Naive Semantics

At this point, I don't want you to be bogged down by the syntax of the grammar. You only need to recognize the terminals and tokenize these terminals, which will be passed to the parser. The terminals are shown in bold, and the italicized words are the non-terminals in the grammar. Identifiers, including variable and function names follow the C/C++.

Constants

There are four classes of constants: boolean, integer, real, and string.

Boolean

The Booleans consist of two constants, **true** and **false.** The normal complement of Boolean operators is assumed: **and, or, not.**

Integers

The integers are the normal machine integers with the normal operations of **+, -, *,** and **/.** The modulus operator is **%.** Power is denoted by the **^.** The relational operators are equality =, less than <, greater than >, less than equal to <=, greater than equal to >=, and not equal to **!**=.

Reals

The reals have the same operations and relations on double precision numbers as the integers. The modulus operator returns the fractional part of the real number. For basic trigonometry functions we have **sin, cos, tan.**

Strings

There is only one operator for strings: concatenation denoted by +. Strings have the same relation symbols as the numbers.

Types and Variables

Primitive Types

There are four primitive types: one each corresponding to the four classes of constants: **bool, int, real,** and **string.**

Variable Lists

Variables are introduced using the *let* statement, and the variable names follow the C/C++ standard. A *var-list* for the *let* statement is a list of *var-list*s, and a variable list is the form used to associate a type with a variable name. The form is **[:** *var-name type***]**. The list of variable lists looks something like: **[[a int][b real]]** , which declares that **a** is an **int** and **b** is a **real**

Expressions

Basically everything in the language can be considered a function, which is why the class of languages including Lisp is called *functional languages*. The expression syntax is the list: **[ ]**. For example **[+ 1 2]** is to be interpreted as the obvious infix analogue **1+2.** Other examples from arithmetic are **[sin x]** and **[+ 1 [* 2 3]].** This notation is called *Cambridge polish* .

Statements

Certain special forms are included in the language to make it easier to program. Those forms are discussed here.

Print Statement

The **stdout**statement has the form **[stdout** *atom***]** or [stdout *expression***]**. The stdout-statement prints the results of a constant, variable, or expression followed by a newline.

If Statement

The *if*-statement is written as a three or four element list. **[if** *expression expression***]** is to be interpreted as follows: If $expr_1$ is evaluated to *true,* then $expr_2$ should be evaluated; otherwise, there is no action. Similarly, the four element list would be **[if** *expression expression expression***]** would evaluate $expr_2$ if the condition $expr_1$ is *true* and evaluate $expr_3$ if *false*.

While Statement

The *while*-statement has a three component list: **[while** *expression exprlist***]** with the obvious analogue to the normal *while* statement: *exprlist* is executed until *expression* is *false*. If *expression* is false initially, then the *exprlist* isn't executed.

Let Statement

The *let*-statement introduces variables and functions into a computation and delineates the scope for those variables. The *let* statement has the form **[let** [*var-list***]]**

Assign Statement

The *assign*-statement is used to set a variable's value. It is written as **[:=** *lvalue rvalue***]**

Note on the Symbol Table

The symbol table is used by every phase of the process. For starters, the symbol table should be thought of as an SQL style database. This means that conceptually the individual atoms are associated with various definitions that the various phases need to do their job. For example, the type checker must be able to retrieve all types associated with an atom and the interpreter must be able to associate an atom with a value.

In implementation, the symbol table can use *association lists* as the organizing principle. The most obvious lookup mechanism is a hash table. Notice that lookup of an atom is a very small piece of the symbol table processing.