SomeLife: A Simple Language for Practice Implementation

1 Introduction

This document describes the SomeLife programming language. SomeLife is a simple programming language designed for practice implementation. SomeLife is a simplified version of Pascal in which one may perform simple integer and floating point calculations, and simple function calls.

SomeLife supports two basic data types: *integer* and *floating point*. Each of these types may be aggregated into one dimensional arrays. A number of operators are defined for each type. You can assume that the underlying hardware supports integers with 32 bit twos complement arithmetic and floating point with a 32 bit implementation of the IEEE floating point standard.

Control structures in SomeLife are limited. It has an *if* statement, a *while* statement and a *compound* statement. SomeLife only supports one level of nested functions which contain no arguments.

The language is intended to be *strongly typed*; that is, the type of each expression should be determinable at compile time. However, some *coercions* from one type to another will be permitted. Since there is no *boolean* data type, integers are used as logicals in a manner similar to C.

2 Lexical Properties of SomeLife

1. In SomeLife, blanks are significant.
2. In SomeLife, keywords always consist of capital letters. All keywords are reserved; that is, the programmer cannot use a SomeLife keyword as the name of a variable. The valid keywords are: `AND`, `ARRAY`, `BEGIN`, `DO`, `ELSE`, `END`, `FLOAT`, `IF`, `INTEGER`, `DIV`, `NOT`, `OR`, `PROGRAM`, `FUNCTION`, `READ`, `THEN`, `VAR`, `WHILE`, `WRITE`. (Note that SomeLife is *case sensitive,* that is, the variable `X` differs from `x`. Thus, `END` is a keyword, but `end` can be a variable name.)
3. The following special characters have meanings in a SomeLife program. (See the grammar and notes for details.)
   { } ' < > = + − ∗ [ ] ( ) . , ;
4. Comments are delimited by the characters { and }. A { begins a comment; it is valid in no other context. A } ends a comment; it cannot appear inside a comment. (This means comments may not be nested. { can appear in a comment; the first } closes the comment.) Comments may appear before or after any other token.
5. Identifiers are written with upper and lowercase letters and are defined as follows:
   $\langle Letter \rangle \rightarrow$ `a` | `b` | `c` | $\cdots$ | `z` | `A` | `B` | $\cdots$ | `Z`
   $\langle Digit \rangle \rightarrow$ `0` | `1` | `2` | $\cdots$ | `9`
   $\langle Identifier \rangle \rightarrow \langle Letter \rangle \; (\langle Letter \rangle \mid \langle Digit \rangle)^*$
   The implementor may restrict the length of identifiers so long as identifiers of at least 31 characters are legal.
6. Constants are defined as follows:
   $\langle Positive \rangle \rightarrow$ `1` | `2` | `3` | ... | `9`
   $\langle Sign \rangle \rightarrow$ `+` | `-` | ϵ
   $\langle Intnum \rangle \rightarrow \langle Positive \rangle \; \langle Digit \rangle^* \mid$ `0`
   $\langle Floatnum \rangle \rightarrow \langle Intnum \rangle$ `.`
   | $\langle Intnum \rangle$ `.` $\langle Intnum \rangle$
   | $\langle Intnum \rangle$ `.` `E` $\langle Sign \rangle \; \langle Intnum \rangle$
   | $\langle Intnum \rangle$ `.` $\langle Intnum \rangle$ `E` $\langle Sign \rangle \; \langle Intnum \rangle$
   Special string constants are acceptable in WRITE statements:
   $\langle StringConstant \rangle \rightarrow \; '\langle Letter \rangle^{*}{}'$

3 SomeLife Syntax

This section gives a syntactical description of SomeLife. The sections following the grammar provide implementation notes on the various parts of the grammar.

3.1 BNF

The following grammar describes the context-free syntax of SomeLife:

$$
\begin{array}{lcl}
\langle \mathit{Program} \rangle & \rightarrow & \texttt{PROGRAM}\ \langle \mathit{Identifier} \rangle\ ; \\
 & & \langle \mathit{Decls} \rangle \\
 & & \langle \mathit{SubprogramDecls} \rangle \\
 & & \langle \mathit{CompoundStatement} \rangle\ \texttt{.} \\
\langle \mathit{Decls} \rangle & \rightarrow & \texttt{VAR}\ \langle \mathit{DeclList} \rangle \\
 & | & \epsilon \\
\langle \mathit{DeclList} \rangle & \rightarrow & \langle \mathit{IdentifierList} \rangle\ \texttt{:}\ \langle \mathit{Type} \rangle\ \texttt{;} \\
 & | & \langle \mathit{DeclList} \rangle\ \langle \mathit{IdentifierList} \rangle\ \texttt{:}\ \langle \mathit{Type} \rangle\ \texttt{;} \\
\langle \mathit{IdentifierList} \rangle & \rightarrow & \langle \mathit{Identifier} \rangle \\
 & | & \langle \mathit{IdentifierList} \rangle\ \texttt{,}\ \langle \mathit{Identifier} \rangle \\
\langle \mathit{Type} \rangle & \rightarrow & \langle \mathit{StandardType} \rangle \\
 & | & \langle \mathit{ArrayType} \rangle \\
\langle \mathit{StandardType} \rangle & \rightarrow & \texttt{INTEGER} \\
 & | & \texttt{FLOAT} \\
\langle \mathit{ArrayType} \rangle & \rightarrow & \texttt{ARRAY [}\ \langle \mathit{Dim} \rangle\ \texttt{] OF}\ \langle \mathit{StandardType} \rangle \\
\langle \mathit{Dim} \rangle & \rightarrow & \langle \mathit{Intnum} \rangle\ \texttt{..}\ \langle \mathit{Intnum} \rangle \\
\langle \mathit{SubProgramDecls} \rangle & \rightarrow & \langle\ \mathit{SubProgramDecls}\ \rangle\ \langle \mathit{SubProgramDecl} \rangle \\
 & | & \epsilon \\
\langle \mathit{SubprogramDecl} \rangle & \rightarrow & \langle \mathit{SubprogramHead} \rangle\ \langle \mathit{Decls} \rangle \langle \mathit{CompoundStatement} \rangle \\
\langle \mathit{SubprogramHead} \rangle & \rightarrow & \texttt{FUNCTION}\ \langle \mathit{Identifier} \rangle \texttt{:} \langle \mathit{StandardType} \rangle \\
\langle \mathit{Statement} \rangle & \rightarrow & \langle \mathit{Assignment} \rangle \\
 & | & \langle \mathit{IfStatement} \rangle \\
 & | & \langle \mathit{WhileStatement} \rangle \\
 & | & \langle \mathit{IOStatement} \rangle \\
 & | & \langle \mathit{CompoundStatement} \rangle
\end{array}
$$

⟨*Assignment*⟩	→	⟨*Variable*⟩ `:=` ⟨*Expr*⟩
⟨*IfStatement*⟩	→	`IF` ⟨*Expr*⟩
		`THEN` ⟨*Statement*⟩
		`ELSE` ⟨*Statement*⟩
	|	`IF` ⟨*Expr*⟩ `THEN` ⟨*Statement*⟩
WhileStatement	→	`WHILE` ⟨*Expr*⟩ `DO` ⟨*Statement*⟩
⟨*IOStatement*⟩	→	`READ (` ⟨*Variable*⟩ `)`
	|	`WRITE (` ⟨*Expr*⟩ `)`
	|	`WRITE (` ⟨*StringConstant*⟩ `)`
⟨*CompoundStatement*⟩	→	`BEGIN` ⟨*StatementList*⟩ `END`
⟨*StatementList*⟩	→	⟨*Statement*⟩
	|	⟨*StatementList*⟩ `;` ⟨*Statement*⟩
⟨*Expr*⟩	→	⟨*Expr*⟩ ⟨*LogOp*⟩ ⟨*RelExpr*⟩
	|	⟨*RelExpr*⟩
⟨*Logop*⟩	→	`OR` | `AND`
⟨*RelExpr*⟩	→	⟨*RelExpr*⟩ ⟨*RelOp*⟩ ⟨*AddExpr*⟩
	|	⟨*AddExpr*⟩
⟨*Relop*⟩	→	`<` | `<=` | `>=` | `>` | `=` | `<>`
⟨*AddExpr*⟩	→	⟨*AddExpr*⟩ ⟨*AddOp*⟩ ⟨*MulExpr*⟩
	|	⟨*MulExpr*⟩
⟨*Addop*⟩	→	`+` | `-`
⟨*MulExpr*⟩	→	⟨*MulExpr*⟩ ⟨*MulOp*⟩ ⟨*Factor*⟩
	|	⟨*Factor*⟩
⟨*Mulop*⟩	→	`*` | `DIV`
⟨*Factor*⟩	→	⟨*Variable*⟩
	|	⟨*Constant*⟩
	|	`NOT` ⟨*Factor*⟩
	|	`(` ⟨*Expr*⟩ `)`
⟨*Variable*⟩	→	⟨*Identifier*⟩
	|	⟨*Identifier*⟩ `[` ⟨*Expr*⟩ `]`
⟨*Constant*⟩	→	⟨*Intnum*⟩ | ⟨*Floatnum*⟩

3.2 Section Notes

3.2.1 Declarations

SomeLife has two standard types: `INTEGER` and `FLOAT`. Integers and floats occupy in a single X86-64 machine "double word" which consists of four bytes. These standard types may be composed into the structured `ARRAY` type. An identifier may represent one of four types of objects:

1. an integer variable or array
2. a floating point variable or array

Identifiers are declared to be variables or arrays by a `VAR` declaration. Only singly dimensioned arrays are permitted in SomeLife, but arbitrary upper and lower index bounds are permitted.

Example:

```
VAR x,y : INTEGER;
   f1, f2, f3 : FLOAT;
   a : ARRAY [ 1 .. 15 ] OF INTEGER;
   s1, s2 : ARRAY [0 .. 79 ] OF FLOAT;
```

3.2.2 Assignment Statement

The assignment statement requires that the *left hand side* (the ⟨*Variable*⟩ non-terminal) and *right hand side* (the ⟨*Expr*⟩ non-terminal) evaluate to have the same type. If they have different types, either coercion is required or a context-sensitive error has occurred. The coercion rules for assignment are simple. If both sides are numeric (of type `INTEGER` or `FLOAT`), the right hand side is converted to the type of the left hand side.

3.2.3 If Statement

The grammar for the `IF-THEN-ELSE` construct embodies one of the classical solutions to the dangling else ambiguity. It provides a unique binding of the *else-part* to a corresponding *if* and *then-part*.

To evaluate an if statement, the expression is evaluated. If the expression's type is `FLOAT`, it should be converted to an `INTEGER`. For an integer value, SomeLife defines `0` as *false*; any other value is equivalent to *true*.

Examples:

```
IF c=d THEN d := a
IF b=0 THEN b := 2*a ELSE b := b/2
```

3.2.4 While Statement

The while statement provides a simple mechanism for iteration. SomeLife's while statement behaves like the while statement in many other languages; it executes the statement in the loop's body until the controlling expression becomes false.

The controlling expression will be treated as a boolean value encoded into an `INTEGER` expression. If the expression is not of type `INTEGER`, the same coercion rules apply as in the if statement.

3.2.5 Expressions

SomeLife expressions compute simple values of type `INTEGER` or `FLOAT`. For both integer and floating point numbers, addition, multiplication, division, and comparison are defined.

Coercion: If an expression contains operands of only one type, evaluation is straight forward. When an operand contains mixed types, the situation is more complex. If an *Addop* or *Mulop* has an `INTEGER` operand and a `FLOAT` operand, the `INTEGER` operand should be converted to a `FLOAT` before the operation is performed.

Relational operators always produce an integer. Comparisons between integers and floats produce integer results. To perform the comparison, the integer is converted to a float. For the numbers, comparison is based on both sign and magnitude.

Note: in an assignment, the value of a numeric expression gets converted to match the type of the variable that appears on its left hand side.

Booleans: Because SomeLife has no booleans, relational expressions are defined to yield integer results. Thus, a relational expression of the form `a = b` is considered to be an arithmetic expression whose value is `TRUE` if the relation holds and `0` otherwise. Hence, both the `IF-THEN-ELSE` and `WHILE` statements test integer values; the expression is considered *false* if it evaluates to `0` and to `TRUE` if it evaluates to anything else. Consider the following example which tests for either of two conditions being true:

```
BEGIN
    READ (a); READ (b); READ (c); READ (d);
    IF (a = b) + (c < d) THEN WRITE ('error')
END
```

Note that relational expressions must be enclosed in parentheses because they have very low precedence. In the above example, `a`, `b`, `c`, and `d` may be variables of any type.

In the above example, the special operator `OR` could have been used. In SomeLife the operator `OR` takes two integer operands, two floating-pointer operands, or an integer operand and a floating-pointer operand. `OR` produces the result `0` if both operands evaluate to `0`; otherwise, it produces `TRUE`. The operator `AND` evaluates to `TRUE` if both operands are nonzero; otherwise it evaluates to `0`. The unary logical operator `NOT` evaluates to `TRUE` if its argument is zero and to `0` otherwise. The operand of `NOT` must be an integer.

4 An Example Program

The following program represents a simple example program written in SomeLife. This program successively reads pairs of integers from the input file and prints out their greatest common divisor.

```
PROGRAM example;
   VAR x, y : INTEGER;
   FUNCTION gcd :INTEGER;
      VAR t : INTEGER;
      BEGIN
         IF y=0
            THEN RETURN x
            ELSE BEGIN
              t := x;
              x := y;
              y := t - y * (t DIV y);
              RETURN gcd()
            END
       END;
      BEGIN
         READ (x);
         READ (y);
         WHILE (x <> 0) OR (y <> 0) DO
            BEGIN
               WRITE (gcd());
               READ (x);
               READ (y)
             END
      END.
```