EECS 6083: Project Description

Comments

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>
        ([<parameter_list>] **)**

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

<parameter> ::=
    <variable_declaration> (**in** | **out | inout**)

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

<variable_declaration> ::=
    <type_mark> <identifier>
    [**[** <array_size> **]**]

<type_mark> ::=
    integer
| **float**
| **bool**
| **string**
| **char**

<array_size> ::= <number>

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

<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**

<identifier> ::= [**a**-**zA**-**Z**][**a**-**zA**-**Z0**-**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> **)**
| [-] <name>
| [-] <number>
| <string>
| <char>
| **true**

| **false**

<name> ::=

<identifier> [**[** <expression> **]**]

<argument_list> ::=

<expression> **,** <argument_list>

| <expression>

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

<string> :: = **"**[**a-zA-Z0-9 _,;:.'**]***"**

<char> ::= **'**[**a-zA-Z0-9 _;:."**]**'**

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 and chars) into upper or lower case letters to make your work easier.
3. Non-local variables and functions are not visible except for those variables and functions in the outermost scope prefixed with the **global** reserved word. Functions currently being defined are visible in the statement set of the function itself (so that recursive calls a possible).
4. No forward references are permitted or supported.
5. 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 (integers are converted to boolean as: false → 0, true → 1; integer values other than 0 and 1 should throw a runtime data conversion error).
6. The type signatures of a procedures arguments must match exactly their parameter declaration.
7. Arithmetic operations (add, sub, multiply, divide, bitwise and "&", bitwise or "|") 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.
8. Relational operations are defined for integers and booleans. Only comparisons between the compatible types is possible . Relational operators return a boolean result.
9. 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).

2. Builtin Functions

The language has the following built I/O functions:

bool getBool()

integer getInteger()

string getFloat()

string getString()

char getChar()

integer putBool(bool)

integer putInteger(integer)

integer putFloat(float)

integer putString(string)

integer putChar(char)

The put operations always returns the value 0.

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: 1/19/16

- Initial version.