EECS 6083: Project Description

The scanner will process and discard all whitespace and comments. We will assume a C++ short comment style that start with the string "//" and continue to the next newline character and block comments of the form "/* …. */". Block comments can be nested. In addition to comments, whitespace is defined as the space character, newline characters, and tab characters. Terminals are in bold font and non-terminals are placed in angled brackets "<>"; be careful not to confuse BNF operators with terminals. If unclear, ask. The start state for this grammar is the very first grammar rule below. Be careful to note the period character at the end of a program.

Syntax

<program> ::=
    <program_header> <program_body> **.**

<program_header> ::= **program** <identifier> **is**

<program_body> ::=
        (<declaration> **;**)*
    begin
        (<statement> **;**)*
    end program

<declaration> ::=
    [**global**] <procedure_declaration>
| [**global**] <variable_declaration>

<procedure_declaration> ::=
  <procedure_header> <procedure_body>

<procedure_header> :: =
    procedure <identifier> **:** <type_mark>
        ([<parameter_list>] **)**

<parameter_list> ::=
    <parameter> **,** <parameter_list>
| <parameter>

<parameter> ::= <variable_declaration>

<procedure_body> ::=
        (<declaration> **;**)*
    begin
        (<statement> **;**)*
    end procedure

<variable_declaration> ::=
    variable <identifier> **:** <type_mark>
    [**[** <bound> **]**]

<type_mark>
    integer | **float** | **string** | **bool**

<bound> ::= <number>

<statement> ::=
    <assignment_statement>
| <if_statement>
| <loop_statement>
| <return_statement>

<procedure_call> ::=
    <identifier> **(** [<argument_list>] **)**

<assignment_statement> ::=
    <destination> := <expression>

<destination> ::=
    <identifier> [**[** <expression> **]**]

<if_statement> ::=
    if (<expression> **) then** (<statement> **;**)*
    [**else** (<statement> **;**)*]
    end if

<loop_statement> ::=
    for (<assignment_statement> **;**
        <expression> **)**
        (<statement> **;**)*
    end for

<return_statement> ::= **return** <expression>

<identifier> ::= [**a**-**z**A-**Z**][**a**-**z**A-**Z**0-**9**_]*

<expression> ::=
    <expression> **&** <arithOp>
| <expression> | <arithOp>
| [**not**] <arithOp>

<arithOp> ::=
    <arithOp> + <relation>
| <arithOp> - <relation>
| <relation>

<relation> ::=
    <relation> < <term>
| <relation> >= <term>
| <relation> <= <term>
| <relation> > <term>
| <relation> == <term>
| <relation> != <term>
| <term>

<term> ::=
    <term> * <factor>
| <term> / <factor>
| <factor>

<factor> ::=
    (<expression> **)**
| <procedure_call>
| [-] <name>
| [-] <number>
| <string>
| **true**
| **false**

<name> ::=
    <identifier> [**[** <expression> **]**]

<argument_list> ::=
    <expression> **,** <argument_list>

| <expression>

<number> ::= [**0-9**][**0-9_**]*[**.**[**0-9_**]*]

<string> :: = **“**[^”]***”**

1. Semantics

1. Procedure parameters are transmitted by value. Recursion is supported.
2. Identifiers and reserved words are case insensitive (e.g., tmp, Tmp, TMP all denote the same identifier name). Your compiler (scanner) should probably map everything (except strings) into upper or lower case letters to make your work easier.
3. All variables/procedures defined in the outermost scope (of **program**) are in the global scope (independent of the presence/ absence of the **global** reserved word). Any variable or procedure with the **global** reserved word prefix is a member of the global scope (no matter where it is described). All variables and procedures declared without the **global** prefix are local to that procedure and not visible outside of that procedure. Global names must be unique. Local names (procedures or variables) can be reused in other procedures.
4. Procedures/functions currently being defined are visible in the statement set of the procedure/function itself (so that recursive calls a possible). Otherwise forward references are not permitted (you cannot reference a global variable at the beginning of the program that is not already declared before that reference).
5. No forward references are permitted or supported.
6. Expressions are strongly typed and types must match. However there is automatic conversion in the arithmetic operators to allow any mixing between integers and floats. Furthermore, the relational operators can compare booleans with integers (booleans are converted to integers as: false → 0, true → 1; integers are converted to bools as: the integer value 0 is converted to false, all other integer values are converted to true).
7. The type signatures of a procedures arguments must match exactly their parameter declaration.
8. Assignment of array variables (copying the entire array) are permitted provided the source and destination array types are of the same length. Operators on unqualified array references apply to the entire array (*e.g.*, a[i]:=a[i]+1 increment array entry i; a:= a+1 [where a is an array] increment all elements of the array by 1; and a:=b+c [where, a, b, and c are all arrays of length n] performs an element by element addition of arrays b and c and stores the resulting array into a).
9. Arithmetic operations (add, sub, multiply, divide) are defined for integers and floats only. The bitwise and “&”, bitwise or “|”, and bitwise **not** operators are valid only on variables of type integer. Finally boolean operators for the logical operations &, |, and **not** are supported.
10. Relational operations are defined for integers, floats, and booleans. Only comparisons between the compatible types is possible. Relational operators return a boolean result. Equality/inequality tests between strings are supported, but other relational tests between strings are not legal. The equality test is on the string, not on the pointer to the string.

11. In general, you should treat string variables as pointers to a null terminated string that is stored in your memory space. Strings are then read/written to/from this space using the get/put functions. The in memory string values cannot be destroyed or changed (although string variables can be assigned to point to different strings).
12. Array indexes must be of type integer. Bounds checking to insure that the index is within the upper and lower bound (always 0) is required for all indexed array references.
13. The target and expression of assignment statements must, in general, be of the same type. However, (i) bool and integer types are compatible, and (ii) integer and float types are compatible. Expressions (in assignment statements) with compatible types will be cast to the type of the target.
14. The expression on a for or if statement must resolve to a bool. Expressions of type integer are cast to a bool for evaluation.

2. Builtin Functions

The language has the following built I/O functions:

getBool() : bool value

getInteger() : integer value

getFloat() : float value

getString() : string value

putBool(bool value) : bool

putInteger(integer value) : bool

putFloat(float value) : bool

putString(string value) : bool

sqrt (integer value) : float

The put operations do not return a value.

These functions read/write input/output to standard in and standard out. If you prefer, these routines can read/write to named files such as “input” and “output”.

3. Change Log

Revision 0: 1/10/21

- Initial version.

Revision 1: 1/29/21

- Underlined colon character to denote it as a terminal in <procedure_header> and <variable_declaration>.

Revision 2: 2/18/21

- Clarified global variables and scoping.

Revision 3: 3/2/21

- Removed enum declaration from <type_mark> rule and created a new rule (<type_def>) to include it in <type_declaration>. This removes the possibility of having an unnamed enum in a function argument list.

Revision 4: 3/4/21

- Removed type declarations entirely from the language.

Revision 5: 3/9/21

- Removed identifier from type_mark as it is no longer required.
- Clarified scoping rules. Explicitly disallowed forward references.