_pair*** (first , second)
- list structure: a dotted pair could be represented as an **Icon** list
  [first , second]

The first option would closely resemble the path taken by the original implementors of **lisp**. Lists would then be represented by nested sequences of dotted pairs as we have previously discussed.

The second option would take advantage of very useful **Icon** programming language features.

I have chosen to pursue the second alternative in this **skeme** interpreter. However, even as I type this paragraph I see a lot of merit in first alternative. I consider my choice the easier and less confusing of the two; but while the programming might be a bit easier, the first alternative provided me with a lot of valuable insights into the problems early programmers had to contend with.

I believe my choice will make our work ahead a bit easier, but only time will tell. In the words of the poet Robert Frost:

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I . . .
I took the one less traveled by,
And that has made all the difference.

dotted pairs

Dotted pairs will be one of the underlying basic building blocks for **skeme** interpreter.

Dotted pairs combine two **skeme atoms** into a new object, which will be denoted

$$\text{dotted\_pair (item}_1\text{ , item}_2\text{)}$$

Dotted pairs are built using the constructor primitive **cons**.

$$\text{(\textbf{cons} item}_1\text{ item}_2\text{)} \Rightarrow \text{dotted\_pair (item}_1\text{ , item}_2\text{)}$$

The **cons** primitive probably should have been called **pair** and the next two building block components called **first** and **rest** respectively. However, for historical and technical reasons, these basic elements have been referred to as: ***cons*** for pair, ***car*** for first item, and ***cdr*** for the second item (or "the rest").

Both FORTRAN and LISP were implemented on the IBM 704. The IBM 704, introduced by IBM in 1954, was the first mass-produced computer with floating-point arithmetic hardware. Two registers on the IBM 704 were used to implement dotted pairs and more general lists:

- CAR: the **contents address register**
- CDR: the **contents decrement register**

These register names ***stuck*** and became integral to the LISP-like languages!

Personally, I prefer **first** and **rest** for the names of the two building building block components, but the synonyms **car** and **cdr** are so engrained in the **lisp**-psyche that skeme will actually make all four primitives available even though unnecessary.

lists

Lists will be the second of the underlying basic building blocks for **skeme**. Lists are simply sequences of **skeme** atoms and lists. A typical list might typically look like:

(a (b c ( )) d (e (f g)))

Lists combine one **skeme atom** and one **skeme list** into a new object, which will be denoted as a single extended list

$$\text{(item}_1\text{ list}_2\text{)}$$

Lists are built using the constructor primitive **cons**, as were dotted pairs!

(**cons** item_1 list_2) $\Rightarrow$ [item_1] ||| list_2

Lists, in the original LISP implementation, would be represented internally as nested dotted pairs. For example, consider the following input stream:

(a (b c ( )) d (e (f g)))

The internal representation for this list would be the following nested sequence of dotted pairs:

dotted_pair (a , dotted_pair (dotted_pair (b , (dotted_pair (c ,dotted_pair (nil , nil)))) , dotted_pair (d , dotted_pair (dotted_pair (e , dotted_pair (dotted_pair (f , (dotted_pair (g , nil)) , nil)))))))

Please note: the **empty list** is represented by the **Icon** list [ ] and it is the only object in the **skeme** interpreter which is considered to be ***both*** an atom ***and*** a list.

other important combinations

What I call a ***list*** in **skeme** is what LISP would more precisely refer to as a ***proper list***.

A proper list in LISP is a sequence of nested dotted pairs where each item is either an atom or a proper list and the *last item* is the empty list (also referred to as **nil**).

LISP designers originally conceived of several useful data structures that could be built based solely on the **cons** primitive and dotted pairs. **lisp** advocates seem to prefer working primarily with dotted pairs as the basic data structure; **scheme** advocates seem to prefer working primarily with lists and apologize for the confusion caused by dotted pairs.

Regardless of your personal preference, you should be aware of the following two special categories.

An **s-expression** is basically the collection of all atoms and proper lists! An s-expression is the fundamental template for input in LISP in order for that expression to be evaluated.

In order to evaluate something in LISP, it must be one of the following:

- a self-evaluating atom
- a symbol which has been previously **define**d
- a proper list
  - the **car** (first) of the list specifies a function
  - the **cdr** (rest) of the list specifies the arguments

An **association list** starts out being an ordinary list but its elements, instead of being atoms or proper lists, are instead dotted pairs.

A simple illustrative example would be:

(("anne",23) ("bob",32) ("carl",15) ("deb",27) ("emil",45))

Association lists allow for basic tables: first item in the pair is typically a **key** value, second item in the pair is the **data** associated with that key. In the example above, we have a table of name and age pairs!

23.3 Los Tres Amigos

So now, let us actually do something concrete! We implement the fundamental three instructions in **lisp** and **scheme**, ***the three amigos***: cons, car (or first), and cdr (or rest)

cons

Given two objects, return the result from combining the two.

If the second item is a proper list, then cons returns an extended list with the first item prepended to the second.
I the second item is not a proper list, then cons returns a true dotted pair.

los tres amigos

```
# pair            has very specific (limited) role in skeme
#                 is NOT fundamental constructor for lists!
#                 may be used for an association list
#                    i.e., a list of actual pairs!
#                    [a . b]

record            dotted_pair (first,rest)

# list            represented as an icon list
#                 in Lisp (a b c d e f) it would be
#                    [a . [b . [c . [d . [e . [f . []]]]]]]

procedure _cons (item1,item2)
# if item2 is proper list,
#   then cons returns proper list
#      with item1 prepended to the front of item2

  _listp(item2) & return ([item1]|||item2)

# if item 2 is not proper list,
#   then cons returns true dotted pair

  return dotted_pair(item1,item2)
end
```

car / first

Given a dotted pair, return the first item in the dotted pair.

cdr / rest

Given a dotted pair, return the second item in the dotted pair.

los tres amigos

```
procedure _first (item)
# at this point in developing the interpreter
# i prefer to use the name "first" rather than "car"
# we will provide the alternate name shortly
  _atomp(item) & fail
  _listp(item) & (*item > 0) & return item[1]
  _pairp(item) & return item.first
end

procedure _rest (item)
# at this point in developing the interpreter
# i prefer to use the name "rest" rather than "cdr"
# we will provide the alternate name shortly

  _atomp(item) & fail
  _listp(item) & (*item > 0) & return item[2:0]
  _pairp(item) & return item.rest
end
```

Actually, about the only interesting aspect to the topics found on this page is the proper pronunciation of these procedures! **cons** and **car** are no problem! But, what is the proper pronunciation for something with no vowels?

- candidate number one: k-uh-der, like "udder"
- candidate number two: k-ould-der, like "could"

Oh, no! I feel a song coming on!

You like potato and I like potahto
You like k-ould-der and I like k-uh-der
Potato, potahto, k-ould-der, k-uh-der
Let's call the whole thing off!

With all possible apologies to the Gershwin brothers, let us move on to another quirky topic!

LISP-like languages typically implement a variety of **car** and **cdr** combinations to pick out specific elements from within a list.

For example,

The second item in a list would normally be **(car (cdr x))**
but is commonly shortened to **(cadr x)**.

Similarly, the third item in a list x would normally be **(car (cdr(cdr x)))**
but is commonly shortened to **(caddr x)**

Believe it or not, there are 28 different such additional combinations up to a nesting level of four!

CAR/CDR Pronunciation Guide

Function	Pronunciation	Alternate Name
CAR	*kar*	FIRST
CDR	*cou-der*	REST
CAAR	*ka-ar*	
CADR	*kae-der*	SECOND
CDAR	*cou-dar*	
CDDR	*cou-dih-der*	
CAAAR	*ka-a-ar*	
CAADR	*ka-ae-der*	
CADAR	*ka-dar*	
CADDR	*ka-dih-der*	THIRD
CDAAR	*cou-da-ar*	
CDADR	*cou-dae-der*	
CDDAR	*cou-dih-dar*	
CDDDR	*cou-did-dih-der*	
CADDDR	*ka-dih-dih-der*	FOURTH
and so on		

23.4 Primitives and Predicates

Primitives and Predicates – that sounds a bit like **Dungeons and Dragons** or **Chutes and Ladders**!

But these words actually refer to elements found within **lisp** and **scheme**. Specifically, they refer to functions that can be applied to objects in the language.

primitives

A primitive is a low-level function which is normally implemented very early on in the development of an interpreter. A primitive might be considered an essential component that comes with the language.

Search the Internet and you will find many references to the primitive elements that comprise **lisp** and **scheme**. However, many of them will suggest that a minimal collection of about a dozen or so are sufficient to build all the rest! The ones that follow in the remainder of this chapter are the ones that I have chosen to implement for simplicity and convenience. They will also be the components that are used to further build and expand the **skeme** language.

predicates

Predicates are simply boolean primitives! In **lisp**, they should return **nil** for FALSE and anything else for TRUE! In **scheme**, they should return **#t** for TRUE and **#f** for FALSE. But right now, we are implementing basic tools within **Icon** and **Icon** allows for success or failure by its procedures.

With this in mind, we implement our low-level predicates written in **Icon** as procedures which either return (success) or fail. As we continue implementing our **skeme** interpreter, we will redefine these predicates to return the proper boolean values **#t** and **#f** expected from **scheme**.

One concluding comment regarding names for predicates:

- **lisp** predicates always end with the ASCII character "p"
- **scheme** predicates always end with the ASCII character "?"
- our low-level predicates will begin with the ASCII underscore character ("_") and will end with the ASCII character "p"

23.5 core.icn

Please refer to **Chapter 29: The End Result** for my **Icon** source file **core.icn** which implements all the basic elements we have discussed in this chapter. We do not have any main procedure or driver code to test out the accuracy and correctness of the coding. That will come shortly in the following chapter.

However, the successful compilation of **core.icn** is the first good sign that we have seriously begun the project. Once we have implemented the **skeme reader** and the **skeme printer**, we will have sufficient components to test our code for reading the input stream, building the internal representation, and displaying the results.

A brief overview of the file **core.icn** would reveal that the core implementation focuses on the building block elements found in **scheme**. Each of the basic types is represented together with a predicate to recognize that building block: The slight variations in the names are the result of preferences and limitations for identifiers in the various languages.

Lisp prefers all boolean functions end with the letter "p"; **scheme** prefers they end with a question mark "?".

type	scheme name	lisp name	core implementation
integer	integer?	integerp	_integerp
real	real?	realp	_realp
number	number?	numberp	_numberp
boolean	boolean?	booleanp	_booleanp
char	char?	charp	_charp
string	string?	stringp	_stringp
symbol	symbol?	symbolp	_symbolp
atom	atom?	atomp	_atomp
null	null?	nullp	_nullp
dotted_pair	pair?	pairp	_pairp
list	list?	listp	_listp
alist	alist?	alistp	_alistp
primitive	primitive?	primitivep	_primitivep
special_form	special-form?	special-formp	_special_formp
lambda_form	lambda-form?	lambda-formp	_lambda_formp
macro_form	macro-form?	macro-formp	_macro_formp
procedure	procedure?	procedurep	_procedurep
promise	promise?	promisep	_promisep

The contents of the source code **core.icn** should be very readable and understandable, especially with my explanatory comments of the obvious!

At the end of the source file I have included a handful of primitives which are also germane to the concepts we have been discussing.

Length is specific to proper list structures; **length** returns the number of elements in the proper list.

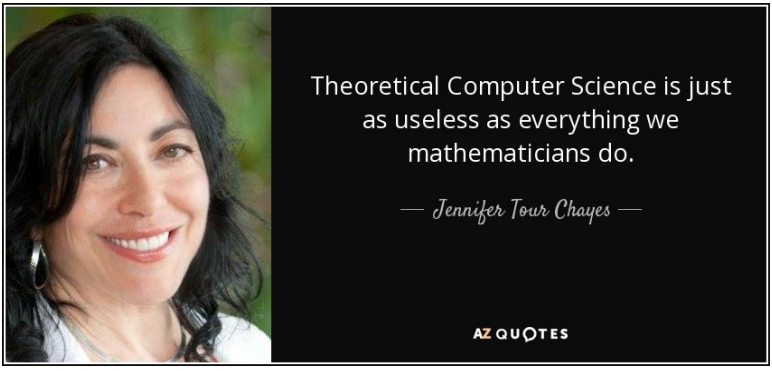
eq?, **eqv?**, and **equal?** are three predicates in the original **lisp** that have continued on. The first item **eq?** shall be last in our brief discussion here!

eqv? tests two ***atoms*** to see if they are equal in value – i.e., the same data type and with the same value.

equal? extends the test for equality to ***lists*** – for dotted pairs, are the cars equal and are the cdrs equal. We must recursively compare, in order, all the components for equality, ultimately using **eqv?** on the atoms that comprise the structure.

eq? is a more restrictive form of equality. Not only does it say that two atoms have the same value, it is really saying that the two atoms are found at the same storage location and that is the reason they have the same value!

We will not be implementing **eq?** in our **skeme** interpreter for two reasons. First, **Icon** does not provide for testing such a restrictive version of equality; and second, **eqv?** is probably sufficient in most cases.


Theoretical Computer Science is just as useless as everything we mathematicians do.
— Jennifer Tour Chayes —
AZ QUOTES

Chapter 24

Read-Eval-Print Loop

This chapter initiates the first major steps to implementing our **skeme** interpreter. The key elements in just about every simulator is some form of fetch-decode-execute cycle. We saw that in machine level language and assembly language (**kcode**). Even with procedural languages there is a read-recognize-translate cycle. Once again a similar triple appears before us.

For LISP-like languages, the sequence is very straight forward. Initially, we need a **reader** which processes an input stream of ASCII characters and converts them into the building blocks and structures which the language can process. Lastly, we need a **printer** which will convert results back to a form which is fathomable by humans. In the middle, is the **evaluator** which has the unenviable task of determining what to do and then actually do it.

We postpone discussion of the **evaluator** until the very end of this chapter. It is much more complicated and detailed than either of the other two elements presented here. It is also absolutely essential to understand in order to comprehend how LISP-like languages actually work.

24.1 REPL

The **skeme** main driver is a basic read-eval-print infinite loop. The source file **skeme.icn**, which may be found in **Chapter 29: The End Result**, is relatively short and fairly simple to understand.

Key global symbols need to be defined, the main driver needs to be implemented in **Icon** coding, and a minimal error handling mechanism needs to be implemented. I really do not think any further explanation is necessary!

Read-Eval-Print Loop

```
# _gensym is a symbol generator
global _gensym

# global variables for scheme REPL elements
global skeme_input
global input_source
global quasiquote_flag

# icon csets for lexical analysis of skeme_input
global white_space
global taboos
global identifier_chars
global relops
global arithops

# icon csets for identifying specific ASCII symbols
global single_quote
global double_quote
global quasi_quote
global back_slash
global comma
global hash
global left_paren
global right_paren
global period
global eol

procedure main ()
# main procedure for skeme: basic read-eval-print-loop

   _gensym := create ("!S"||seq())
   initialize_symbols()
   initialize_environment()
   write("welcome to skeme")
   write("a scheme interpreter written in icon/unicon")
```

```
  load(["init.scm"])
  while (1) do
    repl()
end

procedure initialize_symbols ()
# initialize global variables used across the source files

  input_source          := &input
  quasiquote_flag       := &null

  white_space           := ' \t'
  taboos                := '()[]{}.,;\"\'`|#'
  identifier_chars      := &ascii--taboos--white_space
  relops                := '=><'
  arithops              := '+-*/'

  single_quote          := '\''         # quote
  double_quote          := '\"'
  quasi_quote           := '`'          # quasiquote
  back_slash            := '\\'
  comma                 := ','          # unquote
  splice                := '@'          # unquote splice
  hash                  := '#'
  eol                   := ';'          # skeme eol marker
  left_paren            := '('
  right_paren           := ')'
  period                := '.'
end

procedure repl ()
# the fundamenta read-eval-print loop in any interpreter
# comparable to fetch-decode-execute in machine language

#     _read  is implemented in source file translator.icn
#     _print is implemented in source file translator.icn
#     _eval  is implemented in source file core.icn

  _print(_eval(_read()))
  write()
end

procedure Error (message)
# this is a trivial error handler
# just crash and burn with a simple message

  stop(message)
end
```

24.2 Read

The **skeme reader** does basically two tasks: read single atoms and reads lists. However, single atoms may come in various flavors – integers, reals, characters, strings, boolean values, and symbols. Lists have to be converted as well from parenthesized expressions into **Icon** lists.

As we encountered previously, scanning (lexical analysis) requires processing simple regular expressions to recognize the building block items to pass forward in the process. This very focused processing of a variety of specific patterns implies that a large number of pattern matching procedures must be incorporated in the **Icon** source code for our **skeme reader**.

The **skeme reader** may be found in its entirety in the file **translator.icn** in **Chapter 29: The End Result**.

overview of scheme reader

```
procedure _read ()
# processes input stream of skeme items (s-expressions)
# return a valid skeme internal representation


procedure read_item ()
# read_item determines the category of item to be read
# an atom or a list


procedure read_atom ()
# read the value of a skeme atom
# and convert it to its internal representation


procedure read_list ()
# read the value of a skeme list
# and convert it to its internal representation
```

24.2.1 skeme reader support

However, two features implemented within the reader requires a more detailed explanation.

Two procedures, ***initialize_stream*** and ***append_stream*** facilitate the input process by making the input stream appear to be one long sequence of ASCII characters. If expected input seems to end prematurely, then the user will be prompted for additional input. If extraneous symbols are found after an s-expression, then they will simply be ignored.

However, limiting input to only terminal input (standard input) would be a severe restriction. Having **lisp** or **scheme** code available in a data file which can subsequently be loaded at any time into the interpreter environment can be of great assistance.

Such a such flexibility is enabled within the two procedures identified above. They support reading an input stream from either **stdin** or an input file. We will discuss how to change the input stream from **stdin** to a data file later on, when we discuss the **scheme** primitive **load**.

skeme reader support

```
procedure initialize_stream (prompt)
# input_source initially is stdin
# however, we may redirect reading input from a file

   if (input_source === &input) then
   {
      writes("skeme> ")
      (line := read()) | fail
   }
   else
      (line := read(input_source)) | fail
   skeme_input :=
      reverse(trim(reverse(trim(line),white_space)),
                   white_space) || eol
   if (skeme_input ? any(eol)) then
      append_stream() | fail
   return
end

procedure append_stream ()
# input_source initially is stdin
# however, we may redirect reading input from a file
```

```
   if (input_source === &input) then
   {
      writes("more>  ")
      (line := read()) | fail
   }
   else
      (line := read(input_source)) | fail
   skeme_input :=
      reverse(trim(reverse(trim(line),white_space)),
              white_space) || eol
   if (skeme_input ? any(eol)) then
      append_stream() | fail
   return
end
```

24.2.2 reader tokens

This part of the **scheme reader** focuses on lexical analysis. **Icon** is a great programming language for this task. The following are a list of procedures which are intended to perform the lexical analysis:

- **read_hash**: the ASCII hashtag character (#) begins either **true** (#t) or **false** (#f) boolean values or a single character (\#x) value
- **read_string**: the ASCII double-quote character (") begins a string
  - the next double-quote character terminates the string
  - no escape sequences are permitted between
- **read_quote**: the ASCII single-quote character (') is an abbreviation for the **quote** symbol
- **read_quasiquote**: the ASCII back-quote character (\) is an abbreviation for the **quasiquote** symbol
- **read_unquote**: the ASCII comma character (,) is an abbreviation for the **unquote** symbol
- **read_arithops**: the ASCII characters +, -, *, /, and % are abbreviations for the **add**, **sub**, **mul**, **div**, and **mod** symbols
- **read_relops**: the ASCII strings =, <, >, <=, and >= are abbreviations for the **eq**, **lt**, **gt**, **le**, and **ge** symbols
- **read_number**: the ASCII characters (+ or -) or any digit begins a numeric value – either integer or real
- **read_symbol**: the last valid possibility is a **Scheme** symbol

You have probably already noticed from the brief descriptions above, that many of these lexical analysis elements recognize an operator and replace the short-hand notation with a longer **scheme** symbol, e.g., +, - , *, /, \, ', and ,.

Once again, he **skeme reader** may be found in its entirety in the file **translator.icn** in **Chapter 29: The End Result**.

24.3 Print

The **skeme printer** also does basically two tasks: print single atoms and print lists. The only new wrinkles to these two projects are (1) single items not only include atoms and but also functions (both built-in and user-defined) and (2) dotted pairs do not conform with either of the two basic categories!

The representation of functions is essentially implementation dependent. Built-in functions have a pretty obvious representation highlighting its identifier. User-defined functions (lambda forms), which we will discuss in the very near future, are comprised of two components, a list of its formal arguments and a body (i.e., a list) of executable code. Typically, the printer routine will display the three elements: identifier, formal arguments, and body for a lambda form.

Dotted pairs require an addition to the printer routine to represent a dotted pair in the output stream as:

[first . rest]

overview of scheme printer

```
procedure _print (x)
# processes an internal data representation
# to display a more human readable output

procedure print_item (x)
# print_item determines category of item to be displayed

   if (type(x) == "list") then
   else if (type(x) == "dotted_pair") then
   else if (type(x) == "primitive") then
   else if (type(x) == "special_form") then
   else if (type(x) == "lambda_form") then
   else if (type(x) == "macro_form") then
   else if (type(x) == "promise") then
   else if (_atomp(x)) then

procedure print_atom (x)
# display the value of a skeme atom in a readable form

   if (_integerp(x)) then
   else if (_realp(x)) then
   else if (_stringp(x)) then
   else if (_charp(x)) then
   else if (_booleanp(x)) then
```

```
   else if (_symbolp(x)) then

procedure print_list (x)
# print_item determines category of item to be displayed
# an atom or a dotted pair or a procedure

   if (_nullp(x)) then
     writes ("()")
   else
   {
     writes("(")
     every (i := 1 to *x) do
     {
       print_item(x[i])
       if (i < *x) then
         writes(" ")
       else
         writes(")")
     }
   }
end
```

24.4 Eval

Everything we have been discussing has been building up to this very moment. Everything that follows will be icing on the cake. But this section is the heart of **lisp** ... the heart of **scheme** ...

Unfortunately, this topic is very intricate and introduces a large number of new aspects regarding LISP-like languages that it is best for me to use the remaining pages of the chapter to provide an overview of what lies ahead.

At present, we have an operational read-print driver which is capable of reading input s-expressions and printing output in a similar readable format. However, nothing is taking place between input and output!

24.4.1 eval

Our first task is to understand the concept of evaluation. At first glance it seems pretty obvious what we should do. But unfortunately it introduces a new collection of terminology for us to master. For example, the simplest element in **scheme** is an atom. What does it mean to evaluate an atom? The answer is: it depends!

Some atoms, such as integers, reals, characters, strings, and boolean values, actually represent themselves. These are referred to as **self-evaluating** atoms and have an obvious evaluation!

$$(\text{eval } 123.45) \Rightarrow 123.45$$
$$(\text{eval } \text{"paul kaiser"}) \Rightarrow \text{"paul kaiser"}$$

All that remains to consider are atoms which are symbols, the component within **scheme** that plays the role of variables in other programming languages.

In order to understand evaluation for a symbol, we need to introduce the concept of an **environment**. The environment is more that just vaguely reminiscent of creating local storage on the activation stack while having global storage in main memory. When we start an interpreter, we have only a handful of predefined elements available. In order to utilize a symbol as a variable, we need to assign it a value – and to do this we must use a **define** instruction.

The **define** instruction will add the symbol to the environment and remember the designated value.

(define pi 3.1415) ⇒ environment augmented
(eval pi) ⇒ 3.1415

(define name "paul") ⇒ environment augmented
(eval name) ⇒ "paul"

(define x pi) ⇒environment augmented
(eval x) ⇒ 3.1415

However, the **define** instruction is **not** intended to modify the contents of a symbol. Although this may work in some implementations, it is considered poor practice. Rather, the **set!** instruction is preferred.

(eval name) ⇒ "paul"
(set! name "kaiser") ⇒ environment updated
(eval name) ⇒ "kaiser"

(eval x) ⇒ 3.1415
(set! x 1.414) ⇒ environment updated
(eval x) ⇒ 1.414

In essence, symbols are meaningless until they have been given meaning using the **define** instruction and symbols retain that meaning until they have been redefined using the **set!** instruction.

The two instructions **define** and **set!** add information into the current environment. There will be other instructions in the future that also impact the environment. These new instructions will roughly parallel the *calling of another procedure* and the *return from a procedure to its caller*, creating a form of **scope** for symbols.

But, we do not need to concern ourselves with these details just yet!

There is, however, one last detail – one last instruction – we need to introduce before moving away from discussing the evaluation of atoms.

Sometimes in programming, we are not so much interested in manipulating the value associated with a variable but rather in the variable itself. Both **lisp** and **scheme** are very focused on generally evaluating everything in sight! But, every now and then, we might **not** want to evaluate an item!

The **quote** instruction is the vehicle for indicating that *non-evaluation* requirement! It does not evaluate what follows.

(quote pi) ⇒ 'pi
(quote name) ⇒ 'name

In fact, an astute reader of my **skeme_reader** source code might have recognized the quote instruction has already been implemented within the processing of the input stream. The ASCII character single quote automatically generates the appropriate s-expression:

'x ⇒ (quote x)
'(a b c) ⇒ (quote (a b c))

We are now ready to progress from evaluating basic atoms to evaluating more complicated s-expressions.

When I say **s-expressions**, I really mean **s-expressions**. I do **not** mean **dotted pairs** specifically. Although **cons** and the resulting **dotted pair** may have been the fundamental constructor to create lists and s-expressions in LISP, the definition of **eval** is limited specifically to **s-expressions**.

Some **lisp/scheme** texts try to extend concepts to incorporate constructs which fall short of being an s-expression. For example,

A **proper list** is may be defined as a list of atoms that *correctly* ends with **nil** as the **cdr** of the final dotted pair.

An **improper list** is then defined as a list of atoms *but* for the detail that the **cdr** of the final dotted pair is not **nil**.

Consider the following two expressions. The first is a proper list; the second is an improper list.

```
( cons 1 ( cons 2 ( cons 3 '() ) ) )
(cons 1 ( cons 2 ( cons 3 4 ) ) )
```

The result of the first expression will be displayed in the usual list representation:

```
'( 1 2 3 )
```

while the result of the second expression will be be displayed in a slightly altered form than might be expected:

```
'( 1 2 3 . 4)          "list-like" in most implementations
[1 . [2 . [3 . 4]]]    "dotted pair" notation in skeme
```

Although structures comprised of dotted pairs of two atoms may prove useful in many contexts, as a general rule they must be defined, recognized, and maintain by the programmer and will have little native support within **lisp/scheme** interpreter.

So let us return to the question of evaluation for an s-expression, specifically a list of items each of which may, in turn, be an s-expression. The task is at hand is a two-parter:

- evaluate the initial item (**first**) to determine the function to apply
- apply that function to the argument list which immediately follows (**rest**)

If the s-expression were an atom, we would evaluate it as previously discussed. Simply look in the environment to retrieve its defined value.

If the s-expression were a list, the initial item must evaluate to some type of a function – either a *builtin function* provided by the language itself or a *previously defined user-implemented function* (called a **lambda form**). The general principle in evaluating a function is that, after determining the function to apply, we then evaluate all the subsequent items in the s-expression to determine the actual arguments. This technique of evaluating all the arguments immediately is commonly referred to as ***applicative evaluation***. The final step is to **apply** the *function* to the evaluated *actual arguments*!

If the s-expression included interior nested s-expression, we would recursively evaluate each of them as discussed in the paragraph immediately above. However, it is vital to remember that evaluating the initial item (even though it is itself an s-expression) yields the function which needs to be applied to the rest of the s-expression!

24.4.2 apply

Now we confront a new variation on a familiar theme. We have discussed the building blocks found in the **skeme** and we have written **Icon** programming language source code which implement these components and numerous very basic functions simple tests and simple functions.

We have already been implementing the first wave of *builtin functions*! Builtin Functions may also be referred to as **primitive functions**.

The next phase requires us to implement a real **apply** library of primitive functions which lie halfway between pure **scheme** code in our **skeme** interpreter and pure **Icon** code in our **core.icn** file. This library of primitive functions will have **Icon** procedure name that resemble its **skeme** name as closely as possible (given the subtle difference between identifier requirements for the two languages).

However, all arguments to these primitive functions will be through a single formal argument which is itself a **skeme** list – which reflects a higher level of abstraction in our implementation.

We have moved **Icon** listing its arguments in order toward **scheme** having all its arguments in a list (possibly empty).

(***func*** arg1 arg2 ...) in **Scheme**

to

func ((arg1 arg2 ...)) in **Icon**

This next phase, implementing the **apply** library of primitive functions, will be the focus of the next chapter. Right now we will conclude our discussion regarding the topic of evaluation by highlighting the exception to the general principle of "evaluate everything!"

24.4.3 quote

There are times when we **either** *do not need* to evaluate everything **or** *do not want* to evaluate everything!

examples

Consider the **define** primitive which we presented earlier:

(**define** x 123)

We do not want to evaluate the symbol x! It has not been defined yet! That is what we are attempting to do! But we do want to evaluate the integer 123.

Consider the standard **if** (< test >) **then** ... **else** ... control structure in procedural languages. Its **lisp** incarnation is the following s-expression:

(**if** <test-expression> <true-expression> <false-expression>)

We must surely evaluate the test-expression. But we do not know which of the two subsequent expressions will be evaluated until we know the result of the test!

Consider the boolean logical operator **and** which is illustrated immediately below:

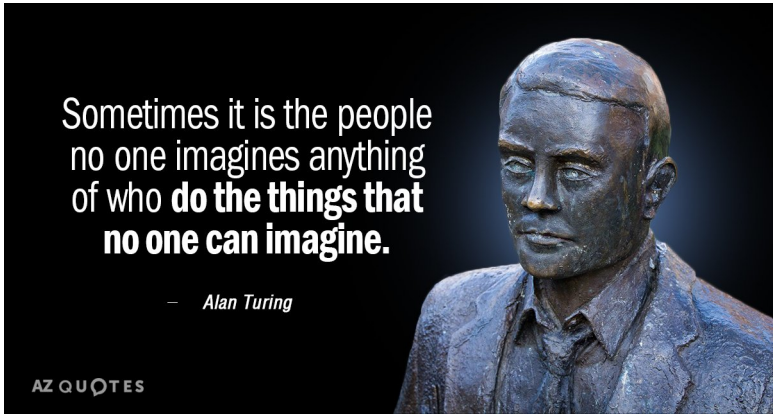
(**and** (> a 0) (> b 0) (> c 0))

Suppose we evaluate the first comparison and it is false? We certainly know the outcome without having to evaluate the subsequent comparisons. Short-circuit logic is advantageous because it helps speed up computation and also helps simplify code at the same time!

Lastly, consider the programmer who wants to reference a specific symbol, not its evaluation! This is the purpose behind the **quote** primitive.

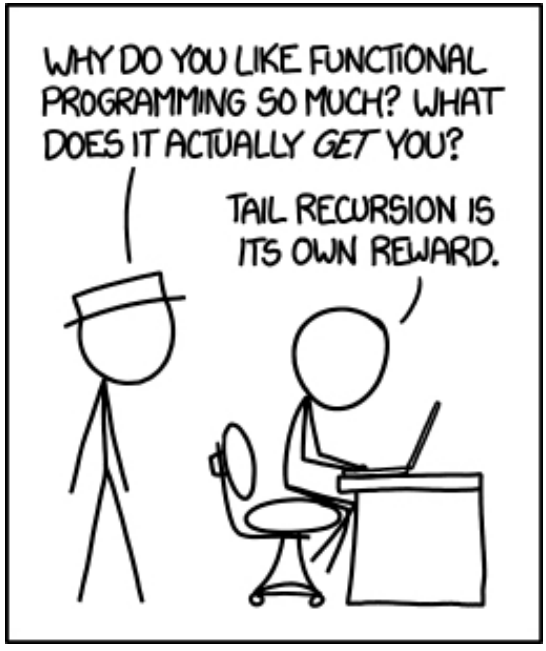
'x or (quote x)
'(a b c) or (quote (a b c))

This primitive is recognized by its short-hand notation within the **skeme-reader**.


Sometimes it is the people
no one imagines anything
of who **do the things that**
no one can imagine.
– *Alan Turing*
AZ QUOTES

Chapter 25

Built-in Procedures



In this chapter we will implement the most important elements of the scheme interpreter. We know all the building blocks for the language. We have implemented these building blocks (in **core.icn** and in **translator.icn**),. We have implemented a lexical scanner (**_read**) to recognize s-expressions, and we have implemented a display utility (**_print**).

So now the focus is on the ***eval*** component in the **read-eval-print** loop!

Recall the **scheme** s-expressions are commonly lists: lists of atoms, symbols, and other lists). Our scanner provides the **skeme** interpreter with expressions that look like:

(**function** x_1 x_2 x_3 ...)

We now to unwind these **skeme** expressions into something our **Icon** implementation of an interpreter can process – most probably calling upon our **core.icn** building blocks, which have the form:

_function (x_1, x_2, x_3, ...)

Our primitive procedures will provide a transition between s-expressions and procedural language statements – they will be a hybrid halfway house along the way:

function (***arg_list***)

Arg_list will be a **scheme** s-expression provided by our lexical scanner (**_read**).

Primitives will pass information through to *core procedures* to evaluate or they will call on *other primitives* to perform the desired action.

25.1 Introduction to Procedures

We have to begin somewhere. And we may be a a bit overwhelmed by all the possibilities! As just mentioned, **eval** is the place to start! Or, better stated, the triple **eval** – **apply** – **quote** is the place to start!

The _**eval** procedure in **core.icn** is responsible for the determining the correct value associated with a given s-expression:

- Is it a "self-evaluating" atom: empty list, number, character, string, boolean?
- Is it a symbol? If so, has it been defined?
- Is it a procedure? If so, what is its definition?
- And if it is a procedure? Then it needs to be "applied" to the subsequent arguments.

The _**apply** procedure in **core.icn** oversees the application of a function to its arguments:

- a **primitive** is generally applied to its arguments only after *all* arguments have been evaluated!
- a **special form** is also a primitive *but* it selectively chooses which arguments to evaluate!
- a **lambda form** is a user-defined procedure that adheres to the general principle: evaluate everything!
- a **macro form** is a user-defined procedure that selectively chooses which arguments to evaluate!

The _**quote** procedure in **core.icn** allows the **lisp/scheme** programmer a bit of control regarding evaluation:

- If **quote** is the procedure to be applied, then the immediately following s-expression is returned without evaluation.
- Otherwise, the normal eval / apply process is performed.

Core Implementation for Eval/Apply/Quote

```
procedure _eval (item)
# remember the general principle: evaluate everything!

  # check for "self-evaluating" atoms
  _nullp(item) & return []
  (_numberp(item) | _charp(item) | _stringp(item) |
   _booleanp(item) | _procedurep(item)) &
     return item
  # check for atoms previously defined within environment
  _symbolp(item) &
     return (search(item) | item.identifier||" not found!")
  # check for valid s-expression to evaluate
  _listp(item) &
     return _apply(_eval(_first(item)),_rest(item))
  # promise is returned unevaluated
  _promisep(item) &
     return item
  # dotted pair is returned unevaluated
  _pairp(item) &
     return item
  # associative list is return unevaluated
  _assoc_listp(item) &
     return item
end


procedure _apply (proc,args)
# func may be:
#      either a previously defined symbol
#      or an anonymous lambda form or macro form

# remember the general principle: evaluate everything!
#      "call by value" adheres to this principle
#      "call by name" does not

  _primitivep(proc) &
     return proc.identifier(_evaluate_arguments(args))
  _special_formp(proc) &
     return proc.identifier(args)
  _lambda_formp(proc) &
     return _procedure_evaluation(proc,_evaluate_args(args))
  _macro_formp(proc) &
     return _eval(_procedure_evaluation(proc,args))
end


procedure _quote (item)
# the exception to general principle: evaluate everything!

  return item
end
```

```
procedure _evaluate_args (args)
# helper procedure to evaluate actual args for function

   _nullp(args) & return []
   return [_eval(_first(args))] |||
          _evaluate_args(_rest(args))
end
```

Moving one up notch within the layers of abstraction, these same core procedures would appear within our primitives (**primitives.icn**) file as follows:

Primitive Implementation for Eval/Apply/Quote

```
procedure eval (args)
  (_length(args) = 1) | fail
  item := _first(args)
  # the single argument to eval must be an s-expression
  _listp(item) | fail
  return _eval(item)
end

procedure apply (args)
  (_length(args) = 2) | fail
  proc := _first(args)
  # the first argument is a procedure
  _procedurep(proc) | fail
  arguments := _first(_rest(args))
  # the second argument is a list of actual arguments
  _listp(arguments) | fail
  return _apply(proc,arguments)
end

procedure quote (args)
  (_length(args) = 1) | fail
  item := _first(args)
  # the single argument to quote could be anything!
  # no need to call _quote in core.icn!
  return item
end
```

The first obvious addition to **primitives.icn** would be ***los tres amigos***: **cons**, **first**, **rest**.

Primitive Implementation for Los Tres Amigos

```
procedure cons (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
  item2 := _first(_rest(args))
  return _cons(item1,item2)
end

procedure first (args)
  (_length(args) = 1) | fail
  return _first(_first(args))
end

procedure rest (args)
  (_length(args) = 1) | fail
  return _rest(_first(args))
end
```

Do It Now? or Do It Later?

Once we have the six procedures above implemented and working, we are on the cusp of something really fun and rewarding.

We can enter very simple s-expressions into our interpreter and see immediately whether or not things are working correctly. Debugging can be a bit nasty, especially if you are new to functional programming. But once the basic six are working, you have a functioning interpreter which will help with future work.

I personally found it useful to have **mit-scheme** or **drracket** handy to compare results. It is always a very good sign if the results from your **skeme** interpreter match the results from thoroughly tested software.

I now recommend you consider implementing any or all of the following primitives!

simple predicates

- **integer?**, **real?**, **number?**
- **char?**, **string?**

- **boolean?**, **symbol?**
- **atom?**
- **null?**, **pair?**
- **list?**, **sexp?**
- **primitive?**, **special-form?**
- **lambda-form?**, **macro-form?**
- **procedure?**

equality predicates

- **eq?**, **eqv?**, **equal?**

Please note that the structure for primitive procedures is very similar for all entries:

- check the length of the argument list
- retrieve the actual arguments from the argument list
- call the underlying core procedure to determine the return value

This first collection of primitives are the basic predicates to recognize data types:

Primitives Implementation

```
procedure integerp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_integerp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure realp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_realp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

        :
        :
        :
```

This second collection of primitives are the basic tests for equality:

Primitives Implementation

```
procedure eqp (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
  item2 := _first(_rest(args))
  if (_eqp(item1,item2)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure eqvp (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
  item2 := _first(_rest(args))
  if (_eqvp(item1,item2)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure equalp (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
  item2 := _first(_rest(args))
  if (_equalp(item1,item2)) then
    return boolean("#t")
  else
    return boolean("#f")
end
```

The good news is that so much of the hard lifting for the work required in the primitives above was already done, having already being implemented in the core routines! Unfortunately, the bad news is that there are so many possibilities for primitive procedures we have already and could possibly implement in the future!

At this point in our study, we have the luxury of picking and choosing items that are immediately useful and postponing others for later. I include a lot of other primitives in my **Icon** source code **primitives.icn** for you to consider. Rome was not built in a day; neither should a **skeme** interpreter! Get a small collection of building blocks working properly. Then, over time, gradually but steadily add other collections to it.

You should take some time now and start enhancing your own **skeme** interpreter. Or you can move on to the next section and learn about environments and scope in **scheme**. It is completely your option!

However, eventually you will want to return to any unfinished business from this section. And ... **spoiler alert** ... subsequent sections in this chapter will highlight even more primitives for your consideration!

25.2 Introduction to Environments

There is one major question we glossed over in previous discussion. Recall:

Symbols are just symbols and have no meaning
until they are given meaning using ***define****.*

We have briefly discussed two key **scheme** primitives:

- **define** is used to give meaning to a previously *undefined* symbol
- **set!** is used to update a previously *defined* symbol

For programmers familiar with procedural languages, the **assignment statement** is a very powerful and familiar element. However, some of the most difficult errors to find in procedural coding are caused by side effects created by aliases (i.e., alternate references) to the same storage location in memory. LISP languages tend to downplay assignment statements and rely more heavily on combining results from s-expression (i.e., *function*) evaluation into the final value that is then returned. Hence, the name ***functional programming***!

Like all programming languages, **scheme** does need to keep track of its defined symbols. And like all programming languages, **scheme** does utilize some form of a symbol table. A **scheme** symbol table is an especially simple structure:

- a symbol
- its meaning (value)

The meaning or value of a symbol can be in several forms:

- a **scheme** constant value
- a **scheme** symbol (not a constant value)
- a **scheme** primitive name
- a **scheme** lambda form (to be discussed later)
- a **scheme** macro form (to be discussed later)
- a **scheme** promise (to be discussed later)

Initially, all we need is one universal symbol table. The only items to maintain are essentially: symbols that have been defined and primitives that are builtin.

So, our **initial environment** will be a single symbol table.

This will work fine *until* we expand our vision of the language to include user-defined functions. At that point, we will have to reconsider such a simple implementation of environment. But that waits in the future.

Unfortunately, my source code in **environment.icn** already provides a window into that future. Our sole symbol table will ultimately morph into a stack of symbol tables. With that in mind, you should be able to understand the code which follows immediately. A more detailed description of the environment and its implementation will be delayed to the appropriate time.

Environment Initial Symbol Table

```
global environment
record scope_info (symbol_table,current,parent)

procedure initialize_environment ()
  environment := []
  symbol_table := table()

  # predefined atoms

  symbol_table["nil"]           := []
  symbol_table["true"]          := boolean("#t")
  symbol_table["false"]         := boolean("#f")
  symbol_table["else"]          := boolean("#t")
  symbol_table["e"]             := 2.718281828
  symbol_table["pi"]            := 3.141592654

  # primitive predicates

  symbol_table["integer?"]      := primitive(integerp)
  symbol_table["real?"]         := primitive(realp)
  symbol_table["number?"]       := primitive(numberp)
  symbol_table["char?"]         := primitive(charp)
  symbol_table["string?"]       := primitive(stringp)
  symbol_table["boolean?"]      := primitive(booleanp)
  symbol_table["symbol?"]       := primitive(symbolp)
  symbol_table["atom?"]         := primitive(atomp)

  symbol_table["null?"]         := primitive(nullp)
  symbol_table["list?"]         := primitive(listp)
```

```
  symbol_table["pair?"]          := primitive(pairp)
  symbol_table["alist?"]         := primitive(alistp)

  symbol_table["primitive?"]     := primitive(primitivep)
  symbol_table["special-form?"] := primitive(special_formp)
  symbol_table["lambda?"]        := primitive(lambdap)
  symbol_table["macro?"]         := primitive(macrop)
  symbol_table["procedure?"]     := primitive(procedurep)
  symbol_table["promise?"]       := primitive(promisep)
  symbol_table["eq?"]            := primitive(eqp)
  symbol_table["eqv?"]           := primitive(eqvp)
  symbol_table["equal?"]         := primitive(equalp)

  # initial primitives

  symbol_table["cons"]           := primitive(cons)
  symbol_table["first"]          := primitive(first)
  symbol_table["rest"]                 := primitive(rest)

  symbol_table["apply"]          := primitive(apply)
  symbol_table["eval"]           := primitive(eval)
  symbol_table["force"]          := primitive(force)
  symbol_table["gensym"]         := primitive(gensym)
  symbol_table["load"]           := primitive(load)
  symbol_table["newline"]        := primitive(newline)
  symbol_table["print"]          := primitive(print)
  symbol_table["promise-forced?"]:= primitive(promise_forcedp)
  symbol_table["promise-value"] := primitive(promise_value)
  symbol_table["read"]           := primitive(skeme_read)
  symbol_table["quit"]                 := primitive(quit)

  symbol_table["environment"]    := primitive(dump)

  put(environment,scope_info(symbol_table,1,0))
end
```

This will work fine *until* we expand our vision of the language to include user-defined functions. At that point, we will have to reconsider such a simple implementation of environment. But that waits in the future.

My source code in **environment.icn** already provides a window into that future. Our sole symbol table will ultimately morph into a stack of symbol tables. With that in mind, you should be able to understand the code which follows immediately. A more detailed description of the environment and its implementation will be delayed to the appropriate time.

Environment Support Routines

```
procedure search (symbol)
# searches for a symbol in the database
# starting with the current environment
#     and searching backwards through the static chain
# either succeeds or fails

procedure level (symbol)
# searches for a symbol in the database
# starting with the current environment
#     and searching backwards through the static chain
# either succeeds or fails

procedure insert_environment (symbol,value)
# defines a previously undefined symbol

procedure update_environment (symbol,value)
# updates a previously defined symbol

procedure dump ()
# display the entire set of symbol tables,
#     its current level and its parent level
#     for each level in the environment
#     starting with the current level
#     and working backword to initial environment
```

25.3 Even More Primitives

Now that we have a few builtin procedures under our belt and we have created an initial universal environment for defining symbols, we can tie together some loose ends.

This particular section will be short on substance but a bit long on suggestions regarding the **skeme** interpreter. We have already identified numerous primitives that provide flexibility and breadth to the language. This section will summarize several more possibilities to consider.

More Candidates to Implement as Primitives

- gensym
- load
- newline
- print
- read (n.b., this causes a name collision with **Icon**keyword!)
- quit

Even More Candidates to Implement

- numeric operators: add, sub, mul, div
  all four operate on a *list* of operands
- numeric comparisons: n-eq, n-gt, n-lt, n-ge, n-le
  all four operate on a *list* of operands
- other numeric primitives: ceiling, floor, sqrt,
  sin, cos, tan, exp, log
- character primitives: char=?, char<?, char>?,
  char<=?, char>=?,
  char-alphabetic?, char-numeric?, char-whitespace?,
  char-upper-case?, char-lower-case?,
  char->integer, integer->char,
  char-upcase, char-downcase
- string primitives: make-string, string,
  string-length, string-ref,

string=?, string<?, string>?, string<=?, string>=?
string-copy, substring, string-append,
string->list, list->string, string->symbol, symbol->string

- list primitives: list, length, append, reverse,
  list-tail, list-head, list-ref
  memq, memv, member,
  assq, assv, assoc

The following **Icon** source code implements the first collection of additional candidates:

Additional Primitives

```
procedure gensym ()
# generate a unique scheme symbol upon request
  return symbol(@_gensym)
end

procedure load (args)
# process skeme input from a file
# rather than from stdin
# using a read-eval-butnoprint loop (REBL?)
  static LOADLIB
  LOADLIB := "/opt/skeme/libskeme/"
  (_length(args) = 1) | fail
  file_name := _first(args)
  _stringp(file_name) | fail
  if (fin := (open(file_name) | open(file_name||".scm") |
              open(LOADLIB||file_name) |
              open (LOADLIB||file_name||".scm"))) then
    write("reading "||file_name||" ... ")
  else
  {
    write("can't open "||file_name)
    fail
  }
  # redirect input from stdin to fin
  old_source := input_source    # save previous source
  input_source := fin           # identify new source
  while (item := _read()) do
    _eval(item)
  close(fin)
  write("      ... done!")
  # redirect input from fin to stdin
  input_source := old_source    # restore previous source
  return
end

procedure newline ()
```

```
# send a carriage return to output

procedure print (args)
# use procedure _print from skeme_reader
# to print single elements (atoms, lists, ... ) to output

procedure skeme_read (args)
# skeme_read avoids conflict with Icon procedure read
# read from standard input

procedure quit ()
# no explanation needed!
```

25.4 Special Forms

Recall, there are times when we either **do not need** to evaluate everything or **do not want** to evaluate everything!

Recall the two fundamental instructions for associating a value with a symbol – **define** and **set!**. These two necessary commands **can not** be implemented as primitives (in the very precise sense of a primitive function)!

If **define** is a primitive function, it must evaluate all its arguments. The first argument is the very symbol it is trying to define! It needs the actual symbol, and not its value.

If **set!** is a primitive function, it must evaluate all its arguments. Once again, the first argument is the symbol it is trying to update. The current value of the symbol has no bearing on the desired outcome; it is the actual symbol that is important.

A **special form** is very similar to a **primitive**. Both are builtin procedures. The difference is that a **primitive** evaluates all its arguments prior to *applying* procedure and that a **special form** does not. A **special form** will selectively choose which arguments to actually evaluate.

Initial Special Forms

- (**begin** expression-list)
  evaluate expression-list in sequence
  return the last evaluation
- (**define** symbol expression) and (**set!** symbol expression)
  evaluate the expression, but not the symbol
- (**quote** expression)
  do not evaluate the expression
  previously discussed and implemented!
- (**if** test-expression true-expression [false-expression])
  evaluate the test-expression
  if #t, then evaluate the true-expression
  if #f, then evaluate the false-expression (if present!)
- (**and** true-false-list) and (**or** true-false-list)
  in **and** the first false-value immediately returns #f
  in **or** the first true-value immediately returns #t

- but not (**not** true-false item)!
  the sole argument must be evaluated
  it is truly a primitive

Special Forms

```
procedure begin (args)
# scheme special form begin
# block of Scheme code to be executed as single unit
# result returned is result of the last statement

  (_length(args) = 0) & return []
  result := _eval(_first(args))
  if (_nullp(_rest(args))) then
    return result
  else
    return begin(_rest(args))
end

procedure define (args)
# scheme special form define
# inserts symbol into environment, assigns initial value
# see procedure set! (setx) below

  if (_listp(_first(args))) then
  {
    signature := _first(args)
    id := _first(signature)
    _symbolp(id) | (write ("invalid define") & return)
    (defined := search(id)) & return
    formals := _rest(signature)
    body := _rest(args)
    scope := *environment
    (define_value := lambda_form(formals,body,scope)) |
      fail
    insert_environment(id,define_value)
    return define_value
  }
  else
  {
    (_length(args) = 2) | fail
    id := _first(args)
    value := _first(_rest(args))
    _symbolp(id) | (write ("invalid define") & return)
    (defined := search(id)) & return
    (define_value := _eval(value)) | fail
    insert_environment(id,define_value)
    return define_value
  }
end

procedure if_then_else (args)
# scheme special form if
```

```
# first  item in args identifies test to be performed
# second item in args identifies what to evaluate if true
# third  item in args identifies what to evaluate if false
#     optional argument! nil if not specified!

  (_length(args) > 3) & fail
  (_length(args) < 2) & fail
  test := _first(args)
  t_expr := _first(_rest(args))
  if (_length(args) = 3) then
    f_expr := _first(_rest(_rest(args)))
  else
    f_expr := []
  if (_eqvp(_eval(test),boolean("#f"))) then
    return _eval(f_expr)
  else
    return _eval(t_expr)
end

procedure setx (args)
# scheme special form set!
# updates symbol found in environment with new value
# see procedure define above

  (_length(args) = 2) | fail
  id := _first(args)
  value := _first(_rest(args))
  _symbolp(id) | (write ("invalid set!") & return)
  (defined := search(id)) | return
  (update_value := _eval(value)) | fail
  update_environment(id,update_value)
  return update_value
end

# procedure and (args)
# scheme special form and
# rather than define this as a builtin special form
# postpone to present this as syntactic sugar


# procedure or (args)
# scheme special form or
# rather than define this as a builtin special form
# postpone to present this as syntactic sugar

# procedure not (args)
# scheme primitive form not
# rather than define this as a builtin special form
# postpone to present this as syntactic sugar
```

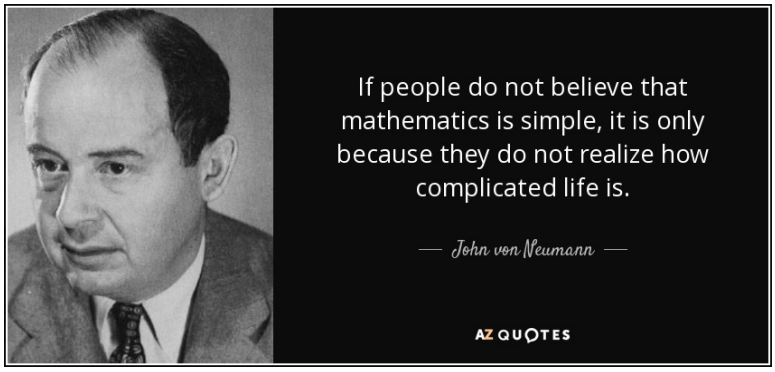
25.5 Closing Thoughts

I would like to conclude this chapter with two comments.

The first comment is that there is no intrinsic problem in combining both *primitive functions* and *special forms* together in a single **builtin.icn** source file! However, I clearly differentiate between the two categories of built-in procedures in their representation within **icon** as different record types and I clearly differentiate between the two categories in the coding found in the source file **core.icn** and in the source file **environment.icn**.

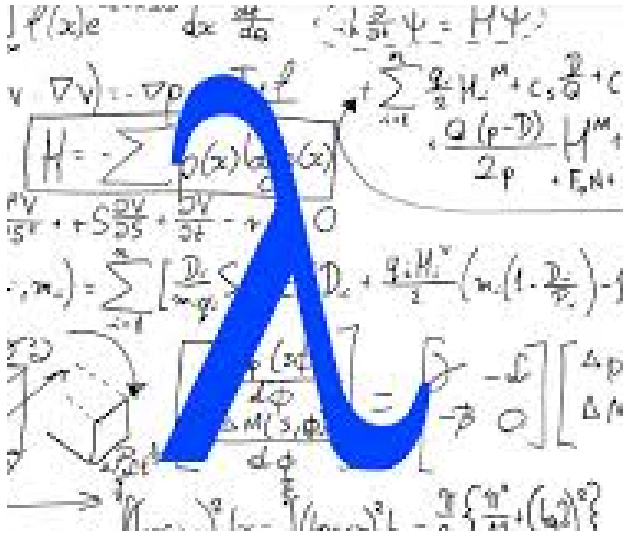
The second comment is to offer encouragement to anyone just beginning the study of computer science. I am seventy-four years old at the time I am writing this. I have been off and on associated with mathematics and computing for sixty years. While studying computer programming and various programming languages, I quickly learned (and even understood) a lot of jargon – call by value, call by variable, scope, pointers, ... But I did not realize until implementing a **skeme** interpreter that the mysterious technique referred to as call by name for transferring data is nothing more than the **quote** special form. I always thought it was some magical transformation that was beyond my comprehension. Now I realize it actually means "do nothing" – just pass the expression through!

So ... no matter how old you are ... and no matter how much you think you know ... there will always be some new insight and opportunity to learn.


If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.
— John von Neumann —
AZ QUOTES

Chapter 26

The *Infamous* Lambda



So we arrive now at a very crucial crossroads in our development of our **skeme** interpreter. We are no longer focusing on the building blocks of the language and how they work. Rather, we are at the point where we will define our own components and define how we want them to work.

One of the coolest things about **lisp** and/or **scheme** is that it is so simple to implement new features. And if you do not like a current feature, then you can replace it with something of your own.

Few languages give the programmer such flexibility as to reshape and redefine the language itself.

Our first step in this journey is to take a focused look at what exactly comprises a mathematical function. **lisp** and **scheme** are referred to as functional programming languages because they are built around this core concept. So we begin there.

26.1 Mathematical Functions

essential components

Consider the good old "square" function

$$f(x) = x^2$$

What is ***absolutely essential*** in the single line above to fully understand the essence of the definition?

Certainly not its name **f**! Why it could just as easily be **g**, or **h**, or **square**? Would a rose by any other name smell as sweet?

And certainly not the parentheses! They just keep the independent variable **x** company.

The variable **x** is actually quite important – it identifies the number of arguments and represents the value that will be given to the function. In this case, the number of arguments is one.

The final important component appears on the right-hand-side of the equal sign – the actual formula, or rule of correspondence, which defines specifically how the arguments will be combined to arrive at the resulting value for the function.

Alonzo Church in his mathematical research regarding functions, their symbolism, and their application developed a notation referred to as the **lambda calculus** which takes its name from the greek letter λ which was his representation for an anonymous function. Using Church's notation, the square function would be represented by

$$\lambda x.x^2$$

LISP seems to have adopted this nomenclature for a generic function. User-defined functions are specified using **lambda forms** which identify:

- the formal argument list: the names and number of values which will be given to the function
- the body: the sequence of steps to be used to arrive at the desired result

implementation issues

Now that we have identified the theoretical essentials that comprise a function, we can look ahead to how we can best implement them in our interpreter. Lambda forms are an excellent candidate requiring only two pieces of information: the formal arguments (a list of symbols to be "substituted for" or "replaced" using actual values) and the rule of correspondence (a list of s-expressions comprising the body of the function). This is a nice and simple representations.

We can specify the actual argument values to be transferred to the function by a simple list. The only important restriction is that actual arguments as a list appear in the same order as the formal arguments so that they match up properly.

Whereas theoretical functions have no concern how the arguments arrive for them to do their magic, computer implementations must consider this issue. So we have returned once again to the question of how an actual argument is transferred to its formal argument equivalent. And in LISP programming languages, the two choices are *call by value* and *call by name*, that is:

- either evaluate the s-expression for the actual argument and define the formal argument to be the result
- or do not evaluate the actual argument and define the formal argument to be the unevaluated (quoted) s-expression

But it is important that we identify the data transfer mechanism that we want for the function we are implementing. It is a key first step.

My suggestion at this point is the following simplification:

- a function which utilizes *call by value* will be represented as a **lambda form** pair – a formal argument list and body (list of statements)
- a function which utilizes *call by name* will be represented as a **macro form** pair – a formal argument list and a body (list of statements)

arguments	**builtin**	**user-defined**
eval (by value)	primitive	lambda form
uneval (by name)	special form	macro form

This characterization may possibly be a gross over-simplification! But it does encapsulate the basic variations of procedures we will encounter within the language. **Lisp** and **scheme** certainly differ significantly in their attitudes towards macros in general. **Lisp** programmers tend to love the flexibility and the freedom that macros provide; **scheme** programmers tend to focus on potential anomalies that might arise from such flexibility.

As a novice programmer approaching both languages essentially for the first time, I found the **lisp** syntax a bit friendlier than **scheme** syntax for defining macros. However, I do not want to introduce an entirely new set of instructions unique to **scheme** at this point in time, but continue to use the building blocks we have been discussing.

So remember that lambda forms and macro forms **both** represent functions in **scheme** and that the primary difference between the two is the transfer of information from actual arguments to formal arguments.

We will see shortly that this difference in transfer mechanism allows macro forms to perform significantly different applications than those traditionally performed by lambda forms.

environments

At this time we must also focus specifically on the concept of environment – what it is and how it works?

Until now, everything we have done has been executed within a single global or universal environment. We can certainly define new symbols to add to the environment, but they have always been defined within this universal environment. Other programming languages distinguish between global variables and local variables: variables that are known everywhere versus variables that are only known within a smaller region. **lisp** and **scheme** also distinguish between local and global.

When we built our **kize** compiler we discussed the difference between simple linear scope and nested scope. The **kize** programming language used linear scope to simplify implementing an activation stack for keeping track of function and subroutine variables and information necessary to properly **return** to the **calling** procedure.

One big advantage of simple linear scope is that any variable encountered is either local or global (or undefined). Finding the address for a variable is a very simple process. And the steps necessary to call and later return from a procedure only requires remembering the return address for the caller.

But now we are considering a different programming language with some different rules regarding how things are organized and how things interact.

First, the actual arguments to a function are to be determined within the current environment, the "calling" environment. Ignore for the moment the transfer mechanism that will be used.

Second, the formal arguments for a function are symbols which have meaning in an entirely new environment, the "called" environment. Formal arguments may duplicate symbols that may have been used elsewhere; comparable to local variables may duplicate global variables. And as we encountered previously local variables take precedence over global variables when determining which value they represent.

Lastly, the environment in which a function is *defined* contains the information which is available for the function to freely use – not just the information within the actual arguments. Since functions create a new environment within the previous environment and functions may in turn define new symbols and new functions, we generate a hierarchical structure much like **Matryoshka Dolls** nested within one another in ever decreasing size.. And the last complication we must comprehend is that the "calling" procedure need not be the "defining" procedure (but it must be inside the "defining" procedure as well).

The very first steps in activating a function will be:

- determine (within the current environment) the actual arguments
- augment the environment with a new component
  - a new symbol table (initially empty)
  - a reference to the **calling** environment
  - a reference to the **parent** environment or the **defining** environment
- define each formal argument with its actual argument in the new environment
  - the evaluated result (call by value)
  - the unevaluated result (call by name)

Previously, with simple scope, we did not need to concern ourselves with the parent environment! Everything was either local or global. In the **kize** programming language, the activation stack only had to remember the return information (i.e., the "calling" procedure). In the **skeme** programming language, the environment stack has to remember **two** items (i.e., the "calling" environment and the "defining" environment).

The calling environment sequence is typically referred to as the **dynamic chain** and returns to the initial environment in reverse order of the original calling sequence. The defining environment sequence is typically referred to as the **static chain** and returns to

the initial environment also in reverse order but typically skipping over intervening environments.

When searching the environment to **define** a symbol, we need only consider the **current** environment! If the symbol is not defined in the current environment, then we may define its value; if it is already defined, then we have a problem.

When searching the environment to **set!** a symbol, we need to not only consider the **current** environment, but also its **parent** environment, . . . , all the way back to the initial environment. Only if we find a defining environment may we update the value of that symbol.

activation

Activation of functions is frequently done as one might expect from our previous experience with primitives and special forms. The **function_name** appears as the **car** of an s-expression, and the **actual_arguments** immediately follow as the **cdr**).

(**function_name** actual_arg$_1$ actual_arg$_2$ . . .)

However, both **lisp** and **scheme** both allow for ***anonymous*** (nameless) functions and, although you may never opt to use them, you will certainly encounter them.

((**lambda** (formal_arg$_1$ formal_arg$_2$ . . .) body) actual_arg$_1$ actual_arg$_2$ . . .)
((**macro** (formal_arg$_1$ formal_arg$_2$ . . .) body) actual_arg$_1$ actual_arg$_2$ . . .)

Rather than using **define** to pair a symbol with the lambda form or the macro form, you can provide the function definition with the actual arguments for immediate evaluation. Unfortunately, that will be the only time you ever use that incarnation of the function! Unless you provide it with a name it will have vanished without a trace.

Simple Illustrative Example

named functions

(define square (lambda (n) (* n n))

(square 7) $\Longrightarrow$ 49
(square (* 3 4)) $\Longrightarrow$ 144

anonymous functions

((lambda (n) (* n n n)) 3) $\Longrightarrow$ 27
((lambda (m n) (* (+ m n) (- m n))) (+ 2 3) 4) $\Longrightarrow$ 9

(((lambda (x) (lambda (y) (- x y)) 5)) 3) $\Longrightarrow$ -2
((lambda (x) (lambda (y) (- x y)) 5)) $\Longrightarrow$ anonymous function Lx.x-5
(lambda (x) (lambda (y) (- x y))) $\Longrightarrow$ anonymous function Lx.Ly.x-y

26.2 Lambda Forms vs Macro Forms

At this point we need reflect for a moment on a very subtle distinction that has very far reaching consequences. We have categorized functions into two distinct camps, yet functions are essentially very basic and straight forward mathematical concept. Why the distinction?

As LISP evolved and developed users began to notice that the data transfer mechanism (call by value versus call by name) facilitated different types of applications.

Lambda forms using call by value was more than adequate for implementing a very traditional type mathematical function. Send in the data to be manipulated, calculate the appropriate value, and return the generated result.

It was simple and it worked!

Macro forms using call by name seems at first glance to just be the mirror image of a lambda form. Does the relationship between macro forms and lambda forms simply parallel the relationship between special forms and primitives?

Or are there other possibilities lurking within this alternate implementation?

Exaggerating this example to an extreme, consider the difference between numeric calculations and string manipulations in a procedural language. The former generates a new numeric value; the latter generates a new string.

lambda forms operate on atoms

5, 12, -8

macro forms operate on s-expressions

(+ x y), (* a b), (- (* b b) (* 4 a c))

Macro forms provide a simple mechanism for cutting and pasting s-expressions into new s-expressions and ultimately evaluating the result. A macro form has the power to rearrange a simple s-expression into a very different, more complicated form.

Macros: Lisp versus Scheme

Unfortunately, this exciting feature has created a schism in the LISP world!

Lisp programmers appear to be in love with the concept of macros! They extol their simplicity, their flexibility, their capability. **Lisp** books highlight the virtues of **lisp** macros while making fun of most other attempts at implementing a macro in other programming languages.

Scheme programmers, on the other hand, seem to be afraid and intimidated by macros! Because symbols can have different meanings (or no meaning at all) in different environments, macros have the potential to create scope issues. Preoccupation with this possibility, **scheme** has implemented alternative syntax to the basic macro form found in **lisp**.

At this point in the book, I do not want to introduce a new layer of terminology and language syntax into our discussion. I would prefer to cover basic concepts and implementation. I temporarily am siding with the **lisp** programmers regarding macros to move ahead. So **skeme** macros will more closely resemble **lisp** macros in syntax than any **scheme** implementation of macros.

26.3 Modifications: skeme files

core.icn

The core implementation must provide for four types of procedures/functions:

- primitive functions (using call by value)
- special forms (using call by name to selectively evaluate arguments)
- lambda forms (user-defined functions using call by value)
- macro forms (user-defined functions using call by name)

These four types are represented as **Icon** records. The four types each have a core predicate associated with it.

Lastly, toward the bottom of the **core.icn** file are three support procedures associated with **skeme** functions.

- _procedure_evaluation
- _define_arguments
- _ evaluate_arguments

Modifications to Core

```
# basic procedures / functions
# builtin components:
#    primitive    -- arguments are all evaluated
#    special_form -- arguments are not evaluated
record          primitive (identifier)
record          special_form (identifier)

# user defined procedures
#    lambda_form  -- arguments are all evaluated
#    macro_form   -- arguments are not evaluated
record          lambda_form (formals,body,scope)
record          macro_form (formals,body,scope)

procedure _primitivep (item)
  return (type(item) == "primitive")
end

procedure _special_formp (item)
  return (type(item) == "special_form")
end
```

```
procedure _lambda_formp (item)
  return (type(item) == "lambda_form")
end

procedure _macro_formp (item)
  return (type(item) == "macro_form")
end

procedure _procedurep (item)
  return (type(item) ==
          ("primitive" | "special_form" |
           "lambda_form" | "macro_form"))
end

procedure _procedure_evaluation (proc,args)
# helper procedure to facilitate evaluation
# of either lambda form or macro form

  formals := proc.formals        # proper list formal args
  executables := proc.body
  scope := proc.scope
  actuals := args                # proper list actual args
  add_scope(scope)               # set up activation record
  _define_arguments(formals,actuals)
  result := begin(executables)   # executive body
  remove_scope()                 # remove activation record
  return result
end

procedure _define_arguments (formals,actuals)
# helper procedure to transfer actual args to formal args
# prior to executing either  lambda form or macro form

  if (_symbolp(formals)) then
  {
    # insert variable-value pair into new environment
    insert_environment(formals,actuals)
    return
  }
  _nullp(formals) & _nullp(actuals) & return
  if (_length(formals) = _length(actuals)) then
  {
    formal_variable := _first(formals)
    actual_value := _first(actuals)
    _symbolp(formal_variable) | fail
    # insert variable-value pair into new environment
    insert_environment(formal_variable,actual_value)
    return _define_arguments(_rest(formals),_rest(actuals))
  }
end

procedure _evaluate_arguments (args)
```

```
# helper procedure to evaluate actual args for function

   _nullp(args) & return []
   return [_eval(_first(args))] |||
      _evaluate_arguments(_rest(args))
end
```

special_forms.icn

lambda and **macro** are both instructions that are selective in the arguments that are evaluated and in the arguments that are left unevaluated. Hence they are both special forms, and their implementation may be found in the file **special_forms.icn**.

In addition to the two fundamental instructions above, a third is included, **macro_expand** to display the s-expression generated by the macro without evaluating it. This instruction can prove invaluable when you are initially defining a new macro and then testing it for correctness.

Modifications to Special Forms

```
procedure lambda (args)
# creates a lambda form -- anonymous function
# lambda forms will evaluate actual arguments

   (_length(args) >= 1) | fail
   formals := _first(args)
   body := _rest(args)
   scope := *environment
   return lambda_form(formals,body,scope)
end

procedure macro (args)
# creates a macro form -- anonymous function
# macro forms will not evaluate actual arguments

   (_length(args) >= 1) | fail
   formals := _first(args)
   body := _rest(args)
   scope := *environment
   return macro_form(formals,body,scope)
end

procedure macro_expand (args)
# applies macro form to supplied args to build result
# but macro-expand does not evaluate result

   (_length(args) = 1) | fail
```

```
   name := _first(_first(args))
   _macro_formp(macro := _eval(name)) | fail
   actual_args := _rest(_first(args))
   return _procedure_evaluation(macro,actual_args)
end
```

26.4 Quasiquote and Unquote

The final section in this chapter discusses several important special forms that play a significant role in our implementation of the **skeme** programming language.

The concepts of **quasiquote** and **unquote** provide the basic *cut-and-paste* features I described earlier with macro forms. To truly experience macro forms, it is essential to implement these features.

quasiquote

Quasiquote (‘) is similar to **quote** (’). The s-expression which immediately follows is to be unevaluated. However, while **quote** leaves the following expression totally untouched, **quasiquote** allows for some selective evaluation, that is **unquote**, within the item immediately following.

unquote

Unquote (,) indicates that the following item is an exception and should be evaluated! Furthermore, the evaluated item is to be inserted in place either as is, whether as an atom or as a list.

unquote-splicing

Unquote-splicing (,@) indicates that the following item should be evaluated! However, the evaluated item is not to be inserted in place as above; rather the item is to be absorbed as elements within the current list structure (appended in place).

Examples

```
> (define x '(1 2 3))
> (define y '(a b c d e))
> (define z '(#t #f))
> x
'(1 2 3)
> y
'(a b c d e)
> z
'(#t #f)
> `(x y z)
'(x y z)
> `(x ,y ,@z)
'(x (a b c d e) #t #f)
> `(x (,y ,@x) (,y ,@y) ,z ,@z)
'(x ((a b c d e) 1 2 3) ((a b c d e) a b c d e) (#t #f) #t #f)
```

On the page which follows is a bit more **Icon** code than I would normally provide. But the next page illustrates a technique of improving/enhancing a programming language by defining new elements in terms of elements that already exist within the language.

This technique is referred to as **syntactic sugar**. And LISP-like languages utilize this facility to the max!

So, how exactly does **quasiquote** work?

Experiment with the following substitution rules on several quasiquote s-expressions of your own design. Compare the results of your substitution s-expression with those from **mit-scheme** or **drracket**.

For individual items:

‘ x	$\Longrightarrow$	’x (same as quote – unevaluated!)
‘ ,x	$\Longrightarrow$	x (evaluated!)
‘ ,@x	$\Longrightarrow$	not defined

For lists of items:

‘ (... x ...)	$\Longrightarrow$	(... (’x) ...) (insert unevaluated list!)
‘ (... ,x ...)	$\Longrightarrow$	(... (x) ...) (insert evaluated!list)
‘ (... ,@x ...)	$\Longrightarrow$	(... x ...) (append evaluated list!)

Once you have a good understanding for what these substitutions are doing, look at the code on the following page.

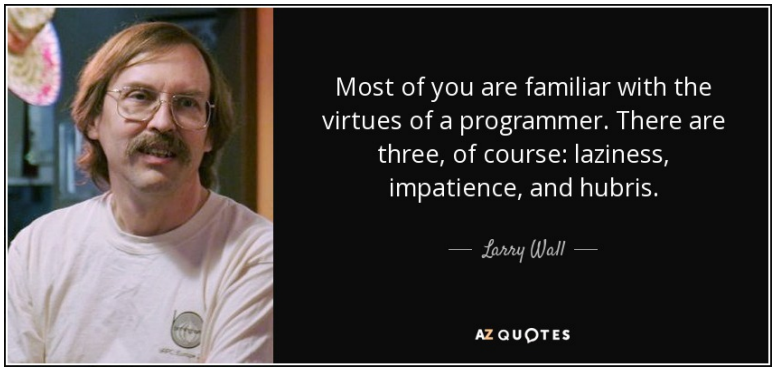
quasiquote_conversion transforms a quasiquote expression into an equivalent s-expression using only **append**, **list**, and **quote**! The final step in the process is to evaluate the newly-created s-expression.

In this section we have illustrated the implementation of syntactic sugar at the lowest possible level – combining previous primitives and special forms to define a new special form. The next chapter demonstrates that syntactic sugar can be implemented at higher levels as well.

Modifications to Special Forms (cont'd)

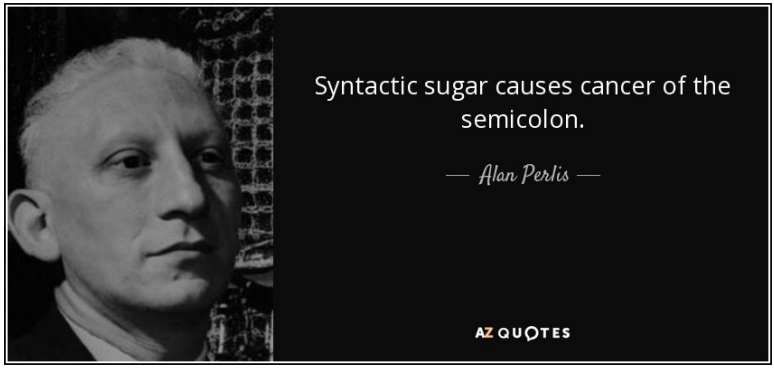
```
procedure quasiquote (args)
# scheme special form quasiquote; similar to quote
# but scans item for elements to unquote, i.e., evaluate!

  return _eval(quasiquote_conversion(args))
end

procedure quasiquote_conversion (args)
# this is very basic form of quasiquote:
#     - no nesting of quasiquoting
#     - both unquote and unquote-splicing are permitted

  (_length(args) = 0) & return []
  item := _first(args)
  if (_atomp(item)) then
    return [symbol("quote"),item]
  else if (unquotep(item)) then
    return _first(_rest(item))
  else if (unquote_splicingp(item)) then
  {
    write("undefined unquote-splicing context")
    fail
  }
  else
    return [symbol("append")] ||| unquote_scan (item)
end

procedure unquotep (item)
# predicate procedure: is the item "unquote"

procedure unquote_splicingp (item)
# predicate procedure: is the item "unquote-splicing"

procedure unquote_scan (args)
# scan list of args for "unquote" or "unquote-splicing"

  _nullp(args) & return []
  return [unquote_item(_first(args))] |||
         unquote_scan(_rest(args))
end

procedure unquote_item (item)
# scan a single item within unquote_scan

  if (_atomp(item)) then
    return [symbol("list"),[symbol("quote"),item]]
  else if (unquotep(item)) then
    return [symbol("list"),_first(_rest(item))]
  else if (unquote_splicingp(item)) then
    return _first(_rest(item))
  else
```

```
    return [symbol("list"),[symbol("append")] |||
           unquote_scan(item)]
end
```


Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris.
— Larry Wall —
AZ QUOTES

Chapter 27

Syntactic Sugar



Syntactic sugar is a euphemism for augmenting the syntax of a language by combining already existing components in new ways to accomplish a desired task.

In other words, the "new" feature is really available already in the language. But now we have a simpler, more concise way to express it. Ever since its inception, LISP has been notorious for enabling flexibility. If you do not like some specific feature within the language, then you can replace it with something else that you prefer!

Macro forms open the door to this cool new world!

Please refer to the scheme input file **init.scm** found in **Chapter 29: The End Result** for the implementation of the following elements of syntactic sugar.

- **cond**
- **let**
- **let***
- **letrec**
- **and** and **or** (short-circuit logic)
- **when** and **unless**
- **while** and **until**

Many **scheme** interpreters look for a loadable file **init.scm** upon startup. If found, the file is read and becomes part of the current environment. Since syntactic sugar is convenience to the user, I have chosen to implement these components not as additional primitives or special forms but rather as an application of macro forms.

Whatever the implementation of **scheme** you may be working with syntactic sugar provides an opportunity to customize the language to your preferences. Macro forms can be a bit quirky and hard to work with at times! But once successfully created they can simplify one programming experience significant.

Whatever the syntax that must be mastered in your particular version of **scheme** – either *macro* or *syntax-rules* – understanding and using macro forms is definitely worthwhile.

27.1 cond

Cond is **lisp**'s or **scheme**'s alternative to a **case** / **selection** statement in procedural languages.

syntax for cond

(**cond**
    (test$_1$ statement_list$_1$)
    (test$_2$ statement_list$_2$)
      :
    (**else** statement_list$_n$)
)

The keyword **else** on the last line could also be the boolean value **#t**. It is considered good programming practice to include a default statement list at the end of a **cond** statement.

Like in procedural languages, **cond** can be easily implemented as *nested* **if** statements.

implementation

(**if** test$_1$
    (**begin** statement_list$_1$)
    (**if** test$_2$
        (**begin** statement_list$_2$)
        (**if** test$_3$
              :
            (**if else** / **#t**
                (**begin** statement_list$_n$)) ...)))

27.2 let

Let is a LISP alternative to its very own **lambda** statement! The purpose for **let**, and its siblings **let*** and **letrec**, is to create new local environments within the existing program structure and to execute statements within that local environment.

Consider the following two items:

```
( ( lambda ( x y ) ( * ( + x y ) ( - x y ) ) ) 7 ( - 5 2 ) )
```

The above statement executes an anonymous lambda form on the two data values 7 and 3, ultimately returning the result 40.

```
( let ( ( x 7 ) ( y ( - 5 2 ) ) ) ( * ( + x y ) ( - x y ) ) ) )
```

The above two statements are ***equivalent***!

The second statement above is our first example of a **let** statement which defines (1) a collection of local variables within the the let statement (with any optional defined values evaluated within in the **current** environment) and (2) a sequence of executable statements.

syntax for let

```
( let
    ( (symbol_1 value_1 )
      (symbol_2 value_2 )
          :
      (symbol_n value_n ) )
    statement_list )
```

It is important to note that the let symbol definitions are done in the **current** environment and not in the **local let** environment. What that means in English is this: even though it may appear that you just defined a new symbol **x** to have the value 7, you can not use **x** in the definition for any subsequent new symbols, for example **y**!

That is the reason why we will shortly introduce you to **let**'s brother – **let*** . But before doing so, we first describe how **let** may be implemented using **lambda**.

implementation

((**lambda** (symbol_1 ... symbol_n)
    statement_list)
        value_1 ... value_n)

27.3 let*

The first thing an astute reader should notice is that the syntax for **let** and **let*** are identical!

syntax for let*

(**let***
    ((symbol_1 value_1)
      (symbol_2 value_2)
          :
      (symbol_n value_n))
    statement_list)

The difference in the two statements is in the handling of the definitions for the local symbols. **Let** does not permit the use of a *previously defined* local symbol in the evaluation of a *subsequently defined* local symbol. **Let*** permits this behavior.

The **scheme** programmer has a bit more flexibility and possibly a more natural transfer of information into the local scope of a **let** statements with this second form.

However, it is not an absolutely necessary addition to the language.

Consider the following two items:

(let * ((x 7) (y (- x 2))) (* (+ x y) (- x y))))

The above let* statement creates a let environment in which a new variable x is first defined and then that new variable is used to define a second variable y in the same let environment. Now consider the following:

(let ((x 7)) (let (y (- x 2))) (* (+ x y) (- x y)))))

The above statement is accomplishes **exactly** the same thing by **nesting** let statements within one another.

implementation

(**let** ((symbol_1 value_1))
    (**let** ((symbol_2 value_2))
          :
        (**let** ((symbol_n value_n))
            statement_list) . . .))

And now we need to turn the page again and meet **let**'s other brother – **letrec**.

27.4 letrec

Once again, an astute reader should notice that the syntax for all three let statements – **let**, **let***, and **letrec** – are identical!

syntax for letrec

(**letrec**
  ((symbol$_1$ value$_1$)
    (symbol$_2$ value$_2$)
      :
    (symbol$_n$ value$_n$))
  statement_list)

The primary use for **letrec** is to define local functions in the symbol-value list that are mutually recursive, i.e., symbol$_1$ may reference symbol$_2$ and symbol$_2$ may in turn reference symbol$_1$,

The **let** statement can not reference a previously defined symbol in the symbol-value list; the **let*** statement can reference a previously defined symbol only; the **letrec** statement can reference both a previously defined symbol and a subsequently defined symbol in the symbol-value list!

illustrative example

letrec example

```
(letrec ((is-even? (lambda (n)
                     (or (zero? n)
                         (is-odd? (dec n)) ) ))
         (is-odd?  (lambda (n)
                     (and (not (zero? n))
                          (is-even? (dec n)) ) )) )
  ( is-odd? 11) )
```

implementation

(**let**
  ((symbol$_1$ "not defined") ... (symbol$_n$ "not defined"))
    (set! symbol$_1$ value$_1$)
      :
    (set! symbol$_n$ value$_n$)
      statement_list)

We can now close the book on **let**, his brother **let***, and his other brother **letrec**. I feel like I have been watching reruns of the Bob Newhart television show!

Hello! I'm Larry.
This is my brother, Daryl.
This is my other brother, Daryl.

27.5 miscellaneous

The following items may also be found implemented in the loadable file **init.scm**.

27.5.1 and, or, not

The three instructions **and**, **or**, and **not** are obviously integral to any **scheme** interpreter. I have chosen to delay their implementation until now for two reasons:

- implementing short-circuit evaluation within the **icon** programming language is a bit more detailed
- implementing as a macro form in **scheme** is both illustrative of its power and trivial!

27.5.2 when and unless

These two instructions provide conditional execution of a single **scheme** statement.

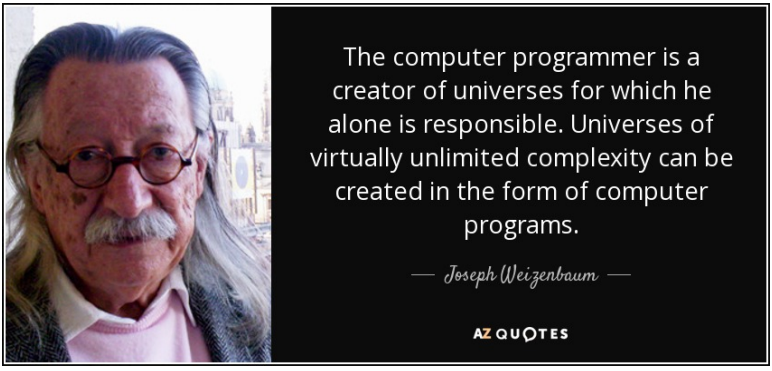
The first will execute the given statement only if
the conditional expression evaluates to **#t**.
The second will execute the given statement only if
the conditional expression evaluates to **#f**.

27.5.3 while and until

These two instructions provide conditional repetitive execution of a group of **scheme** statements.

The first will execute the statement block only while
the conditional expression evaluates to **#t**.
The second will execute the statement block only while
the conditional expression evaluates to **#f**.

The first is a *top-tested loop*, i.e., conditional expression is evaluated *before* the statement block is executed.
The second is a*bottom-tested loop*, i.e., conditional expression is evaluated *after* the statement block has executed.


The computer programmer is a creator of universes for which he alone is responsible. Universes of virtually unlimited complexity can be created in the form of computer programs.
— Joseph Weizenbaum —
AZ QUOTES

Chapter 28

What's Next? What's Not Next!

Here we are at the very end of the book and I throw a handful of new topics at you! I consider this chapter not so much as a conclusion but rather a transition into a new and interesting world.

Almost everything we have considered in Part Four has had an immediacy to it: define a variable, define a lambda form, execute a lambda form, ... The argument transfer technique *call by name* is possibly the sole exception. Yes, we do pass information into the macro form; but, no, it is not evaluated **now**, it is to be evaluated **later** when the macro form terminates. And if I can delay evaluation of arguments until later can I delay the evaluation of other components as well?

In this chapter we introduce the reader to the concept of a **promise** and the procedures that implement delayed-evaluation of expressions:

- delay
- force
- promise?
- promise-forced?
- promise-value

A **promise** in a nutshell is a commitment to execute a specific expression at some unspecified time in the future. A promise essentially has two components: the first is a **scheme** expression and the second is a value (initially undefined). Later, when the promise has been kept, the expression is no longer necessary and the value is defined to be the evaluated expression.

Note: A promise is executed at most once. If a promise is executed repeatedly, then the original value is returned each time.

28.1 promises and delayed evaluation

At this point in our discussion, it is quite possible and also quite tempting to implement promises as syntactic sugar! However, I have chosen not to do that for two reasons.

The first reason is that I find functional programming attractive because of its mathematical nature and its proximity to the foundations of computing. These topics I find intriguing because of their logical complexity and always being on the edge of a paradox. But, just because I am intrigued with being on the edge, that does not justify me taking you with me if some of our earlier discussion proves a bit flakey.

The second reason is the concept of a promise seems very straightforward and easily implemented in **icon**. I would rather return to the core building blocks to develop the new concept.

Whether one chooses to implement promises as syntactic sugar or as part of the core, either way it will open an interesting new world of applications to explore!

28.1.1 delay and force

The **scheme** syntax for creating a promise is the s-expression

(**delay** <expression>)

or more likely in practice

(**define** <promise-name> (**delay** <expression>)

It is important to remember that a promise remains unevaluated until it is forced some time in the future.

It is also important to remember that a promise may be forced repeatedly in the future, but in such a situation an identical value is returned each time – namely the value evaluated at the time of the initial force.

28.1.2 three supporting procedures

In addition to the fundamental **delay** and **force** procedures, **scheme** typically provides the following three procedures or their equivalent: **promise?**, **promise-forced?**, and **promise-value**. All three procedures are fairly obvious in what they do.

promise? is a predicate which returns #t if the expression is a promise. **promise-forced?** is also a predicate which returns #t if the expression has already been forced. And **promise-value** returns resulting value determined when promise was first forced.

The advantage of implementing delayed evaluation and later forcing evaluation on demand is not immediately apparent – other than sounding pretty cool. The next section introduces two concepts that demonstrate the power of having this additional power at our fingertips.

28.2 streams

Lists are one of the core building blocks of functional programming. The entire programming language is essentially built around his concept. Even though a list may be arbitrarily long, it must be finite in length and all entries evaluated and available as the time of its creation.

Streams are very similar to lists. List are built by taking a single item and **cons**-ing with the rest of the list. Streams are built by taking a single item and **cons**-ing with a **promise** for the rest of the list!

(**cons** <first-expression> <rest-expression>)

versus

(**cons** <first-expression> (**delay** <rest-expression>)

28.2.1 lists versus streams

These two structures are closely related to one another. Operations on streams parallel almost exactly operations on lists:

- **the-empty-list** and **the-empty-stream**
- **empty-list?** and **empty-stream?**
- **list-cons** and **stream-cons**
- **list-car** and **stream-car**
- **list-cdr** and **stream-cdr**
- **list-ref** and **stream-ref**
- **list-map** and **stream-map**
- **list-filter** and **stream-filter**

The implementation for all of the above may be found in **Chapter 29: The End Result**.

Note: Fundamental list procedures have been discussed and defined previously. In this chapter I have chosen to prepend the symbol **list-** to emphasize the ***list*** version of the procedure versus a ***stream*** version of the similar procedure. However, both names for the list version are equivalent to one another.

list-cons and stream-cons

cons implementation

```
(define list-cons cons)
; i.e., list-cons is the primitive core function
; (cons x y)

(define stream-cons (macro (x y)
  `(cons ,x (delay ,y)) ))

;;; note: list-cons is a proper lambda form
;;; actual arguments are evaluated
;;; stream-cons is a macro form
;;; evaluating the promise during stream construction
;;;    defeats the purpose!
```

list-car and stream-car

car implementation

```
(define list-car car)
; i.e., list-car is the primitive core function
; (car x)
(define stream-car car)
; i.e., stream-car is the primitive core function
; (car x)

;;; both list-car and stream-car are proper lambda forms
;;; actual arguments are evaluated!
```

list-cdr and stream-cdr

cdr implementation

```
(define list-cdr rest)
; i.e., list-cdr is the primitive core function
; (cdr listx))

(define stream-cdr (lambda (streamx)
  (force (cdr streamx)) ))

;;; note: both list-cdr and stream-cdr are proper lambda forms
;;; however, stream-cdr must force (evaluate) the promise
;;;    list-cdr does not
```

list-map and stream-map

map implementation

```
(define list-map map)
; i.e. list-map is the procedure map defined earlier in the text
; it is equivalent to the following
(define list-map (lambda (proc listx)
  (if (list-null? listx)
      the-empty-list
      (list-cons (proc (list-car listx))
                 (list-map proc (list-cdr listx)) ) ) ))

(define stream-map (lambda (proc streamx)
  (if (stream-null? streamx)
      the-empty-stream
      (stream-cons (proc (stream-car streamx))
                   (stream-map proc (stream-cdr streamx)) ) ) ))

;;; note: list-map and stream-map are both proper lambda forms
;;; actual arguments are evaluated
```

28.2.2 infinite streams

This last section introduces a very intriguing topic – one very significant difference between the power of lists and the power of streams.

Consider a simple mathematical sequence:

$$a_1, a_2, a_3, a_4, a_5, \ldots$$

If this sequence were a **finite** sequence, then this sequence could be easily represented in **scheme** using either a list or a stream.

However, if this sequence were an **infinite** sequence, then a list representation is out of the question! A stream representation is **not** guaranteed, but remains a possibility.

If the next item in the sequence a_n can be determined by its predecessors $a_1, a_2, a_3, \ldots, a_{n-1}$, then a stream representations is an option!

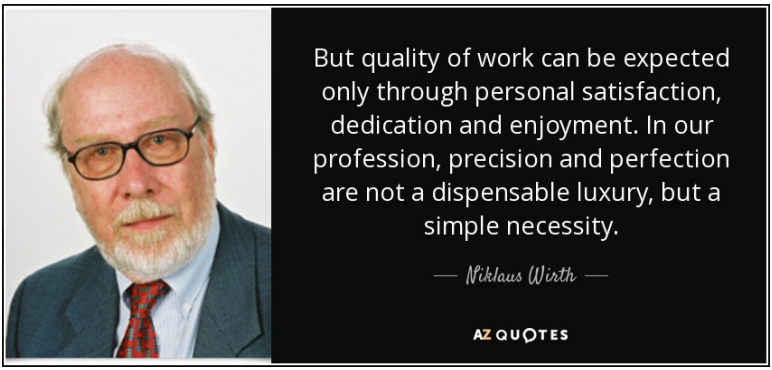
We will consider in order the following examples:

- constant sequence
- natural numbers
- alternating sequence
- fibonacci numbers

infinite streams

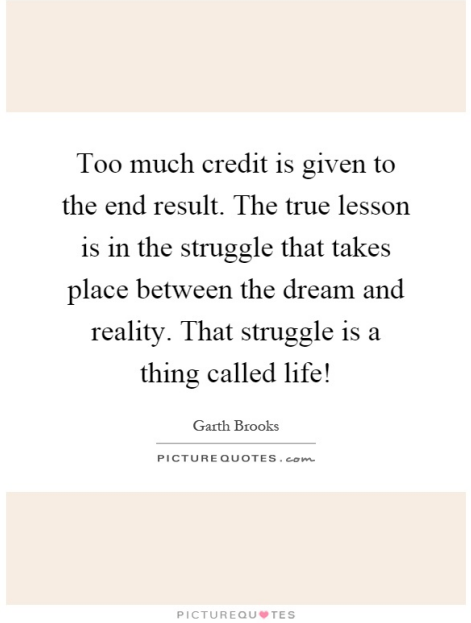
```
;;; introduction to infinite streams

(define const-next (macro (a)
  `(stream-cons ,a (const-next ,a)) ))
(define ones (const-next 1))
(define twos (const-next 2))
(define threes (const-next 3))

(define integers-starting-from (macro (n)
  `(stream-cons ,n (integers-starting-from ,(+ n 1))) ))
(define nats (integers-starting-from 1))

(define alt-next (macro (a)
  `(stream-cons ,a (alt-next ,(- a))) ) )
(define alts (alt-next 1))
```

```
(define fib-next (macro (a b)
  `(stream-cons ,a (fib-next ,b (+ ,a ,b))) ) )
(define fibs (fib-next 1 1))
```


But quality of work can be expected only through personal satisfaction, dedication and enjoyment. In our profession, precision and perfection are not a dispensable luxury, but a simple necessity.
— Niklaus Wirth —
AZ QUOTES

Chapter 29

The End Result



29.1 core: low level components

The **core.icn** file focuses on the implementation of the basic building blocks within **scheme** using the **icon** programming language.

core.icn

```
# ------------------------------------------------------------ #
#
#
#    core building block components for skeme
#
#
#
# ------------------------------------------------------------ #

# basic atoms
# integer       icon integer
# real          icon real
# char          icon cset
# string        icon string
record          boolean (value)
record          symbol (identifier)

# basic lists
# lists in Lisp/Scheme are typically built
#    using "cons" constructor which create
#    what are called "dotted_pairs"
# lists in skeme are what would normally be
#    referred to as "proper_lists"
#    in other implementations

# pair          has a very limited role in skeme
#               a pair is NOT the fundamental constructor
#                    as it is in Lisp
#               a pair may be used for an association list
#                  i.e., a list of actual pairs!
record          dotted_pair (first,rest)

# list          represented as an icon list
#               in Lisp (a b c d e f) would be represented
#                  [a . [b . [c . [d . [e . [f . []]]]]]]

# alist         short for "association_list"
#               a list comprised of dotted pairs!
#                  ( [a1 . b1] [a2 . b2] ... )
#               the first entry in the dotted pair
#                  often represents a key
#               the second entry in the dotted pair
#                  often represents the associated data

# basic procedures / functions
```

```
# builtin components:
#    primitive    -- arguments are all evaluated
#    special_form -- arguments are not evaluated
record          primitive (identifier)
record          special_form (identifier)

# user defined procedures
#    lambda_form  -- arguments are all evaluated
#    macro_form   -- arguments are not evaluated
record          lambda_form (formals,body,parent)
record          macro_form (formals,body,parent)

# promises = delayed evaluations of expressions
#    if promise has just been delayed
#       expression is defined and result is set to &null
#    if promise has been forced
#       result is defined and expression is set to &null
record          promise (expression,result)

#---------------------------------------------------------#
#
#
#          useful core predicates for atoms
#
#
#
#---------------------------------------------------------#

procedure _integerp (item)
  return (type(item) == "integer")
end

procedure _realp (item)
  return (type(item) == "real")
end

procedure _numberp (item)
  if (type(item) == ("integer" | "real")) then
    return
end

procedure _booleanp (item)
  return (type(item) == "boolean")
end

procedure _charp (item)
  return (type(item) == "cset")
end

procedure _stringp (item)
  return (type(item) == "string")
end
```

```
procedure _symbolp (item)
  return (type(item) == "symbol")
end

procedure _atomp (item)
  if (_numberp(item) | _booleanp(item) |
      _charp(item)   | _stringp(item)  |
      _symbolp(item) | _nullp(item)) then
    return
end

#-------------------------------------------------------------#
#
#
#          useful core predicates for lists
#
#
#
#-------------------------------------------------------------#

procedure _nullp (item)
  return ((type(item) == "list") & (*item = 0))
end

procedure _pairp (item)
  return (type(item) == "dotted_pair")
end

procedure _listp (item)
  return (type(item) == "list")
end

procedure _alistp (item)
  _listp(item) | fail
  _nullp(item) & return
  _pairp(_first(item)) & return _alistp(_rest(item))
  fail
end

#-------------------------------------------------------------#
#
#
#        useful core predicates for functions
#
#
#
#-------------------------------------------------------------#

procedure _primitivep (item)
  return (type(item) == "primitive")
end

procedure _special_formp (item)
```

```
  return (type(item) == "special_form")
end

procedure _lambda_formp (item)
  return (type(item) == "lambda_form")
end

procedure _macro_formp (item)
  return (type(item) == "macro_form")
end

procedure _procedurep (item)
  return (type(item) ==
    ("primitive"    | "special_form" |
     "lambda_form" | "macro_form"))
end

#----------------------------------------------------------- #
#
#
#       useful core predicates for promises
#
#
#
#----------------------------------------------------------- #

procedure _promisep (item)
  return (type(item) == "promise")
end

#----------------------------------------------------------- #
#
#
#          core predicates for "equality"
#
#
#
#----------------------------------------------------------- #

procedure _eqp (item1,item2)
# icon does not provide for address-like comparisons
# this very strict implementation of "equal"
#    is unavailable in icon
# in most cases, _eqvp is what people actually want

  return _eqvp(item1,item2)
end

procedure _eqvp (item1,item2)
# this procedure tests for "equal_in_value"
# for basic elements in the language

  (type(item1) == type(item2)) | fail
```

```
   if (type(item1) ==
     ("integer" | "real" | "cset" |"string")) then
       return (item1 == item2)
   else if (type(item1) == "boolean") then
     return (item1.value == item2.value)
   else if (type(item1) == "symbol") then
     return (item1.identifier == item2.identifier)
   else if (_nullp(item1)) then
     return _nullp(item2)
   else if (type(item1) ==
     ("primitive" | "special_form")) then
       return (item1.identifier === item2.identifier)
   else if (type(item1) ==
     ("lambda_form" | "macro_form")) then
       write ("unspecified_result") & fail
   else if (type(item1) == "promise") then
     write ("unspecified_result") & fail
   else
     write ("eqv?_does_not_apply_to_lists") & fail
end

procedure _equalp (item1,item2)
# this procedure tests for "equal_in_structure"
#      and "equal_in_corresponding_values"
#      for more complicated elements in the language

   (type(item1) == type(item2)) | fail
   if (type(item1) == "list") then
   {
     if (_nullp(item1)) then
       return (_nullp(item2))
     else if (_equalp(_first(item1),_first(item2))) then
       return _equalp(_rest(item1),_rest(item2))
     else
       fail
   }
   else
     return _eqvp(item1,item2)
end

# ------------------------------------------------------------ #
#
#
#           fundamental core functions
#
#
#
# ------------------------------------------------------------ #

procedure _cons (item1,item2)
# if item2 is proper list, cons returns proper list
#   with item1 prepended to the front of item2
   _listp(item2) & return ([item1]|||item2)
```

```
# if item 2 is not proper list, cons returns dotted pair
  return dotted_pair(item1,item2)
end

procedure _first (item)
# at this point in developing the interpreter
# i prefer to use the name "first" rather than "car"
  _atomp(item) & fail
  _listp(item) & (*item > 0) & return item[1]
  _pairp(item) & return item.first
end

procedure _rest (item)
# at this point in developing the interpreter
# i prefer to use the name "rest" rather than "cdr"
  _atomp(item) & fail
  _listp(item) & (*item > 0) & return item[2:0]
  _pairp(item) & return item.rest
end

procedure _length (listx)
# determine the length of proper list
  _listp(listx) | fail
  return *listx
end

procedure _reverse (listx)
# reverse the order of a proper list
  (_listp(listx)) | fail
  (_nullp(listx)) & return []
  return _reverse(listx[2:0]) ||| [listx[1]]
end

procedure _eval (item)
# remember the general principle: evaluate everything!

  # check for "self-evaluating" atoms
  _nullp(item) & return []
  (_numberp(item) | _charp(item) | _stringp(item) |
   _booleanp(item) | _procedurep(item)) &
    return item
  # check for atoms previously defined in the environment
  _symbolp(item) &
    return (search(item) | item.identifier||"_not_found!")
  # check for a valid s-expression to evaluate
  _listp(item) &
    return _apply(_eval(_first(item)),_rest(item))
  # all other items are returned unevaluated
  return item
end

procedure _apply (proc,args)
```

```
# func may be:
#     either a previously defined symbol
#     or an anonymous lambda form or macro form

# remember the general principle: evaluate everything!
#     "call_by_value" adheres to this principle
#     "call_by_name" does not

  _primitivep(proc) &
    return proc.identifier(_call_by_value(args))
  _special_formp(proc) &
    return proc.identifier(_call_by_name(args))
  _lambda_formp(proc) &
    return _procedure_eval(proc,_call_by_value(args))
  _macro_formp(proc) &
    return _eval(_procedure_eval(proc,_call_by_name(args)))
end

procedure _quote (item)
# exception to the general principle: evaluate everything!

  return item
end

#------------------------------------------------------------#
#
#
#             support utilities
#
#
#
#------------------------------------------------------------#

procedure _procedure_eval (proc,args)
# helper procedure to facilitate evaluation of
#    user-defined functions both lambda forms macro forms

  formals := proc.formals       # proper list formal args
  executables := proc.body
  parent := proc.parent
  actuals := args               # proper list actual args
  # save the current environment
  # and extend the environment for the lambda/macro forms
  previous_environment := current_environment
  current_environment := environment (table(),parent)
  _define_arguments(formals,actuals)
  result := begin(executables) # using begin special form
  # reinstate the original environmnet
  current_environment := previous_environment
  return result
end

procedure _define_arguments (formals,actuals)
```

```
# helper procedure to transfer actual arguments
#    to the corresdponding formal arguments
# _  for both lambda/macro forms

   if (_symbolp(formals)) then
   {
     # insert variable-value pair into the new  environment
     insert_environment(formals,actuals)
     return
   }
   _nullp(formals) & _nullp(actuals) & return
   if (_length(formals) = _length(actuals)) then
   {
     formal_variable := _first(formals)
     actual_value := _first(actuals)
     _symbolp(formal_variable) | fail
     # insert variable-value pair into the new environment
     insert_environment(formal_variable,actual_value)
     return _define_arguments(_rest(formals),_rest(actuals))
   }
end

procedure _call_by_name (args)
# do NOT evaluate actual args within current environment
# formal arguments assigned actual argument expressions
# note: _call+by_name is essentially a pass through

   return args
end

procedure _call_by_value (args)
# evaluate actual args within current environment
# formal arguments assigned actual argument evaluations

   _nullp(args) & return []
   return [_eval(_first(args))] |||
     _call_by_value(_rest(args))
end

# ------------------------------------------------------ #
```

29.2 skeme: the main program

The **skeme.icn** file is a fairly small main driver program implementing the read-eval-print-loop fundamental to **scheme**.

skeme.icn

```
# ------------------------------------------------------------ #
#
#
# skeme.icn -- the main driver for the skeme interpreter
#
#              read-eval-print loop
#
#
#
# ------------------------------------------------------------ #

# _gensym is a symbol generator
global _gensym

# global variables for scheme REPL elements
global skeme_input
global input_source
global quasiquote_flag

# icon csets for lexical analysis of skeme_input
global white_space
global taboos
global identifier_chars
global relops
global arithops

# icon csets for identifying specific ASCII symbols
global single_quote
global double_quote
global quasi_quote
global back_slash
global comma
global hash
global left_paren
global right_paren
global period
global eol

# ------------------------------------------------------------ #

procedure main ()
# main procedure for skeme: read-eval-print-loop (REPL)

   _gensym := create ("!S"||seq())
   initialize_symbols()
```

```
   initialize_environment()
   write("welcome to skeme")
   write("a scheme interpreter written in icon/unicon")
   load(["init.scm"])
   while (1) do
      repl()
end

#---------------------------------------------------------- #

procedure initialize_symbols ()
# initialize global variables

   input_source            := &input
   quasiquote_flag         := &null

   white_space             := ' \t'
   taboos                  := '()[]{}.,;\"\'`|#'
   identifier_chars        := &ascii--taboos--white_space
   relops                  := '=><'
   arithops                := '+-*/'

   single_quote            := '\''          # quote
   double_quote            := '\"'
   quasi_quote             := '`'           # quasiquote
   back_slash              := '\\'
   comma                   := ','           # unquote
   splice                  := '@'           # unquote splice
   hash                    := '#'
   eol                     := ';'           # skeme eol-marker
   left_paren              := '('
   right_paren             := ')'
   period                  := '.'
end

#---------------------------------------------------------- #

procedure repl ()
# the fundamenta read-eval-print loop
# comparable to fetch-decode-execute in machine language

#      _read  is implemented in source file translator.icn
#      _print is implemented in source file translator.icn
#      _eval  is implemented in source file core.icn

   _print(_eval(_read()))
   write()
end

#---------------------------------------------------------- #

procedure Error (message)
# this is a trivial error handler
```

```
# just crash and burn with a simple message

  stop(message)
end

# ------------------------------------------------------------ #
```

29.3 translator: read and print

The **translator.icn** file focuses specifically on the implementation of the two important primitives: read and print. Read focuses on converting a stream of ASCII characters into **scheme** s-expressions. Print focuses on displaying s-expressions in a more readable human format. Eval, of the read-eval-print loop, is to be found **core.icn** and again in **primitives.icn**.

Both read and print are deceptively longer than might be expected, due to the number of possible **scheme** data types that must be considered on input and output.

translator.icn

```
# ---------------------------------------------------------- #
#
#
#       translator.icn:
#
#             convert between human readable
#
#             and skeme internal representations
#
#
#
# ---------------------------------------------------------- #

procedure _read ()
# processes an input stream (s-expression) of skeme items
# to return a valid skeme internal representation

   initialize_stream() | fail
   return read_item()
end

# ---------------------------------------------------------- #

procedure read_item ()
# read_item determines the category of item to be read
# an atom or a list

   if (skeme_input ? any(eol)) then
     append_stream() | fail
   if (current_is(left_paren)) then
     return read_list()
   else
     return read_atom()
end

# ---------------------------------------------------------- #
```

```
procedure read_atom ()
# read the value of a skeme atom
# and convert it to its internal representation

   if (skeme_input ? any(eol)) then
     append_stream() | fail
   if (skeme_input ? any(hash)) then
     return read_hash()
   else if (skeme_input ? any(single_quote)) then
     return read_quote()
   else if (skeme_input ? any(double_quote)) then
     return read_string()
   else if (skeme_input ? any(quasi_quote)) then
     return read_quasiquote()
   else if (skeme_input ? any(comma)) then
     return read_unquote()
   else if (skeme_input ? any(arithops)) then
     return read_arithops()
   else if (skeme_input ? any(relops)) then
     return read_relops()
   else if (skeme_input ? any(&digits)) then
     return read_number()
   else if (skeme_input ? any(identifier_chars)) then
     return read_symbol()
#  else
#    Error("what is this (" || skeme_input[1] || ")")
end

# ------------------------------------------------------------ #

procedure read_list ()
# read the value of a skeme list
# and convert it to its internal representation

   if (skeme_input ? any(eol)) then
     append_stream() | fail
   if (current_is(right_paren)) then
     return []
   first_item := read_item()
   rest_items := read_list()
   return [first_item] ||| rest_items
end

# ------------------------------------------------------------ #

procedure _print (x)
   /x & fail
   print_item(x)
   return x
end

# ------------------------------------------------------------ #
```

```
procedure print_item (x)
# print_item determines the tye of item to be displayed

   if (type(x) == "list") then
     print_list(x)
   else if (type(x) == "dotted_pair") then
   {
     writes("[")
     print_item(x.first)
     writes(".")
     print_item(x.rest)
     writes("]")
   }
   else if (type(x) == "primitive") then
     writes("<primitive>")
   else if (type(x) == "special_form") then
     writes("<special_form>")
   else if (type(x) == "lambda_form") then
   {
     writes("<lambda_form: ")
     print_item(x.formals)
     writes(":")
     print_item(x.body)
     writes(":")
     print_item(x.parent)
     writes(">")
   }
   else if (type(x) == "macro_form") then
   {
     writes("<macro_form: ")
     print_item(x.formals)
     writes(":")
     print_item(x.body)
     writes(":")
     print_item(x.parent)
     writes(">")
   }
   else if (type(x) == "promise") then
   {
     writes("<promise: ")
     if (\x.expression) then
       print_item(x.expression)
     else
       writes("na")
     writes(" , ")
     if (\x.result) then
       print_item(x.result)
     else
       writes("na")
     writes(">")
   }
   else if (_atomp(x)) then
```

```
    print_atom(x)
end

#---------------------------------------------------------------#

procedure print_atom (x)
# display the value of a skeme atom in a readable form

   if (_integerp(x)) then
     writes(x)
   else if (_realp(x)) then
     writes(x)
   else if (_stringp(x)) then
     writes("\"" || x || "\"")
   else if (_charp(x)) then
     writes("#\\" || x)
   else if (_booleanp(x)) then
     writes(x.value)
   else if (_symbolp(x)) then
     writes(x.identifier)
   else
     Error("unrecognized atom: " || type(x))
end

#---------------------------------------------------------------#

procedure print_list (x)
# print_item determines the type of item to be displayed
# an atom or a dotted pair or a procedure

   if (_nullp(x)) then
     writes ("()")
   else
   {
     writes("(")
     every (i := 1 to *x) do
     {
       print_item(x[i])
       if (i < *x) then
         writes(" ")
       else
         writes(")")
     }
   }
end

#---------------------------------------------------------------#

# the first two supporting procedures
#    initialize_stream and append_stream
# enable the interpreter to consider the input stream
# as an unbounded sequence of ASCII symbols
```

```
# the eol symbol (;) is a convenience
#    to recognize the carriage return
#    and, if necessary, append to the input stream

# n.b. reverse-trim-reverse-trim
#    eliminates any leading and trailing white space!

#---------------------------------------------------------- #

procedure initialize_stream ()
# input_source initially is stdin
# however, we may redirect reading input from a file

   if (input_source === &input) then
   {
     writes("skeme>␣")
     (line := read()) | fail
   }
   else
     (line := read(input_source)) | fail
   skeme_input :=
     reverse(trim(reverse(trim(line),white_space)),
       white_space) || eol
   if (skeme_input ? any(eol)) then
     append_stream() | fail
   return
end

procedure append_stream ()
# input_source initially is stdin
# however, we may redirect reading input from a file

   if (input_source === &input) then
   {
     writes("more>␣␣")
     (line := read()) | fail
   }
   else
     (line := read(input_source)) | fail
   skeme_input :=
     reverse(trim(reverse(trim(line),white_space)),
       white_space) || eol
   if (skeme_input ? any(eol)) then
     append_stream() | fail
   return
end

#---------------------------------------------------------- #

# the balance of this file contains icon source code
# for the lexical analysis of the input stream
# each component focuses on one specific building block
```

```
# ------------------------------------------------------------ #

procedure current_is (char)
# recognizes a specific character
# and removes it from the stream

   if (skeme_input[1] == eol) then
     append_stream() | fail
   skeme_input ?:= {
     tab(any(char)) | fail
     tab(many(white_space))
     tab(0)
     }
   return
end

# ------------------------------------------------------------ #

procedure read_hash ()
# recognizes the boolean values #t and #f
# recognizes ASCII character values #\<char>

   skeme_input ?:= {
     result := tab(any(hash))
     result ||:= tab(any('tf'))
     if (pos(3)) then
       result := boolean(result)
     else
     {
       tab(any(back_slash))
       if (pos(3)) then
       {
         result := move(1)
         result := cset(result)
       }
       else
         stop ("invalid use of # character")
     }
     tab(many(white_space))
     tab(0)
     }
   return result
end

# ------------------------------------------------------------ #

procedure read_string ()
# the input of strings is very basic
# no backslash character are as an escape character
# the next double quote terminates the string!

   skeme_input ?:= {
     move(1)
```

```
    result :=
      tab(bal(double_quote,double_quote,double_quote))
    move(1)
    tab(many(white_space))
    tab(0)
    }
  return result
end

#---------------------------------------------------------- #

procedure read_quote ()
# recognize the ASCII single quote character
# and replace it with the skeme symbol quote

  skeme_input ?:= {
    move(1)
    tab(many(white_space))
    tab(0)
    }
  return [symbol("quote")] ||| [read_item()]
end

#---------------------------------------------------------- #

procedure read_quasiquote ()
# recognize the ASCII back quote character
# and replace it with the skeme symbol quasiquote

  quasiquote_flag := "true"
  skeme_input ?:= {
    move(1)
    tab(many(white_space))
    tab(0)
    }
  result := [symbol("quasiquote")] ||| [read_item()]
  quasiquote_flag := &null
  return result
end

#---------------------------------------------------------- #

procedure read_unquote ()
# recognize the ASCII comma character
# and replace it with
#    either the skeme symbol unquote
#    or the skeme symbol unquote splicing

  if (\quasiquote_flag) then
  {
    skeme_input ?:= {
      (=",@" & result := "unquote-splicing") |
      (tab(any(',')) & result := "unquote")
```

```
      tab(many(white_space))
      tab(0)
      }
    return [symbol(result)] ||| [read_item()]
  }
  else
    Error("unquote_only_allowable_within_quasiquote!")
end

#---------------------------------------------------------- #

procedure read_arithops ()
# recognize the ASCII arithmetic operators
# replaces them with skeme symbols add, sub, mul, and div
# however, plus and minus may also be a unary operation
# on a number immediately following!

  skeme_input ?:= {
    (tab(any('+')) & result := "add") |
    (tab(any('-')) & result := "sub") |
    (tab(any('*')) & result := "mul") |
    (tab(any('/')) & result := "div")
    tab(0)
    }
  if ((result == "add") &
      (skeme_input ? tab(any(&digits)))) then
    return read_number()
  else if ((result == "sub") &
      (skeme_input ? tab(any(&digits)))) then
    return read_number("neg")
  skeme_input ?:= {
    tab(many(white_space))
    tab(0)
    }
  return symbol(result)
end

#---------------------------------------------------------- #

procedure read_relops ()
# recognize the ASCII relational operators
# replaces them with skeme symbols
#    n_eq, n_gt, n_ge, n_lt, and n_le
# n_ indicates "numeric" to distinguish from:
# string comparison", character comparison, ...

  skeme_input ?:= {
    (=">=" & result := "n_ge") |
    (="<=" & result := "n_le") |
    (tab(any('=')) & result := "n_eq") |
    (tab(any('>')) & result := "n_gt") |
    (tab(any('<')) & result := "n_lt")
    tab(many(white_space))
```

```
    tab(0)
    }
  return symbol(result)
end

# ---------------------------------------------------------- #

procedure read_number (neg)
# recognize a numeric value
#   - integer: {+|-} digit+
#   - real:    {+|-} digit+ [.digit*] [E {+|-} digit+]

  if (\neg) then
    result := "-"
  else
    result := ""
  skeme_input ?:= {
    result ||:= tab(any(&digits))
    result ||:= tab(many(&digits))
    result ||:= tab(any('.'))
    result ||:= tab(many(&digits))
    result ||:= tab(any('eE'))
    result ||:= tab(any('+-'))
    result ||:= tab(many(&digits))
    tab(many(white_space))
    tab(0)
    }
  if (find(".",result) |
      find("e",result) |
      find("E",result)) then
    return real(result)
  else
    return integer(result)
end

# ---------------------------------------------------------- #

procedure read_symbol ()
# recognize a scheme symbol

  skeme_input ?:= {
    result := tab(many(identifier_chars))
    tab(many(white_space))
    tab(0)
    }
  return symbol(result)
end

# ---------------------------------------------------------- #
```

29.4 primitives: initial procedures

The **primitives.icn** file contains mid-level implementations of **scheme** building block functions. **Icon** argument lists are no longer utilized for transferring formal arguments; rather **scheme** lists are the preferred mechanism. This is a good initial step toward programming with a functional programming mindset.

primitives.icn

```
# ------------------------------------------------------------ #
#
#
#              primitive predicates
#
#
#
# ------------------------------------------------------------ #

# the following predicate procedures all adhere to:
#     - args should be a list of a single item
#     - use the core.icn version of the predicate
#     - convert the resulting success/fail to true/false

procedure integerp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_integerp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure realp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_realp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure numberp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_numberp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure charp (args)
```

```
  (_length(args) = 1) | fail
  item := _first(args)
  if (_charp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure stringp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_stringp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure booleanp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_booleanp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure symbolp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_symbolp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure atomp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_atomp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure nullp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_nullp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end
```

```
procedure pairp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_pairp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure listp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_listp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure alistp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_alistp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure primitivep (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_primitivep(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure special_formp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_special_formp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure lambda_formp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_lambda_formp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end
```

```
procedure macro_formp (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_macro_formp(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure procedurep (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_procedurep(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure promisep (args)
  (_length(args) = 1) | fail
  item := _first(args)
  if (_promisep(item)) then
    return boolean("#t")
  else
    return boolean("#f")
end

#-------------------------------------------------------------#
#
#
#            building block primitives
#
#
#
#-------------------------------------------------------------#

# the cornerstone scheme building blocks:
#     - cons
#     - first (car)
#     - rest (cdr)
#     - eval
#     - apply
#     - quote

procedure cons (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
  item2 := _first(_rest(args))
  return _cons(item1,item2)
end

procedure first (args)
```

```
  (_length(args) = 1) | fail
  return _first(_first(args))
end

procedure rest (args)
  (_length(args) = 1) | fail
  return _rest(_first(args))
end

procedure eval (args)
  (_length(args) = 1) | fail
  item := _first(args)
  # the single argument to eval must be an s-expression
  _listp(item) | fail
  return _eval(item)
end

procedure apply (args)
  (_length(args) = 2) | fail
  proc := _first(args)
  # the first argument is a procedure
  _procedurep(proc) | fail
  arguments := _first(_rest(args))
  # the second argument is a list of actual arguments
  _listp(arguments) | fail
  return _apply(proc,arguments)
end

procedure quote (args)
  (_length(args) = 1) | fail
  item := _first(args)
  # the single argument to quote could be anything!
  # no need to call _quote in core.icn!
  return item
end

#-------------------------------------------------------------#
#
#
#                  equality predicates
#
#
#
#-------------------------------------------------------------#

# the following predicate procedures all adhere to:
#     - args should be a list of two items
#     - use the core.icn version of the predicate
#     - convert the resulting success/fail to true/false

procedure eqp (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
```

```
  item2 := _first(_rest(args))
  if (_eqp(item1,item2)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure eqvp (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
  item2 := _first(_rest(args))
  if (_eqvp(item1,item2)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure equalp (args)
  (_length(args) = 2) | fail
  item1 := _first(args)
  item2 := _first(_rest(args))
  if (_equalp(item1,item2)) then
    return boolean("#t")
  else
    return boolean("#f")
end

#----------------------------------------------------------------#
#
#
#               initial primitives
#
#
#
#----------------------------------------------------------------#

procedure force (args)
# force evaluation of a promise

  (_length(args) = 1) | fail
  item := _first(args)
  _nullp(item) & return []
  (_promisep(item)) | fail
# there is a bug right here!!!
  if (\item.result) then
    return item.result
  else
  {
    item.result := _eval(item.expression)
    item.expression := &null
    return item.result
  }
end
```

```
procedure gensym ()
# generate a unique scheme symbol upon request

  return symbol(@_gensym)
end

procedure load (args)
# process skeme input from a file
# rather than from stdin
# using a read-eval-butnoprint loop (REBL?)

  static LOADLIB

  LOADLIB := "/home/kaiserpa/prog_langs/skeme/libskeme/"
  (_length(args) = 1) | fail
  file_name := _first(args)
  _stringp(file_name) | fail
  if (fin := (open(file_name) |
              open(file_name||".scm") |
              open(LOADLIB||file_name) |
              open (LOADLIB||file_name||".scm"))) then
    write("reading "||file_name||" ... ")
  else
  {
    write("can't open "||file_name)
    fail
  }
  # redirect input from stdin to fin
  old_source := input_source    # save previous source
  input_source := fin           # identify new source
  while (item := _read()) do
  {
    _eval(item)
  }
  close(fin)
  write("      ... done!")
  # redirect input from fin to stdin
  input_source := old_source    # restore previous source
  return
end

procedure newline ()
# send a carriage return to output

  write()
end

procedure print (args)
# use procedure _print from skeme_reader
# to print single elements (atoms, lists, ... ) to output

  (_length(args) = 1) | fail
```

```
  item := _first(args)
  return _print(item)
end

procedure promise_forcedp (args)
# predicate to determine whether a promise has been forced

  (_length(args) = 1) | fail
  item := _first(args)
  _promisep(item) | fail
  if (\item.result) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure promise_value (args)
# returns resulting value of a previously forced promise

  (_length(args) = 1) | fail
  item := _first(args)
  _promisep(item) | fail
  if (\item.result) then
    return item.result
end

procedure skeme_read (args)
# read from standard input

  (_nullp(args)) | fail
  return _read()
end

procedure quit ()
# no explanation needed!

  stop ("   ... that's all folks!!!")
end

# ------------------------------------------------------------ #
```

29.5 special forms: unique procedures

The **special_forms.icn** file contains mid-level implementations of **scheme** special forms. As noted in the text, both primitives and special forms are building block elements in the **scheme** programming language. Primitives evaluate all their actual arguments when applied; special forms selectively evaluate their actual arguments as necessary.

special_forms.icn

```
# ------------------------------------------------------------------------
#
#
#                          initial special forms
#
#
#
# ------------------------------------------------------------------------

procedure begin (args)
# scheme special form begin
# creates a block of Scheme code to be executed as a single unit
# result returned from a begin block is the result of the last statemen

   (_length(args) = 0) & return []
   result := _eval(_first(args))
   if (_nullp(_rest(args))) then
      return result
   else
      return begin(_rest(args))
end

procedure define (args)
# scheme special form define
# inserts a new symbol into the environment and assigns an initial valu
# see procedure set! (setx) below

   if (_listp(_first(args))) then
   {
      signature := _first(args)
      id := _first(signature)
      _symbolp(id) | (write ("invalid_define") & return)
      (defined := search(id)) & return
      formals := _rest(signature)
      body := _rest(args)
      scope := *environment
      (define_value := lambda_form(formals,body,scope)) | fail
      insert_environment(id,define_value)
      return define_value
   }
```

```
  else
  {
    (_length(args) = 2) | fail
    id := _first(args)
    value := _first(_rest(args))
    _symbolp(id) | (write ("invalid_define") & return)
    (defined := search(id)) & return
    (define_value := _eval(value)) | fail
    insert_environment(id,define_value)
    return define_value
  }
end

procedure delay (args)
# scheme special form delay

  (_length(args) = 1) | fail
  expression := _first(args)
  return promise(expression,&null)
end

procedure if_then_else (args)
# scheme special form if
# the first  item in args identifies a specific test to be performed
# the second item in args identifies what to evaluate if test is true
# the third  item in args identifies what to evaluate if test is false
#     optional argument! nil if not specified!

  (_length(args) > 3) & fail
  (_length(args) < 2) & fail
  test := _first(args)
  t_expr := _first(_rest(args))
  if (_length(args) = 3) then
    f_expr := _first(_rest(_rest(args)))
  else
    f_expr := []
  if (_eqvp(_eval(test),boolean("#f"))) then
    return _eval(f_expr)
  else
    return _eval(t_expr)
end

procedure lambda (args)
# creates a lambda form -- an anonymous user-defined procedure
# lambda forms will have evaluated actual arguments (call by value)

  (_length(args) >= 1) | fail
  formals := _first(args)
  body := _rest(args)
  scope := *environment
  return lambda_form(formals,body,scope)
end
```

```
procedure macro (args)
# creates a macro form -- an anonymous user-defined procedure
# macro forms will have unevaluated actual arguments (call by name)

   (_length(args) >= 1) | fail
   formals := _first(args)
   body := _rest(args)
   scope := *environment
   return macro_form(formals,body,scope)
end

procedure macro_expand (args)
# applies a macro form to its supplied arguments to generate its result
# but macro-expand does not evaluate the result of macro expansion

   (_length(args) = 1) | fail
   name := _first(_first(args))
   _macro_formp(macro := _eval(name)) | fail
   actual_args := _rest(_first(args))
   return _procedure_evaluation(macro,actual_args)
end

procedure setx (args)
# scheme special form set!
# updates a symbol found in the environment and with a new value
# see procedure define abov

   (_length(args) = 2) | fail
   id := _first(args)
   value := _first(_rest(args))
   _symbolp(id) | (write ("invalid_set!") & return)
   (defined := search(id)) | return
   (update_value := _eval(value)) | fail
   update_environment(id,update_value)
   return update_value
end

# ------------------------------------------------------------------------
#
#
#              quasiquote, unquote, and unquote-splicing
#
#
#
# ------------------------------------------------------------------------

procedure quasiquote (args)
# scheme special form quasiquote; similar to special form quote
# but scans item for elements to unquote, i.e., evaluate!

   return _eval(quasiquote_conversion(args))
end
```

```
procedure quasiquote_conversion (args)
# this is very basic form of quasiquote:
#     - no nesting of quasiquoting
#     - both unquote and unquote-splicing are permitted

   (_length(args) = 0) & return []
   item := _first(args)
   if (_atomp(item)) then
      return [symbol("quote"),item]
   else if (unquotep(item)) then
      return _first(_rest(item))
   else if (unquote_splicingp(item)) then
   {
      write("undefined unquote-splicing context")
      fail
   }
   else
      return [symbol("append")] ||| unquote_scan (item)
end

procedure unquotep (item)
# predicate procedure: is the item "unquote"

   _listp(item) | fail
   x := _first(item)
   (_symbolp(x) & (x.identifier == "unquote")) | fail
   return
end

procedure unquote_splicingp (item)
# predicate procedure: is the item "unquote-splicing"

   _listp(item) | fail
   x := _first(item)
   (_symbolp(x) & (x.identifier == "unquote-splicing")) | fail
   return
end

procedure unquote_scan (args)
# scan a list of arguments for elements "unquote" or "unquote-splicing"

   _nullp(args) & return []
   return [unquote_item(_first(args))] ||| unquote_scan(_rest(args))
end

procedure unquote_item (item)
# scan a single item within unquote_scan

   if (_atomp(item)) then
      return [symbol("list"),[symbol("quote"),item]]
   else if (unquotep(item)) then
      return [symbol("list"),_first(_rest(item))]
   else if (unquote_splicingp(item)) then
```

```
    return _first(_rest(item))
  else
    return [symbol("list"),[symbol("append")] ||| unquote_scan(item)]
end

# ------------------------------------------------------------------------
```

29.6 environment: symbol definition

The **environment.icn** file defines the initial symbol table to be used by the **scheme** interpreter to identify defined symbols and their corresponding value. The environment also has responsibility for implementing the proper nested scope for each symbol.

environment.icn

```
#----------------------------------------------------------#

global base_environment
global current_environment
record environment (symbol_table,parent)

#----------------------------------------------------------#

procedure initialize_environment ()

  symbol_table := table()

  # predefined atoms

  symbol_table["nil"]           := []
  symbol_table["true"]          := boolean("#t")
  symbol_table["false"]         := boolean("#f")
  symbol_table["else"]          := boolean("#t")
  symbol_table["e"]             := 2.718281828
  symbol_table["pi"]            := 3.141592654

  # primitive predicates

  symbol_table["integer?"]      := primitive(integerp)
  symbol_table["real?"]         := primitive(realp)
  symbol_table["number?"]       := primitive(numberp)
  symbol_table["char?"]         := primitive(charp)
  symbol_table["string?"]       := primitive(stringp)
  symbol_table["boolean?"]      := primitive(booleanp)
  symbol_table["symbol?"]       := primitive(symbolp)
  symbol_table["atom?"]         := primitive(atomp)

  symbol_table["null?"]         := primitive(nullp)
  symbol_table["list?"]         := primitive(listp)
  symbol_table["pair?"]         := primitive(pairp)
  symbol_table["alist?"]        := primitive(alistp)

  symbol_table["primitive?"]    := primitive(primitivep)
  symbol_table["special-form?"] := primitive(special_formp)
  symbol_table["lambda?"]       := primitive(lambdap)
  symbol_table["macro?"]        := primitive(macrop)
  symbol_table["procedure?"]    := primitive(procedurep)
  symbol_table["promise?"]      := primitive(promisep)
```

```
symbol_table["eq?"]             := primitive(eqp)
symbol_table["eqv?"]            := primitive(eqvp)
symbol_table["equal?"]          := primitive(equalp)

# initial primitives

symbol_table["cons"]            := primitive(cons)
symbol_table["first"]           := primitive(first)
symbol_table["rest"]            := primitive(rest)

symbol_table["apply"]           := primitive(apply)
symbol_table["eval"]            := primitive(eval)
symbol_table["force"]           := primitive(force)
symbol_table["gensym"]          := primitive(gensym)
symbol_table["load"]            := primitive(load)
symbol_table["newline"]         := primitive(newline)
symbol_table["print"]           := primitive(print)
symbol_table["promise-forced?"]
                                := primitive(promise_forcedp)
symbol_table["promise-value"]   := primitive(promise_value)
symbol_table["read"]            := primitive(skeme_read)
symbol_table["quit"]            := primitive(quit)

# initial special forms

symbol_table["begin"]           := special_form(begin)
symbol_table["define"]          := special_form(define)
symbol_table["delay"]           := special_form(delay)
symbol_table["if"]              := special_form(if_then_else)
symbol_table["lambda"]          := special_form(lambda)
symbol_table["macro"]           := special_form(macro)
symbol_table["macro-expand"]    := special_form(macro_expand)
symbol_table["quasiquote"]      := special_form(quasiquote)
symbol_table["quote"]           := special_form(quote)
symbol_table["set!"]            := special_form(setx)

# initial number primitives

symbol_table["add"]             := primitive(add)
symbol_table["sub"]             := primitive(sub)
symbol_table["mul"]             := primitive(mul)
symbol_table["div"]             := primitive(div)
symbol_table["n_eq"]            := primitive(n_eq)
symbol_table["n_lt"]            := primitive(n_lt)
symbol_table["n_gt"]            := primitive(n_gt)
symbol_table["n_le"]            := primitive(n_le)
symbol_table["n_ge"]            := primitive(n_ge)
symbol_table["ceiling"]         := primitive(ceiling)
symbol_table["floor"]           := primitive(floor)
symbol_table["sqrt"]            := primitive(n_sqrt)
symbol_table["sin"]             := primitive(n_sin)
symbol_table["cos"]             := primitive(n_cos)
symbol_table["tan"]             := primitive(n_tan)
```

```
symbol_table["exp"]             := primitive(n_exp)
symbol_table["log"]             := primitive(n_log)
symbol_table["random"]          := primitive(n_random)

# initial character primitives

symbol_table["char=?"]          :=  primitive(ch_eq)
symbol_table["char<?"]          :=  primitive(ch_lt)
symbol_table["char>?"]          :=  primitive(ch_gt)
symbol_table["char<=?"]         :=  primitive(ch_le)
symbol_table["char>=?"]         :=  primitive(ch_ge)

symbol_table["char-alphabetic?"]
                                :=  primitive(ch_alphap)
symbol_table["char-numeric?"]
                                :=  primitive(ch_digitp)
symbol_table["char-whitespace?"]
                                :=  primitive(ch_whitep)
symbol_table["char-upper-case?"]
                                :=  primitive(ch_ucp)
symbol_table["char-lower-case?"]
                                :=  primitive(ch_lcp)

symbol_table["char->integer"] :=  primitive(ch2int)
symbol_table["integer->char"] :=  primitive(int2ch)

symbol_table["char-upcase"]   :=  primitive(ch2uc)
symbol_table["char-downcase"] :=  primitive(ch2lc)

# initial string primitives

symbol_table["make-string"]   := primitive(st_make_string)
symbol_table["string"]        := primitive(st_string)
symbol_table["string-length"] := primitive(st_length)
symbol_table["string-ref"]    := primitive(st_ref)
symbol_table["string=?"]      := primitive(st_eq)
symbol_table["string<?"]      := primitive(st_lt)
symbol_table["string>?"]      := primitive(st_gt)
symbol_table["string<=?"]     := primitive(st_le)
symbol_table["string>=?"]     := primitive(st_ge)
symbol_table["string-copy"]   := primitive(st_copy)
symbol_table["substring"]     := primitive(st_substring)
symbol_table["substring?"]    := primitive(st_substringp)
symbol_table["string-append"] := primitive(st_append)
symbol_table["string-reverse"]:= primitive(st_reverse)
symbol_table["palindrome?"]   := primitive(st_palindromep)
symbol_table["string->list"]  := primitive(st_st2list)
symbol_table["list->string"]  := primitive(st_list2st)
symbol_table["string->symbol"]:= primitive(st_st2sym)
symbol_table["symbol->string"]:= primitive(st_sym2st)

# initial list primitives
```

```
   symbol_table["list"]          := primitive(lst_list)
   symbol_table["length"]        := primitive(lst_length)
   symbol_table["sublist"]       := primitive(lst_sublist)
   symbol_table["sublist?"]      := primitive(lst_sublistp)
   symbol_table["append"]        := primitive(lst_append)
   symbol_table["reverse"]       := primitive(lst_reverse)
   symbol_table["list-ref"]      := primitive(lst_ref)
   symbol_table["list-copy"]     := primitive(lst_copy)
   symbol_table["memq"]          := primitive(lst_memq)
   symbol_table["memv"]          := primitive(lst_memv)
   symbol_table["member"]        := primitive(lst_member)
   symbol_table["assq"]          := primitive(lst_assq)
   symbol_table["assv"]          := primitive(lst_assv)
   symbol_table["assoc"]         := primitive(lst_assoc)

   symbol_table["environment"]   := primitive(dump)

   base_environment := environment (symbol_table,&null)
   current_environment := base_environment
end

#------------------------------------------------------------#

procedure search (symbol)
# searches for a symbol in the database
# starting with the current environment
#     and searching backwards through the static chain
# either succeeds or fails

   name := symbol.identifier
   env := current_environment
   while (\env) do
      if (value := \env.symbol_table[name]) then
         return value
      else
         env := env.parent
end

#------------------------------------------------------------#

procedure insert_environment (symbol,value)
# defines a previously undefined symbol in the environment

   name := symbol.identifier
   /current_environment.symbol_table[name] | fail
   current_environment.symbol_table[name] := value
   return value
end

procedure update_environment (symbol,value)
# updates a previously defined symbol in the environment

   name := symbol.identifier
```

```
   env := current_environment
   while (\env) do
      if (\env.symbol_table[name]) then
      {
         env.symbol_table[name] := value
         return value
      }
      else
         env := env.parent
end

#--------------------------------------------------------------#

procedure dump ()
# display the entire set of symbol tables,
#      its current level and its parent level
#      for each level in the environment
#      starting with the current level
#      and working backword to initial environment

   env := current_environment
   every (\env) do
   {
      every (item := key(env.symbol_table)) do
         if (type(env.symbol_table[item]) ==
               ("primitive" | "special_form")) then
         {
            writes(left(item,25))
            write(type(env.symbol_table[item]))
         }
         else
         {
            writes(left(item,25))
            print_item(env.symbol_table[item])
            write()
         }
      write ()
   }
end

#--------------------------------------------------------------#
```

29.7 additional primitives

The **scheme** programming language supports a variety of built-in data types: numbers, characters, strings, boolean, and lists. Each of these data types has its own set of primitives associated with. Some of the primitives have already been implemented in the files above; others have not. The following pages contain the implementation of primitives associated with the given data type.

29.7.1 numbers

numbers.icn

```
# ------------------------------------------------------------ #
#
#
#          essential numberic primitives
#
#
#
# ------------------------------------------------------------ #

# the basic arithmetic building blocks:

#     - add
#     - sub
#     - mul
#     - div

# note: these are not binary operations
#    rather they work on the entire argument list!

#     (+) => 0
#     (-) => 0
#     (- x) => -x
#     (- list) => (- (car list) (+ (cdr list)))

#     (*) => 1
#     (/) => 1
#     (/ x) => 1/x as a decimal (real) value
#     (/ list) => (/ (car list) (* (cdr list)))

procedure add (args)
  _nullp(args) & return 0
  _numberp(_first(args)) | fail
  return _first(args) + add(_rest(args))
end

procedure sub (args)
```

```
  _nullp(args) & return 0
  _numberp(_first(args)) | fail
  if (_length(args) = 1) then
    return 0 - _first(args)
  else
    return _first(args) - add(_rest(args))
end

procedure mul (args)
  _nullp(args) & return 1
  _numberp(_first(args)) | fail
  return _first(args) * mul(_rest(args))
end

procedure div (args)
  _nullp(args) & return 1
  _numberp(_first(args)) | fail
  if (_length(args) = 1) then
    return 1.0 / _first(args)
  else
    return _first(args) / mul(_rest(args))
end

# ------------------------------------------------------------ #

# the basic relational building blocks
# specific to numerical values only:

#     - =
#     - >
#     - >=
#     - <
#     - <=

# note: once again these are not binary operations
#    rather they work on the entire argument list
#    proceding from left to right!

# for example,
#   (= list) =>
#      if not (= (car list) (cadr list)) => #f
#      if     (= (car list) (cadr list)) =>
#         if     (= (length list) 2) => #t
#         if not (= (length list) 2) => (= (cdr list)

procedure n_eq (args)
  (_length(args) < 2) & fail
  (_numberp(_first(args))) | fail
  (_numberp(_first(_rest(args)))) | fail
  if (_first(args) = _first(_rest(args))) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
```

```
      return n_eq(_rest(args))
  else
    return boolean("#f")
end

procedure n_lt (args)
  (_length(args) < 2) & fail
  (_numberp(_first(args))) | fail
  (_numberp(_first(_rest(args)))) | fail
  if (_first(args) < _first(_rest(args))) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return n_lt(_rest(args))
  else
    return boolean("#f")
end

procedure n_gt (args)
  (_length(args) < 2) & fail
  (_numberp(_first(args))) | fail
  (_numberp(_first(_rest(args)))) | fail
  if (_first(args) > _first(_rest(args))) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return n_gt(_rest(args))
  else
    return boolean("#f")
end

procedure n_le (args)
  (_length(args) < 2) & fail
  (_numberp(_first(args))) | fail
  (_numberp(_first(_rest(args)))) | fail
  if (_first(args) <= _first(_rest(args))) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return n_le(_rest(args))
  else
    return boolean("#f")
end

procedure n_ge (args)
  (_length(args) < 2) & fail
  (_numberp(_first(args))) | fail
  (_numberp(_first(_rest(args)))) | fail
  if (_first(args) >= _first(_rest(args))) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return n_ge(_rest(args))
```

```
  else
    return boolean("#f")
end

#--------------------------------------------------------------#

# two common conversion functions:
#   real --> integer

procedure ceiling (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_numberp(x)) | fail
  if (integer(x) = x) then return integer(x)
  if (x >= 0) then
    return integer(x) + 1
  else
    return -floor([-x])
end

procedure floor (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_numberp(x)) | fail
  if (integer(x) = x) then return integer(x)
  if (x >= 0) then
    return integer(x)
  else
    return -ceiling([-x])
end

#--------------------------------------------------------------#

# several common real functions:
#   real --> real

procedure n_sqrt (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_numberp(x)) | fail
  return sqrt(x)
end

procedure n_sin (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_numberp(x)) | fail
  return sin(x)
end

procedure n_cos (args)
  (_length(args) = 1) | fail
  x := _first(args)
```

```
  (_numberp(x)) | fail
  return cos(x)
end

procedure n_tan (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_numberp(x)) | fail
  return tan(x)
end

procedure n_exp (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_numberp(x)) | fail
  return exp(x)
end

procedure n_log (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_numberp(x)) | fail
  return log(x)
end

procedure n_random (args)
  (_length(args) = 1) | fail
  x := _first(args)
  (_integerp(x)) | fail
  return ?x - 1
end
```

————————————————————————————————

29.7.2 characters

characters.icn

```
#---------------------------------------------------------- #
#
#
#          characters: primitive elements
#
#
#
#---------------------------------------------------------- #

procedure ch_eq (args)
  (_length(args) < 2) & fail
  value1 := _first(args)
  value2 := _first(_rest(args))
  (_charp(value1)) | fail
  (_charp(value2)) | fail
  if (string(value1) == string(value2)) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return ch_eq(_rest(args))
  else
    return boolean("#f")
end

procedure ch_lt (args)
  (_length(args) < 2) & fail
  value1 := _first(args)
  value2 := _first(_rest(args))
  (_charp(value1)) | fail
  (_charp(value2)) | fail
  if (string(value1) << string(value2)) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return ch_lt(_rest(args))
  else
    return boolean("#f")
end

procedure ch_gt (args)
  (_length(args) < 2) & fail
  value1 := _first(args)
  value2 := _first(_rest(args))
  (_charp(value1)) | fail
  (_charp(value2)) | fail
  if (string(value1) >> string(value2)) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
```

```
      return ch_gt(_rest(args))
  else
    return boolean("#f")
end

procedure ch_le (args)
  (_length(args) < 2) & fail
  value1 := _first(args)
  value2 := _first(_rest(args))
  (_charp(value1)) | fail
  (_charp(value2)) | fail
  if (string(value1) <<= string(value2)) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return ch_le(_rest(args))
  else
    return boolean("#f")
end

procedure ch_ge (args)
  (_length(args) < 2) & fail
  value1 := _first(args)
  value2 := _first(_rest(args))
  (_charp(value1)) | fail
  (_charp(value2)) | fail
  if (string(value1) >>= string(value2)) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return ch_ge(_rest(args))
  else
    return boolean("#f")
end

procedure ch_alphap (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
  if (value ? any(&ucase++&lcase)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure ch_digitp (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
  if (value ? any(&digits)) then
    return boolean("#t")
```

```
  else
    return boolean("#f")
end

procedure ch_whitep (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
  if (value ? any(white_space)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure ch_ucp (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
  if (value ? any(&ucase)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure ch_lcp (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
  if (value ? any(&lcase)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure ch2uc (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
  if (value ? any(&lcase)) then
    return cset(char(ord(value) - ord("a") + ord("A")))
  else
    return cset(value)
end

procedure ch2lc (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
```

```
  if (value ? any(&ucase)) then
    return cset(char(ord(value) - ord("A") + ord("a")))
  else
    return cset(value)
end

procedure ch2int (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_charp(value)) | fail
  value := string(value)
  if ("0" <<= value <<= "9") then
    return ord(value) - ord("0")
end

procedure int2ch (args)
  (_length(args) = 1) | fail
  value := _first(args)
  (_integerp(value)) | fail
  if (0 <= value <= 9) then
    return cset(char(ord("0") + value))
end
```

———————————————————————————————

29.7.3 strings

strings.icn

```
#----------------------------------------------------------#
#
#
#            strings: primitive elements
#
#
#
#----------------------------------------------------------#

procedure st_make_string (args)
  (_length(args) >= 1) | fail
  k := _first(args)
  (_integerp(k) & (k >= 0)) | fail
  if (_length(args) = 2) then
  {
    string_char := _first(_rest(args))
    (_charp(string_char)) | fail
  }
  else
    string_char := "*"
  result := ""
  every i := 1 to k do
    result ||:= string(string_char)
  return result
end

procedure st_string (args)
  (_length(args) = 0) & return ""
  string_char := _first(args)
  (_charp(string_char)) | fail
  return string(string_char) || st_string(_rest(args))
end

procedure st_length(args)
  (_length(args) = 1) | fail
  source := _first(args)
  (_stringp(source)) | fail
  return *source
end

procedure st_ref (args)
  (_length(args) = 2) | fail
  source := _first(args)
  loc := _first(_rest(args))
  (_stringp(source)) | fail
  (_integerp(loc) & (0 <= loc < *source)) | fail
  return source[loc+1]
end
```

```
procedure st_eq (args)
  (_length(args) < 2) & fail
  source1 := _first(args)
  source2 := _first(_rest(args))
  (_stringp(source1)) | fail
  (_stringp(source2)) | fail
  if (source1 == source2) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return st_eq(_rest(args))
  else
    return boolean("#f")
end

procedure st_lt (args)
  (_length(args) < 2) & fail
  source1 := _first(args)
  source2 := _first(_rest(args))
  (_stringp(source1)) | fail
  (_stringp(source2)) | fail
  if (source1 << source2) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return st_lt(_rest(args))
  else
    return boolean("#f")
end

procedure st_gt (args)
  (_length(args) < 2) & fail
  source1 := _first(args)
  source2 := _first(_rest(args))
  (_stringp(source1)) | fail
  (_stringp(source2)) | fail
  if (source1 >> source2) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return st_gt(_rest(args))
  else
    return boolean("#f")
end

procedure st_le (args)
  (_length(args) < 2) & fail
  source1 := _first(args)
  source2 := _first(_rest(args))
  (_stringp(source1)) | fail
  (_stringp(source2)) | fail
  if (source1 <<= source2) then
    if (_length(args) = 2) then
```

```
      return boolean("#t")
    else
      return st_le(_rest(args))
  else
    return boolean("#f")
end

procedure st_ge (args)
  (_length(args) < 2) & fail
  source1 := _first(args)
  source2 := _first(_rest(args))
  (_stringp(source1)) | fail
  (_stringp(source2)) | fail
  if (source1 >>= source2) then
    if (_length(args) = 2) then
      return boolean("#t")
    else
      return st_ge(_rest(args))
  else
    return boolean("#f")
end

procedure st_copy (args)
  (_length(args) = 1) | fail
  source := _first(args)
  (_stringp(source)) | fail
  return copy(source)
end

procedure st_substring (args)
  (_length(args) = 3) | fail
  source := _first(args)
  start := _first(_rest(args))
  stop := _first(_rest(_rest(args)))
  (_stringp(source)) | fail
  (_integerp(start) & _integerp(stop) &
    (0 <= start <= stop <= *source)) | fail
  return source[start+1:stop+1]
end

procedure st_substringp (args)
  (_length(args) = 2) | fail
  stringa := _first(args)
  _stringp(stringa) | fail
  stringb := _first(_rest(args))
  _stringp(stringb) | fail
  if (*stringa > *stringb) then
    return boolean("#f")
  else if (stringa == stringb[1:*stringa+1]) then
    return boolean("#t")
  else
    return st_substringp([stringa,stringb[2:0]])
end
```

```
procedure st_append (args)
  (_length(args) = 0) & return ""
  source := _first(args)
  (_stringp(source)) | fail
  return source || st_append(_rest(args))
end

procedure st_reverse (args)
  (_length(args) = 1) | fail
  source := _first(args)
  (_stringp(source)) | fail
  (*source = 0) & return ""
  return source[-1] || st_reverse([source[1:-1]])
end

procedure st_palindromep (args)
  (_length(args) = 1) | fail
  source := _first(args)
  (_stringp(source)) | fail
  if (source === st_reverse(args)) then
    return boolean("#t")
  else
    return boolean("#f")
end

procedure st_st2list (args)
  (_length(args) = 1) | fail
  source := _first(args)
  (_stringp(source)) | fail
  result := []
  every i := 1 to *source do
    result |||:= [cset(source[i])]
  return result
end

procedure st_list2st (args)
  (_length(args) = 1) | fail
  source := _first(args)
  (_listp(source)) | fail
  result := ""
  every i := 1 to *source do
  {
    (_charp(source[i])) | fail
    result ||:= string(source[i])
  }
  return result
end

procedure st_st2sym (args)
  (_length(args) = 1) | fail
  item := _first(args)
  (_stringp(item)) | fail
```

```
  return symbol(item)
end

procedure st_sym2st (args)
  (_length(args) = 1) | fail
  item := _first(args)
  (_symbolp(item)) | fail
  return item.identifier
end
```

——

29.7.4 lists

lists.icn

```
# ------------------------------------------------------------ #
#
#
#           essential list primitives
#
#
#
# ------------------------------------------------------------ #

procedure lst_list (args)
  (_listp(args)) | fail
  return args
end

procedure lst_length (args)
  (_length(args) = 1) | fail
  listx := _first(args)
  return _length(listx)
end

procedure lst_copy (args)
  (_length(args) = 1) | fail
  source := _first(args)
  (_listp(source)) | fail
  return copy(source)
end

procedure lst_sublist (args)
  (_length(args) = 3) | fail
  source := _first(args)
  start := _first(_rest(args))
  stop := _first(_rest(_rest(args)))
  (_listp(source)) | fail
  (_integerp(start) & _integerp(stop) &
    (0 <= start <= stop <= *source)) | fail
  return source[start+1:stop+1]
end

procedure lst_sublistp (args)
  (_length(args) = 2) | fail
  lista := _first(args)
  _listp(lista) | fail
  listb := _first(_rest(args))
  _listp(listb) | fail
  if (*lista > *listb) then
    return boolean("#f")
  else if _equalp(lista,listb[1:*lista+1]) then
    return boolean("#t")
  else
```

```
    return lst_sublistp([lista,listb[2:0]])
end

procedure lst_append (args)
  (_length(args) = 0) & return []
  listx := _first(args)
  (_listp(listx)) | fail
  return listx ||| lst_append(_rest(args))
end

procedure lst_reverse (args)
  (_length(args) = 1) | fail
  listx := _first(args)
  return _reverse(listx)
end

procedure lst_ref (args)
  (_length(args) = 2) | fail
  listx := _first(args)
  (_listp(listx)) | fail
  k := _first(_rest(args))
  (_integerp(k) & (0 <= k < *listx)) | fail
  return listx[k+1]
end

procedure lst_memq (args)
  (_length(args) = 2) | fail
  obj := _first(args)
  listx := _first(_rest(args))
  if (_nullp(listx)) then
    return boolean("#f")
  if (_eqp(obj,_first(listx))) then
    return listx
  else
    return lst_memq([obj]|||[_rest(listx)])
end

procedure lst_memv (args)
  (_length(args) = 2) | fail
  obj := _first(args)
  listx := _first(_rest(args))
  if (_nullp(listx)) then
    return boolean("#f")
  if (_eqvp(obj,_first(listx))) then
    return listx
  else
    return lst_memv([obj]|||[_rest(listx)])
end

procedure lst_member (args)
  (_length(args) = 2) | fail
  obj := _first(args)
  listx := _first(_rest(args))
```

```
  if (_nullp(listx)) then
    return boolean("#f")
  if (_equalp(obj,_first(listx))) then
    return listx
  else
    return lst_member([obj]||||[_rest(listx)])
end

# an association list in skeme is a list of pairs
#     pairs are proper lists having two or more items
#     pairs are not dotted pairs in the original Lisp sense

procedure lst_assq (args)
  (_length(args) = 2) | fail
  obj := _first(args)
  assoc_list := _first(_rest(args))
  if (_nullp(assoc_list)) then
    return boolean("#f")
  if (_eqp(obj,_first(_first(assoc_list)))) then
    return _first(assoc_list)
  else
    return lst_assq([obj]||||[_rest(assoc_list)])
end

procedure lst_assv (args)
  (_length(args) = 2) | fail
  obj := _first(args)
  assoc_list := _first(_rest(args))
  if (_nullp(assoc_list)) then
    return boolean("#f")
  if (_eqvp(obj,_first(_first(assoc_list)))) then
    return _first(assoc_list)
  else
    return lst_assv([obj]||||[_rest(assoc_list)])
end

procedure lst_assoc (args)
  (_length(args) = 2) | fail
  obj := _first(args)
  assoc_list := _first(_rest(args))
  if (_nullp(assoc_list)) then
    return boolean("#f")
  if (_equalp(obj,_first(_first(assoc_list)))) then
    return _first(assoc_list)
  else
    return lst_assoc([obj]||||[_rest(assoc_list)])
end

# ---------------------------------------------------------- #
```

29.8 loadable skeme files

Once the **scheme** building blocks have been implemented and working successfully, there is almost no limit to what we can do within the language. We can define new functions and macros; we can define new data structures. However, it would be nice to not be required to repeatedly type instructions into the interpreter every time we wish to use such a new item.

That is the rationale behind the **load** primitive. We can read an ASCII text file containing **scheme** statements into our interpreter any time we desire. These loadable files typically have the extension **.scm** and at interpreter startup the interpreter typically searches for a specific loadable file called **init.scm**.

29.8.1 init

This particular **init.scm** file contains all the syntactic sugar that I discussed within the text. I thought it was a very good illustration of how functional programming languages can very easily be extended and enhanced using lambda forms and macro forms.

init.scm

```
;;; euphemisms ------------------------------------------------ #

(define exit quit)
(define bye quit)

(define car first)
(define cdr rest)

(define write print)
(define display (lambda (item)
  (if (string? item)
      (print (string->symbol item))
      (print item) ) ))


;;; useful scheme features ------------------------------------ #

(define compose (lambda (f g)
  (lambda (x)
    (f (g x)) ) ))

(define writeln (lambda args
  (for-each display args)
```

```
  (newline) ))

(define error (lambda args
  (display "ERROR: ")
  (for-each display args)
  (newline) ))


;;; nested car / cdr combos ------------------------------ #

(define caar (lambda (x)
  (car (car x)) ))
(define cadr (lambda (x)
  (car (cdr x)) ))
(define cdar (lambda (x)
  (cdr (car x)) ))
(define cddr (lambda (x)
  (cdr (cdr x)) ))

(define caaar (lambda (x)
  (car (caar x)) ))
(define caadr (lambda (x)
  (car (cadr x)) ))
(define cadar (lambda (x)
  (car (cdar x)) ))
(define caddr (lambda (x)
  (car (cddr x)) ))
(define cdaar (lambda (x)
  (cdr (caar x)) ))
(define cdadr (lambda (x)
  (cdr (cadr x)) ))
(define cddar (lambda (x)
  (cdr (cdar x)) ))
(define cdddr (lambda (x)
  (cdr (cddr x)) ))

(define caaaar (lambda (x)
  (car (caaar x)) ))
(define caaadr (lambda (x)
  (car (caadr x)) ))
(define caadar (lambda (x)
  (car (cadar x)) ))
(define caaddr (lambda (x)
  (car (caddr x)) ))
(define cadaar (lambda (x)
  (car (cdaar x)) ))
(define cadadr (lambda (x)
  (car (cdadr x)) ))
(define caddar (lambda (x)
  (car (cddar x)) ))
(define cadddr (lambda (x)
  (car (cdddr x)) ))
(define cdaaar (lambda (x)
```

```
  (cdr (caaar x)) ))
(define cdaadr (lambda (x)
  (cdr (caadr x)) ))
(define cdadar (lambda (x)
  (cdr (cadar x)) ))
(define cdaddr (lambda (x)
  (cdr (caddr x)) ))
(define cddaar (lambda (x)
  (cdr (cdaar x)) ))
(define cddadr (lambda (x)
  (cdr (cdadr x)) ))
(define cdddar (lambda (x)
  (cdr (cddar x)) ))
(define cddddr (lambda (x)
  (cdr (cdddr x)) ))


;;; logic: syntactic sugar ------------------------------- #

(define not (lambda (arg)
; arg: (boolean-expression)
; simple logical negation

  (if arg #f #t) ))

(define and (macro args
; args: (boolean-expression boolean-expression ... )
; simple logical conjunction
; but using short-circuit implementation

  (if (null? args)
      #t
      `(if ,(car args)
           (and ,@(cdr args))
           #f ) ) ))

(define or (macro args
; args: (boolean-expression boolean-expression ... )
; simple logical disjunction
; but using short-circuit implementation

  (if (null? args)
      #f
      `(if ,(car args)
           #t
           (or ,@(cdr args)) ) ) ))

;; ----------------------------------------------------------- #

(define when (macro args
; args: (boolean-test body)
; body: statement1 statement2 ...
; if boolean-test is true, then execute body
```

```
  `(if ,(car args) (begin ,@(cdr args)) ()) ))

(define unless (macro args
; args: (boolean-test body)
; body: statement1 statement2 ...
; if boolean-test is false, then execute body

  `(if ,(car args) () (begin ,@(cdr args))) ))

;; ------------------------------------------------------------ #

(define cond (macro args
; args: ((test1 body1) (test2 body2) ... )
; testi: boolean expression
; body: statement1 statement2 ...
; if testi is true, then execute bodyi
;    else move on to next (test body) pair

  (if (null? (car args))
      '()
      `(if ,(caar args)
           (begin ,@(cdar args))
           (cond ,@(cdr args)) ) ) ))

(define while (macro args
; args: (test body)
; test: boolean-expression
; body: statement1 statement2 ...
; while the boolean-test evaluates to true
;    execute the body of the loop
; this is a top-tested loop!

  `(when ,(car args)
     (begin ,@(cdr args) (while ,@args))) ))

(define until (macro args
; args: (test body)
; test: boolean-expression
; body: statement1 statement2 ...
; until the boolean-test evaluates to true
;    execute the body of the loop
; this is a bottom-tested loop!

  `(begin ,@(cdr args)
     (unless ,(car args) (until ,@args))) ))


;;; set-car! and set-cdr!: syntactic sugar ---------------- #
; basic mutators for dotted pairs

(define set-car! (macro (p x)
; replaces (car p) with item x
```

```
  `(set! ,p (cons ,x (cdr ,p))) ))

(define set-cdr! (macro (p x)
; replaces (cdr p) with item x
  `(set! ,p (cons (car ,p) ,x)) ))


;;; let: syntactic sugar ------------------------------ #

(define let (macro args
; args: list-pairs body
; list-pairs: ((symbol1 expression1) (symbol2 expression2) ... )
; body: statement1 statement2 ...

; NOTE: expressions in let are limited to symbols
;       from the CURRENT environment!
; no reference is permitted to a previously defined symbol
;       in the NEW let environment

  `((lambda ,(get-cars (car args)) ,@(cdr args))
    ,@(get-cdrs (car args)) ) ))

(define get-cars (lambda (args)
; args: ((symbol1 expression1) (symbol2 expression2) ... )

  (if (null? args)
      '()
      (cons (caar args) (get-cars (cdr args))) ) ))

(define get-cdrs (lambda (args)
; args: ((symbol1 expression1) (symbol2 expression2) ... )

  (if (null? args)
      '()
      (cons (cadar args) (get-cdrs (cdr args))) ) ))


;;; let*: syntactic sugar ----------------------------- #

(define let* (macro args
; args: list-pairs body
; list-pairs: ((symbol1 expression1) (symbol2 expression2) ... )
; body: statement1 statement2 ...

; NOTE: expressions in let* are no longer limited to
;       symbols from the CURRENT environment!
; references are permitted to a previously defined symbol
;       in the NEW let* environment

  (if (null? (car args))
      `(let () ,@(cdr args))
      `(let (,(caar args))
         (let* ,(cdar args) ,@(cdr args))) ) ))
```

```
;;; letrec: syntactic sugar ------------------------------ #

(define letrec (macro args
; args: list-pairs body
; list-pairs: ((symbol1 expression1) (symbol2 expression2) ... )
; body: statement1 statement2 ...

; NOTE: expressions in letrec are typically lambda forms!
;     i.e., the let args define functions
; if we intend for these function to
;     recursively interact with one another
; a simple let statement will not work
;     (due to scope limitations!)
; the letrec statement gets around this limitation

  (if (null? (car args))
      `(let () ,@(cdr args))
      (begin
        (define no-symbols
          (length (car args)))
        (define symbol-list
          (get-cars (car args)))
        (define value-list
          (get-cdrs (car args)))
        (define temp-list
          (create-list no-symbols))
        (define undefined-list
          (create-list no-symbols "na"))
        (define undefined-evals
          (create-let-pairs symbol-list undefined-list))
        (define temp-evals
          (create-let-pairs temp-list value-list))
        (define official-evals
          (create-set-pairs symbol-list temp-list))
        `(let (,@undefined-evals)
           (let (,@temp-evals) ,@official-evals ,
             @(cdr args))) ) ) ))
end

(define create-list (lambda args
; args (no-items value)
; creates a new list containing the specified value
;   repeated no-items
; if value is not specified,
;   then use gensym to generate unique values

  (if (= (car args) 0)
      '()
      (if (null? (cdr args))
          `(,(gensym) ,@(create-list (- (car args) 1)))
          `(,(cadr args) ,@(create-list
```

```
            (- (car args) 1) (cadr args))) ) ) ))

(define create-let-pairs (lambda (symbol-list value-list)
; args: (symbol-list value-list)
; creates a new list containing
;   pairs of corresponding values from each list
; both list must have the same length!

  (if (= (length symbol-list) (length value-list))
      (if (null? symbol-list)
          '()
          `((,(car symbol-list) ,(car value-list))
            ,@(create-let-pairs (cdr symbol-list)
              (cdr value-list)) ) )
      "length mismatch" ) ))

(define create-set-pairs (lambda (symbol-list temp-list)
; args: (symbol-list temp-list)

  (if (= (length symbol-list) (length temp-list))
      (if (null? symbol-list)
          '()
          `((set! ,(car symbol-list) ,(car temp-list))
                ,@(create-set-pairs (cdr symbol-list)
                  (cdr temp-list)) ) )
      "length mismatch" ) ))


;;; ------------------------------------------------------------ #

(load "numbers.scm")
(load "lists.scm")
(load "streams.scm")

;;; ------------------------------------------------------------ #
```

29.8.2 numbers

This file really increases the number of procedures available in **skeme** to manipulate numbers. We already had a significant number of primitives on hand. Once again, lambda forms and macro forms allow the easy creation of addition utilities.

numbers.scm

```
;;; useful numeric procedures

(define int->real (lambda (x)
  (* 1.0 x) ))

(define real->int (lambda (x)
  (if (>= x 0)
      (floor x)
      (ceiling x) ) ))

(define abs (lambda (x)
  (if (number? x)
      (if (>= x 0)
          x
          (- x) )
      'error ) ))

(define square (lambda (x)
  (if (number? x)
      (* x x)
      'error ) ))

(define power (lambda (b n)
  (cond ((= n 0) 1)
        ((> n 0) (* b (power b (- n 1))))
        ( else   (/ 1.0 (power b (- n)))) ) ))

(define total (lambda num-list
  (if (= (length num-list) 0)
      0
      (+ (car num-list)
         (apply total (cdr num-list))) ) ))

(define average (lambda num-list
  (cond ((= (length num-list) 0)
           'error )
        ((= (length num-list) 1)
           (car num-list) )
        ( else
           (/ (apply total num-list)
              (int->real (length num-list))) ) ) ))

(define maximum (lambda num-list
```

```
  (if (= (length num-list) 1)
      (let ((a (car num-list)))
           (if (number? a)
               a
               'error ) )
      (let ((a (car num-list)) (b (cadr num-list)))
           (if (and (number? a) (number? b))
               (if (>= a b)
                   (apply maximum (cons a (cddr num-list)))
                   (apply maximum (cons b (cddr num-list))) )
               'error ) ) ) ))

(define max maximum)

(define minimum (lambda num-list
  (if (= (length num-list) 1)
      (let ((a (car num-list)))
           (if (number? a)
               a
               'error ) )
      (let ((a (car num-list)) (b (cadr num-list)))
           (if (and (number? a) (number? b))
               (if (<= a b)
                   (apply minimum (cons a (cddr num-list)))
                   (apply minimum (cons b (cddr num-list))) )
               'error ) ) ) ))

(define min minimum)

(define quotient (lambda (x y)
  (if (and (integer? x) (integer? y) (not (zero? y)))
      (/ x y)
      'error ) ))

(define remainder (lambda (x y)
  (if (and (integer? x) (integer? y) (not (zero? y)))
      (- x (* y (quotient x y)))
      'error ) ))

(define modulo (lambda (x y)
  (if (and (integer? x) (integer? y) (positive? y))
      (if (>= x 0)
          (remainder x y)
          (+ (remainder x y) y) )
      'error ) ))


;;; useful predicates

(define zero? (lambda (x)
  (if (number? x)
      (= x 0)
      'error ) ))
```

```
(define positive? (lambda (x)
  (if (number? x)
      (> x 0)
      'error ) ))

(define negative? (lambda (x)
  (if (number? x)
      (< x 0)
      'error ) ))

(define even? (lambda (x)
  (if (integer? x)
      (zero? (modulo x 2))
      'error ) ))

(define odd? (lambda (x)
  (if (integer? x)
      (not (zero? (modulo x 2)))
      'error ) ))

(define prime? (lambda (x)
  (define test (lambda (test-value)
    (if (> (square test-value) x)
        true
        (if (zero? (remainder x test-value))
            false
            (test (+ test-value 1)) ) ) ))
  (if (and (integer? x) (positive? x))
      (if (= x 1)
          'one-divides-everything
          (if (= x 2)
              true
              (if (even? x)
                  false
                  (test 3) ) ) )
      'error ) ))


;;; useful integer procedures

(define inc (lambda (n)
  (if (integer? n)
      (+ n 1)
      'error ) ))

(define dec (lambda (n)
  (if (integer? n)
      (- n 1)
      'error ) ))

(define gcd (lambda (a b)
  (if (and (integer? a) (integer? b)
```

```
          (>= a 0)     (>= b 0) )
      (if (zero? (modulo a b))
          b
          (gcd b (modulo a b)) )
      'error ) ))

(define lcm (lambda (a b)
  (if (and (integer? a) (integer? b)
          (>= a 0)     (>= b 0) )
      (/ (* a b) (gcd a b))
      'error ) ))

(define factorial (lambda (n)
  (if (integer? n)
      (if (>= n 0)
          (if (or (= n 0) (= n 1))
              1
              (* n (factorial (- n 1))) )
          'negative )
      'non-integer ) ))

(define fibonacci (lambda (n)
  (if (integer? n)
      (if (>= n 0)
          (if (or (= n 0) (= n 1))
              1
              (+ (fibonacci (- n 1)) (fibonacci (- n 2))) )
          'negative )
      'non-integer ) ))
```

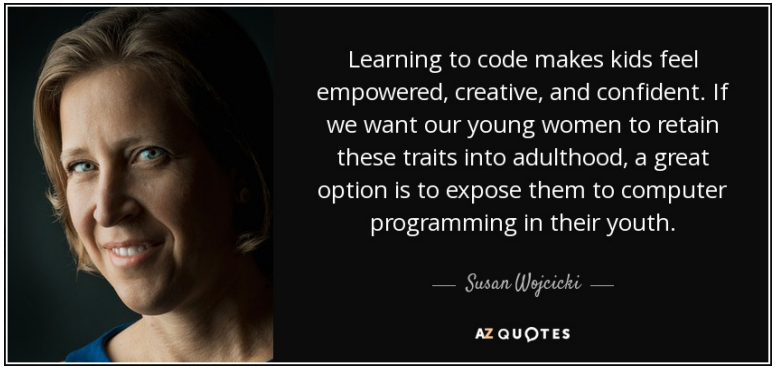
29.8.3 streams

streams.scm

```
;;; streams.scm

;;  although the title above refers only to streams
;;  in reality this file compares lists with streams
;;  lists are sequences of nested dotted pairs
;;     an item and a reference to the rest of the list
;;     terminating with the empty list
;;  streams are sequences of nested dotted pairs
;;     an item and a promise for the rest of the list
;;     terminating with a promise to the empty list

;;  streams are commonly referred to as delayed list

;;  streams have the potential to define
;;     infinitely long lists

;; empty list and empty streams
(define the-empty-list '())
(define the-empty-stream '())

(define list-null? null?)
(define stream-null? null?)

;; list basics operations
(define list-cons cons)
(define list-first first)
(define list-car first)
(define list-rest rest)
(define list-cdr rest)

;; stream basic operations
(define stream-cons (macro (x y)
  `(cons ,x (delay ,y)) ))
(define stream-first first)
(define stream-car first)
(define stream-rest (macro (streamx)
  `(force (rest ,streamx)) ))
(define stream-cdr stream-rest)

;; list and stream creation
(define stream (macro args
  (if (null? args)
      '()
      `(stream-cons ,(car args) (stream ,@(cdr args))) ) ))

;; list and stream components
(define list-pair? pair?)
(define stream-pair? (lambda (item)
  (and (pair? item)
```

```
       (promise? (cdr item)) ) ))

;; list-index and stream-index
(define list-index (lambda (listx item)
  (if (list-null? listx)
      'error
      (if (eqv? item (list-car listx))
          0
          (+ 1 (list-index
                  (list-cdr listx) item)) ) ) ))
(define stream-index (lambda (streamx item)
  (if (stream-null? streamx)
      'error
      (if (eqv? item (stream-car streamx))
          0
          (+ 1 (stream-index
                  (stream-cdr streamx) item)) ) ) ))

;; list-ref and stream-ref
(define list-ref (lambda (listx n)
  (if (< n 0)
      'error
      (if (= n 0)
          (list-car listx)
          (list-ref (list-cdr listx) (- n 1)) ) ) ))
(define stream-ref (lambda (streamx n)
  (if (< n 0)
      'error
      (if (= n 0)
          (stream-car streamx)
          (stream-ref (stream-cdr streamx) (- n 1)) ) ) ))

;; the following handful of lines of code are
;;    of great concern to me!
;; the quote-it lambda form is defined solely
;;    to allow the apply primitive
;; to execute properly on the list-to-stream conversion
;; it is a kludge to make one section of code
;;    behave properly (not a good practice!!)

;; the deeper underlying issue has to do with apply
;;    - is apply a lambda form??
;;         and evaluates all its arguments
;;    - is apply a macro form??
;;         and does not evaluate

(define quote-it (lambda (item)
  (cons 'quote (cons item '())) ))

;; list-to-stream and stream-to-list conversions
(define list->stream (macro (listx)
  `(apply stream (map quote-it ,listx)) ))
(define stream->list (lambda (streamx)
```

```
  (if (null? streamx)
      '()
      (list-cons (stream-car streamx)
                 (stream->list (stream-cdr streamx))) ) ))

;;; introduction to infinite streams

(define const-next (macro (a)
  `(stream-cons ,a (const-next ,a)) ))
(define ones (const-next 1))
(define twos (const-next 2))
(define threes (const-next 3))

(define integers-starting-from (macro (n)
  `(stream-cons ,n (integers-starting-from ,(+ n 1))) ))
(define nats (integers-starting-from 1))

(define alt-next (macro (a)
  `(stream-cons ,a (alt-next ,(- a))) ) )
(define alts (alt-next 1))

(define fib-next (macro (a b)
  `(stream-cons ,a (fib-next ,b (+ ,a ,b))) ) )

(define fibs (fib-next 1 1))

;;; at present my implementation of skeme
;;; does not like recursive definitions for streams

(define recursive-ones (stream-cons 1 recursive-ones))

; (stream-ref recursive-ones 17)
; (stream-ref recursive-ones 23)

; both will work fine,
;    but only after forcing the recursive promise
; a simple eval will cause a stack overflow
;    due to infinite recursion.
```


Learning to code makes kids feel empowered, creative, and confident. If we want our young women to retain these traits into adulthood, a great option is to expose them to computer programming in their youth.
— Susan Wojcicki —
AZ QUOTES