2

INTRODUCTION TO THE PROBLEM

2.1 MEMORANDUM FROM THE PRESIDENT

Congratulations on being named to the company's secret compiler project. Alternative Arithmetics is pleased to have developed a stack-based processor that achieves five petaflops. Such processor speeds make many heretofore impossible computations possible.

In order to make use of this speed, we must develop a new compiler. We have considered the three types of languages: imperative, functional, and logical. We believe our best chance for success lies in developing a functional language that we are tentatively calling *Errett* after the mathematician Errett Bishop. Functional languages generally allow for very powerful type systems (*ML*, for example) and very clean semantics.

In order to proceed on parallel development paths, AA's research has developed a simple version of the language design that can produce information in the appropriate form for you. We have decided to take the same tack as the GNU `gcc` compilers by separating the syntactic processing from the semantics and code generation segments. Your group will be developing the second phase: the semantics processing and code generation aspects. There are not enough chips available for you to work with a live chip, but the new petaflop chip uses a Forth-like input language. Therefore, you will be converting the syntax trees developed for Errett into Forth code. Because of its stable nature, we have decided to use the GNU Gforth system as the emulator. We have also provided a prototype syntactic processor so you can experiment with the interface.

Other memos are in preparation that outline the specific trees that you will have to process and the semantics of each of the operators or constructs.

© 2007 by Taylor & Francis Group, LLC

Best of luck. We're happy to have you here at Alternative Arithmetics and we're counting on you to develop the prototype compiler necessary to our success.

Best regards,
Socrates Polyglot
President

2.2 BACKGROUND

The Simple Object Language (*SOL*), pronounced "soul," is the primitive language for the new Simple Object Action Processor (*SOAP*). The SOAP processor is under development but we can have software emulation by using the Gforth system. Some features of Gforth may not be available in the SOAP.

2.2.1 Cambridge Polish

The prototype you are working on has two top-level constructs: type definitions and `let` statements. The output from these two constructs are linearized versions of their parse trees. Such a format has been long used in the Lisp language, dating back to the mid-1960s, and this format has long been called "Cambridge Polish."

Cambridge Polish is quite simple to understand and its beauty is that it is an efficient way to encode the structure of trees. As you all know, programs can be represented by trees. Let's recall some properties of trees:

- A *binary tree* is a graph, $T = (N, E)$, where N is a set of *nodes* and E is a subset of $N \times N$. Nodes can be anything and recall that the expression $N \times N$ is shorthand for "all possible pairs of nodes." Technically, $E \subseteq N \times N$. An *n-array* tree would be a generalization of a binary tree.
- A tree can be *directed* or *undirected.* From your data structures class you may recall that directed graphs can only be traversed in one direction. Cambridge Polish trees are directed.
- Trees are connected (no isolated nodes) and have no simple cycles. That terminology should be familiar to you from your data structures class.
- Trees can be *rooted* or *unrooted.* Cambridge Polish trees are rooted.
- Any two vertices in T can be connected by a unique simple path.
- Edges of rooted trees have a natural orientation, toward or away from the root. Rooted trees often have an additional structure, such as ordering of the neighbors at each vertex.

© 2007 by Taylor & Francis Group, LLC

- A labeled tree is a tree in which each vertex is given a unique label. The vertices of a labeled tree on *n* vertices are typically given the labels 1, 2, . . . , *n*.

The syntax for Cambridge Polish is simple. There are three types of symbols: the left bracket '[', the right bracket ']', and atoms. Atoms can be one of many different types of strings—integers or strings or real numbers or variable names—but to the syntax they are all the same. The left bracket indicates the beginning of a node and the right bracket indicates the end of a node.

Examples include:

`[]` is the node with nothing in it—NIL in some languages.
`[1]` is a node with just one atom, the integer 1.
`[+ 1 2]` is a node with three atoms: '+', 1, and 2.
`[+ [* 2 3] 4]` is two nodes (because there are two left brackets). There are five total atoms: '+', '*', 2, 3, and 4. This represents a tree that could be drawn as shown in Figure 2.1.

2.2.2 A Note on Notation

When describing the syntax below, we use two forms of notation. Atoms that appear `like-this` are written exactly as shown. Atoms that appear *like-this* can be thought of as variables. For example,

`[let` *variable value*`]`

should be read as follows: The `[` and the word `let` are written as shown; *variable* can be replaced by any allowed variable and *value* can be replaced by any valid expressions. The last statement represents the tree shown in Figure 2.2.

In Cambridge Polish the first element is considered to be the function name and the other elements are arguments.

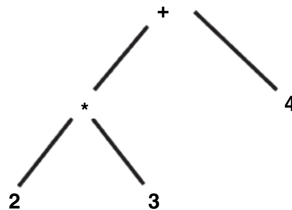

Figure 2.1 Tree for `[+ [* 2 3 ] 4 ]`

© 2007 by Taylor & Francis Group, LLC

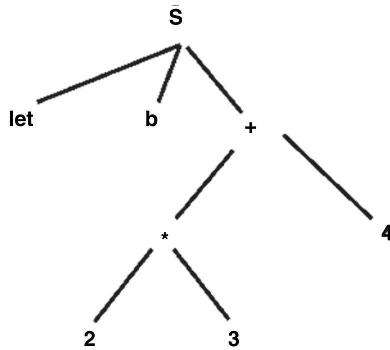

Figure 2.2 Drawing of the Tree `[let b [+ [* 2 3] 4]]`

2.2.3 A Note on Names

There are several uses for what we normally call *variables* or *names* in programming. Many key words and reserved words in programming are "spelled" just like variables. These are *surface spellings*, meaning they are written and input as shown. Internally, though, there are many names that we want to use but must be sure that these names are not something that can be confused with names in the program. We call those *pseudo-functions* and these names are preceded by the number sign (Unicode 0023) ('#'). There are design issues to be decided upon in constructs using these pseudo-functions. For example, an array constant is typed by

```
[1,2]
```

but it is presented to your input as

```
[ #arraydef 1 2 ]
```

2.3 *SOL*

SOL is a cross between an assembler language and a higher-level language. Languges such as SOL were in vogue in the late 1960s and early 1970s (PL360, for example). It adopts a simple syntactic structure, but it also adopts simple statements such as expressions and assignments and the use of higher-level control statements such as `if` and `while`.

The current state of the SOL syntactic processing is described here. The output of the syntactic program is in Cambridge Polish, and the semantic intent is described in this memo. All statements begin with an atom that designates the function to be applied. So, for example,

```
[+ 1 [* 2 3]]
```

© 2007 by Taylor & Francis Group, LLC

is the Cambridge Polish rendition of the tree for $1 + 2 * 3$. Some functions such as `let` are pseudo-functions because they take their meaning at compile time, not run time.

Generally, labels, function names, and type names can be any valid C variable name but without the underscore ('_'). Primitive operations use conventional symbols. Operands can be variable names, C integers, C floating point numbers, or C strings. Operands can also be array and pointer expressions. The variable names `true` and `false` are reserved for Boolean values and hence cannot be used as function or variable names.

The language is primarily functional in orientation: every primitive operator returns a value except the function definition; and `while`. `if` statements are a bit different than the standard C one: `if` can return a value and therefore when used functionally, both branches of the `if` must return the same value. See, too, the comment above on the use of names that begin with the number sign.

Comments do not appear in the Cambridge Polish version.

2.4 PRIMITIVE DATA AND PRIMITIVE OPERATIONS

The elements of the SOL are much like C: booleans, integers, floating point numbers, strings, and files. The language also allows for variables and function symbols.

2.4.1 Constants

`bool:` Boolean: true and false
`int:` Integers: C-type integers
`float:` Floating Point: C-type double
`string:` Strings: C-type strings
`file:` Files

The words in bold code font are the names of primitive types.

2.4.2 Variables

Variables and functions are defined by `let` statements. There are two types of `let`s. One is a "top level" that is roughly equivalent to defining globals in C. The format for the top level `let` is

`[let` [*definiendum definiendum*]`...]`

The words in italics are called *meta-variables* and those in bold are keywords; keywords must be written exactly as shown. *Definiendum* is

© 2007 by Taylor & Francis Group, LLC

Latin for "the thing being defined" and *definiens* is Latin for "the defining thing." The *definiendum* can be one of two forms: a variable or a function.

In order to define a simple scalar variable, we use the following format:

variablename : type

where *variablename* is a new name otherwise undefined and *type* is one of the allowed types: `bool` for Boolean, `int` for integers, `float` for doubles IEEE 754 constants, and `string` for strings.

2.4.2.1 Arrays

Array notation similar to C is used in the language. This translates into terms: the bracketed notation `[3,5]` is translated into `[#arrayapp 3 5]`. On the other hand, if we define an array, we see something quite different:

`[#arraytype` *type shape* `]`

where *type* is the type of the array and *shape* is the number of rows, columns, etc.

2.4.2.2 Assignment

We will have an assignment statement

`[:=` *lvalue rvalue* `]`

The lvalue can be an expression that evaluates to a pointer ("lvalue") and the rvalue is any expression ("rvalue") that evaluates to data. The lvalue and rvalue must be type compatible. In the early stages (before Milestone VIII), we can have a simple approach to assignments and the evaluation of variables. In the complex data types, we will revisit the simple evaluation.

2.5 PRIMITIVE DATA OPERATIONS

Each of the primitive data types comes equipped with pre-defined operations. The tables shown in Figures 2.5 to 2.12 later in the chapter list the primitive operations.

2.5.1 Stack Control Operations

Because the chip uses the Forth paradigm, certain of the stack control operations are available to the programming—but not all.

© 2007 by Taylor & Francis Group, LLC

2.5.1.1 Stack Position Variable

Each Forth stack can be addressed by position. The general format of a stack position expression is a dollar sign ($) followed optionally by one letter followed by a number. The letter can be A (argument) or F (floating); if no stack is specified, then A is assumed. The numbers can be any non-negative integer, with 0 being the top of stack (TOS). If the number is missing, then the TOS (position 0) is assumed. Examples are

- $ is the top of the argument stack. $F is the top of the floating point stack.
- $5 is the *sixth* item in the argument stack.
- $F6 is the *seventh* item in the float stack.

2.5.1.2 Stack Operations

The SOAP processor is a stack-based machine with a number of standard stack operations that are described in the ANSI Forth manual: `drop`, `dup`, `fetch`, `over`, `pick`, `roll`, `store`. Note that stack operations may be implied by the stack position operator.

2.6 CONTROL STRUCTURES

There are three basic control structures: sequential, selection, and repetition.

2.6.1 `If`-Selection

The `if` control structures have the general forms of

`[if` *condition statement1 statement2* `]`

`[if` *condition statement1* `]`

These are exactly like the C standard except that the *condition* must have a Boolean type.

2.6.2 `Select`-Selection

The `select` statement is a generalized *if* statement. The sequential mode does not work here: not all instructions are to be executed. Therefore, we need markers to indicate the end of the "true" side and the end of the "else" side. Nested *if* statements can be quite messy, so we add a purely

© 2007 by Taylor & Francis Group, LLC

human consideration: how can we tell which *if* section ends where? The `select` statement is similar to the `switch` in C, with the exception that the code executed when the condition is true is treated as if it ended with `break`. *select* also differs from C in that the case conditions are arbitrary expressions, not just constants.

[select *expression caselist*]

where *expression* computes a value with the usual interpretation. The *caselist* is a list of pairs:

[#selectcase *test result*].

Each *test* is computed in the order stated. If the *test* returns `true`, then the *result* is computed. If the *test* is `false`, then the next pair is evaluated. This is similar to C's *switch* statement, but the *test* can be a general statement.

One of the pairs can be the `default` condition, which should be the last pair.

2.6.2.1 While

While statements are syntactically very simple:

[while *condition body*]

This statement's semantics is completely similar to C's *while* statement. The *body* may have zero or more statements.

2.6.2.2 Compound Statement: Begin-End

The compound statement can appear anywhere that a simple expression can appear. The need for compound statements is rare in SOL because the *while* statement *body* acts as a compound statement. *if* statements rarely need to use compound statements because those statements must return values.

2.7 EXPRESSIONS

SOL is a functional language. This means that virtually all constructs return a value, the major exception being the *while*. Expressions are written in prefix notation and this is the major reason Cambridge Polish is used as the intermediate notation. In general, then, expressions are written as

[*operator operand1* ... *operandn*]

© 2007 by Taylor & Francis Group, LLC

The list of operations and their attributes is provided later in this chapter starting with Figure 2.5.

One major difference in SOL from C is the fact that *if* and *select* statements must return consistent values because the body of a function might be only a *select* statement.

With regard to stacks, because SOL will work with a stack-based environment, special notation is invented to address the two stacks: the argument-integer stack and the floating point stack. These are represented by `$`*`integer`* and `$F`*`integer`*, respectively. Reading a stack location does not alter its value; *assigning* to a stack location does alter the value.

Because SOL is a numerically oriented language, arrays are first-class citizens. Array constants are represented as

`[#arrayapp` *element1 ... elementn* `]`

Functions are defined using `let` statements. The form of a definition requires some thinking about the Cambridge Polish syntax. In Cambridge Polish, functions are written in *prefix Polish:* the function symbol comes first, followed by the arguments. For example, $a(b, c)$ would be written `[a b c]`. Therefore, a function definition would look like

`[let [ [a b c]` *definiens* `]]`

Local variables in functions (in fact, anywhere) are a special form of the `let` function.

`[let` *definiendum definiens expression* `]`

Notice that this version of the `let` has three elements exactly; the functional definition has only two.

2.8 INPUT FILE STRUCTURES AND FILE SCOPE

A file may contain multiple function definitions. A file may also contain variable statements. Variable statements outside a function are considered to have global scope outside the file but local scope in the file: a variable is not known until it is defined and it is known from that point to the end of the file.

Files may have executable statements outside function or module definitions. These are taken to be statements to be executed at the time the file is loaded by Gforth.

© 2007 by Taylor & Francis Group, LLC

typedefine	→	[*type* [#typedef *name typeterm*]]
typeterm	→	[#bar *type typeterm* | *type*]
type	→	`primitive` |
		[`#of` *name typeterm*] |
		name |
		typevariable |
		[`#tupletype` *typelist*] |
		[`#ptrdef` *type*] |
		[`#arraydef` *commalist*]
		[`#fundef` *type type*] |
		[`#object` *complexinput*] |
		[`#struct` *labeldtypelist*]

Figure 2.3 Type Sublanguage

2.9 DERIVED TYPES

SOL provides for derived types by the type statement typename: `type`.

[`type` *type-expression*]

The *type-expression* is a sublanguage in its own right (Figure 2.3).

Classes are derived types with inheritance, information hiding, and polymorphism. Objects are instances of classes (Figure 2.4). Various conventions based on Java are assumed. As usual, constructors take on the class name and are polymorphic.

Dynamic allocation is provided using `allocate` and `deallocate`, precisely like "malloc" and "free" in C. A pointer value is dereferenced using the "dot" notation outlined above. *allocate* has one argument, a type

`[class`	→	`class-name`
		[`import` *name-list*]
		[`public` *name-list*]
		[`private` *name-list*]
		optional type declaration
		optional `let` *statements for variables*
		...
		`methods`
		optional `let` *statements for variables*
		...]

Figure 2.4 Class Syntax

© 2007 by Taylor & Francis Group, LLC

name from which the space needs are created. `allocate` returns a valid pointer if space is available and 0 if space is not available. The built-in function `isnull` returns true if a pointer is 0 and false if the value is a non-zero pointer.

2.10 SCOPE AND PERSISTENCE

There are two scope types in SOL: file scope and function/compound statement scope. In general, definitions in either scope are considered known from the end of the definition statement until the end of the file (in file scope), function (in function scope), or immediately after the `begin` (in compound statement scope).

Variables defined in file scope are considered static and persist throughout the lifetime of the program. Variables defined in functions local in scope and transient (on the stack). Function names are always considered static.

2.10.1 Scope Definition

All declarations outside of functions are considered to be in the file scope. File scope is defined as follows:

- Every definition in file scope is considered local to the file unless named in an export statement. Variables, functions, and types may be shared in this way.
- Definitions from other files may be used; however, declarations must appear in an export statement in the other files. A declaration can only occur in one export statement. The using (non-global) file must use an import statement.

An export statement has the form

[`export` *name1 name2*]

Each name must be defined in the file in which the export statement occurs.

An import statement has the form

[`import` *filename name1 name2*]

The filename must be a valid SOL file and each name must be mentioned in an export statement.

2.10.2 Scoping within Functions and Compound Statements

Scoping in functions and compound statements is lexical. This means that there can be nesting of scoping (functions within functions and compound statements within functions within compound statements, *ad nauseam*). Such nesting leads to the “nearest definition” use, as in C (Figure 2.5).

© 2007 by Taylor & Francis Group, LLC

Operator Name	*Operator Symbol*	*Semantics*
	Primitive Data	
	When no Semantics are given, C is intended	
	Boolean	
And	$A \& B$	Same as && in C
Or	$A \vert B$	
Not	$!A$	
Implies	$A => B$	False when A is true and B is false
Biconditional	$A <=> B$ Logical equals	

Figure 2.5 Booleans

2.11 TYPE SYSTEM

The intent is that SOL will strongly become a type language. Unfortunately, development schedules for the early versions of the parser will not allow us to produce the code necessary to do type inference. Therefore, we will fall back on the C-type languages and require every user name to be defined using primitive or user-defined types. The type system should act like C.

2.12 PRIMITIVE FUNCTIONS AND ARGUMENTS

The normal precedence conventions hold as shown in the tables below (Figure 2.6 and Figure 2.7).

	Integers	
Plus	$A + B$	
Minus	$A - B$	
Unary Minus	$-A$	
Times	$A * B$	
Divide	A/B	
Remainder	$A\% B$	
Power	A^B	Integer A raised to an integer power B
Less	$A <$	
Equal	$A = B$	Same as C's ==
Greater	$A > B$	
Less than or equal to	$A \leq B$	
Greater than or equal to	$A \geq B$	
Not equal	$A! = B$	

Figure 2.6 Integers

© 2007 by Taylor & Francis Group, LLC

Category	*Operators*	
Assignment	`:=`	
Logical operations	`< = != <= = -> >=`	
Addition	`+ - ||`	
Times	`* / mod`	
Power	`** (this is right associative)`	
Negation	`(unary) -`	
Conversions	`type– type`	
	Floating Point	
Plus	$A + B$	
Minus	$A - B$	
Unary Minus	$-A$	
Times	$A * B$	
Divide	A/B	
Remainder	$A\%B$	
Power	$A^\wedge B$	A float, B either float or integer
Less	$A <$	
Equal	$A = B$	Same as C's ==
Greater	$A > B$	
Less than or equal to	$A \leq B$	
Greater than or equal to	$A \geq B$	
Not equal	$A\,! = B$	
Sine	sin(A)	
Cosine	cos(A)	
Tangent	tan(A)	
Exponential	exp(A)	e^A

Figure 2.7 Floats

2.13 BUILT-IN VALUES, FUNCTIONS, AND PSEUDO-FUNCTIONS

```
sin cos tan open close endoffile readbool
readint readfloat readstring writebool
writeint writefloat writestring allocate free
deallocate
```

2.13.1 Reads, Writes, and File Operations

There are three standard files: `stdin`, `stdout`, and `stderr`. Following Unix convention, these three are always considered opened when the program begins. Attempts to open or close any of the three are ignored.

© 2007 by Taylor & Francis Group, LLC

1. `open.` `open` has one argument, a string. This string designates the file to be opened and is interpreted as a Unix path. open returns one or two elements on the stack.
   a. If the top of the argument stack is true, then the next element on the argument stack is the file.
   b. If the top of the argument is false, the file failed to open; there is nothing else placed on the stack by open.
2. `close.` `close` closes the file, its single argument, and leaves nothing on the stack.
3. `endoffile.` `endoffile` takes a file as its argument and leaves as a Boolean value on the stack: true if the file is at its end; false otherwise.
4. `Read.` The four read functions (readint, for example) have a file name as their argument; the result is left on the appropriate stack.
5. `Write.` The four write statements take a file as their first argument and a value of the appropriate type as the second argument and returns the status (true if the write is successful; false otherwise) on the stack.

	Strings	
Concatenation	$A + B$	
`strcat` in C		
Insert	`insert`(A, B, C)	
Insert character B at location C in string A		
Character of	`charof`(A, B)	Return the Bth character of string A
Less	$A <$	
Equal	$A = B$	Same as C's ==
Greater	$A > B$	
Less than or equal to	$A \leq B$	
Greater than or equal to	$A \geq B$	
Not equal	$A != B$	

Figure 2.8 Strings

© 2007 by Taylor & Francis Group, LLC

	Files	
Open	`open(`*A*`)`	*A* is a path expression as a string. Returns a file value
Close	`close(`*A*`)`	Closes open file *A*
Test end of file	`endoffile(`*A*`)`	Returns true is file *A* at end of file
Read boolean value	`readbool(`*A*`)`	File*A* is read for boolean
Write a boolean value	`writebool(`*A*, *B*`)`	Write boolean value *B* to file *A*
Read integer value	`readint(`*A*`)`	File*A* is read for integer
Write a integer value	`writebool(`*A*, *B*`)`	Write integer value *B* to file *A*
Read float value	`readfloat(`*A*`)`	File*A* is read for float
Write a float value	`writefloat(`*A*, *B*`)`	Write float value *B* to file *A*
Read string value	`readstring(`*A*`)`	File*A* is read for string
Write a string value	`writestring(`*A*, *B*`)`	Write string value *B* to file *A*
	Special Values	
Pi	`Pi`	Closest representable value of $\pi = 3.14\ldots$
Standard in	`stdin`	
Standard out	`stdout`	

Figure 2.9 Files and Special Constants

validstatement	→	*toplet* | *typedefine*
toplet	→	`[LET [`*oper expression*`]]`
typedefine	→	`[`*type* `[#typedef` *name typeterm*`]]`
typeterm	→	`[#bar` *type typeterm* | *type* `]`
type	→	`primitive` |
		`[#of` *name typeterm*`]` |
		name |
		typevariable |
		`[ #tupletype` *typelist* `]` |
		`[ #ptrdef` *type* `]` |
		`[ #arraydef` *commalist* `]`
		`[ #fundef` *type type* `]` |
		`[ #object` *complexinput* `]` |
		`[ #struct` *labeldtypelist* `]`
labeledtypelist	→	*labelentry* | *labelentry*
labelentry	→	`[`*name typeterm*`]`
complexinput	→	*validstatement complexinput*
expression	→	*oper* | *statements*

Figure 2.10 Input Grammar, Part 1

© 2007 by Taylor & Francis Group, LLC

oper	→	[:= *oper oper*] |
		[: *oper type*] |
		[*binaryops oper oper*] |
		[*unaryops oper*] |
		[*oper oper*] |
		constants |
		[LAMBDA *lambdanames expression*] |
		[#arrayapp] |
		[#arrayapp *commalist*] |
		[#tupleapp |
		[#tupleapp *commalist*] |
		NAME |
		STACK
statements	→	*if* | *while* | *begin* | *let* | *select*
if	→	[IF *expression expression expression*]
		| [if *expression expression*]
while	→	[WHILE *expression exprlist*]
begin	→	[BEGIN *exprlist*]
let	→	[LET *expression expression expression*]
select	→	[SELECT *selectlist*]
selectlist	→	*selectterm* | *selecterm selectlist*
selectterm	→	[#seleccase *expression expression*]
		| [#selectcase *DEFAULT expression*]

Figure 2.11 Input Grammar, Part 2

import	→	[#import *namelist*]
namelist	→	NAME | NAME *namelist*
export	→	[#export *namelist*]
commalist	→	*expression commalist* | *expression*
exprlist	→	*expression exprlist* | *expression*
lambdanames	→	[COLON NAME *type*] *lambdanames* |
	→	[COLON NAME *type*]
typelist	→	*typeterm* | *typeterm typelist*

Figure 2.12 *SOL* Input Grammar

2.14 TYPE CONVERSIONS

Type conversions are essentially a one argument function that produces one output. The input is the value to be converted and the output is the value in the new system. Because we would like to have a general ability to

© 2007 by Taylor & Francis Group, LLC

do conversions, we adopt a class of functions of the form `name1→name2`. Whenever this format appears as the operation, the intent is that there be two operands: the first is an input value of type name1 and the output is the name of a variable of the type name2. Name1 is a value on the appropriate stack; it is converted to a value name2, which is placed on the appropriate stack.

As an example,

```
int→float 3
```

would take 3 from the argument stack and put 3.0 on the floating point stack.

Files may not be converted to any other type.

© 2007 by Taylor & Francis Group, LLC