Contents

to embed a function call almost anywhere in a program is a powerful feature. The functional nature of Scheme is one of its most distinguishing characteristics. Functions are also addressed later.

1.6 List Processing

Another distinguishing feature of Scheme (LISP) is its facility for list processing. Whereas languages such as C and Pascal require the programmer to write a library of routines for linked list manipulation, Scheme provides a host of language primitives for list processing. In general, the Scheme programmer uses the list as an abstract data type, whereas the Pascal programmer must build such structures and accompanying routines with pointer variables and pointer manipulators.

As mentioned, the list data structure is central to Scheme. A list is formed by enclosing any number of items within matching left and right parentheses. For example, the following are two Scheme lists:

```
(+ 2 34)
(a b c)
```

The first is composed of three items: a plus sign, the number 2, and the number 34. The most likely interpretation of this list is as a function call to the addition operator with two arguments, in which case the value 36 would be returned. Note that Scheme is a *prefix* language, that is, the operator is specified before the operands. There are several advantages to this, most of which are associated with language and programming flexibility. For example, a varying number of arguments is easily accommodated. That is, only one addition operator is needed to sum four numbers:

```
(+ 1 2 3 4)
```

whereas in an infix-based language, such as Pascal, multiple addition operators are required.

All we know about the second list, **(a b c)**, out of context, is that it is composed of three items. But Scheme allows us to define additional mnemonics for primitives. Although it has little mnemonic value, the "a" could represent the primitive "add", that is, the addition operator. In this case, "b" and "c" would most likely represent variables.

1.7 Function Syntax

A Scheme program is composed of one or more functions, and each function is composed of lists (possibly nested). Typically, lists contain either data, variables, or the names of other functions. For example, the following Scheme code defines a function that squares a number:


```
(define (square n) (* n n))
```

The first item is **define**, which is a Scheme keyword that signals the definition of new identifiers, in this case, a function. Scheme syntax prescribes a prefix-like specification. The second item, a list, names the new function and specifies its parameters – in this case, one parameter, **n**. Subsequent items constitute the function body. In this case, **square** is defined to be the multiplication of a number, the argument/parameter, **n**, by itself.

In the previous function there is no need to specify which function is the "main" function and which functions are subordinate; this issue is resolved at execution time. That is, whether or not **square** will be used by itself, or called by some other function, is immaterial at the time of its creation. In this sense, all functions are created equal. Thus, the programmer is free to think about a function's function, and is less burdened by syntax. In contrast, interacting Pascal subprograms must be arranged in the proper order before compilation. Scheme programmers never have to worry about such syntactic issues during program development.

Note that user-defined functions can have a variable number of arguments; they are defined by a syntactic structure which accommodates a varying number of "arguments;" and the function call "(square n)", that is, its usage, is identical to its specification in the call to **define**. Hence, there is a degree of symmetry and consistency in Scheme that is lacking in many other languages.

1.8 (((((((0)))))))

Scheme employs parentheses to define the bounds of a list. Although nested parentheses can be cumbersome, most languages have syntax-related language components that can be awkward. Moreover, most Scheme systems provide direct assistance with parentheses balancing.

As mentioned, both data and functions are specified in list form. This has many advantages. One of the most important advantages is that it allows programs to create other programs, and this is an especially powerful feature for AI programming. This power is completely absent from many "syntax-bound" languages, such as C, Pascal, and PL/I, and is not readily appreciated by those new to LISP programming.

1.9 Symbolic Processing

Scheme (LISP) differs from other languages in its facility for working with symbols. In more traditional languages we think of identifiers as symbols. In fact, the term "symbol table" is used for the compiler's temporary array of identifier information. As with other languages, Scheme variables and procedures exist as symbols; but in addition, Scheme provides a facility for processing alternative symbols, namely, symbols that are not assigned a value.