ECE 683/685: 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. 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.

Syntax

<function_declaration> ::=
  <function_header> <function_body>

<function_header> :: =
  <type_mark> **function**
  <identifier> **(** [<parameter_list>] **)**

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

<parameter> ::= <variable_declaration>

<function_body> ::=
    (<declaration> **;**)*
  begin
    (<statement> **;**)*
  end function

<declaration> ::=
  <function_declaration>
| <variable_declaration>

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

<type_mark> ::=
  integer
| **float**
| **boolean**
| **string**

<array_size> ::= <number>

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

<function_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_stmt> <expression>
        (<statement> **;**)*
    end for

<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>
| <term>

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

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

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

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

<number> ::= [**0**-**9**]+

<string> :: = **“**[**a**-**zA**-**Z0**-**9**_]***”**

1. Semantics

1. Procedure parameters are transmitted by value. Recursion is supported.
2. Non-local variables are not visible except for those variables in the outermost scope prefixed with the **global** reserved word.
3. No forward references are permitted or supported.

4. 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).
5. The type signatures of a procedures arguments must match exactly their parameter declaration.
6. 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.
7. Relational operations are defined for integers and booleans. Only comparisons between the compatible types is possible . Relational operators return a boolean result.
8. 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).
9. Parameter transmission is by value for integers and booleans; strings are passed by sending the memory pointer to the string.
10. Function return and return values are achieved by an assignment to the function name. Thus, assigning a value to the function names is both defining the value to return *and* triggering a return action from the function call.
11. A function can issue a return value only for itself and not for the containing function names.

2. Code Generation

The code generator is to output a restricted form of C that looks much like a 3-address load/store architecture. You can assume an unbounded set of registers, a 32M memory space containing space for static memory and stack memory. Your machine code should look something like (I forget C syntax, so you may have to translate this to real C):

```
Reg[3] = MM[Reg[SP]];
Reg[SP] = Reg[SP] – 2;
Reg[4] = MM[12]; // assume a static
                 // variable at
                 // location 12
Reg[5] = Reg[3] + Reg[4]
MM[12] = Reg[5];
```

You must use simple C: assignment statements, goto statements, and if statements. No procedures, switch statements, etc.

You must evaluate the conditional expressions in "if statements" and simply reference the result (stored in a register) in the if statement of your generated C code.

Basically you should generate C code that looks like a simple 3-address assembly language.

3. Runtime Support

There are several (globally visible) predefined procedures provided by the runtime support environment, namely:

- boolean getBool()
- integer getInt()
- string getString()
- integer putBool(boolean)
- integer putInt(integer)

- integer putString(string)
- integer sqrt(integer)

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”. The int2bool converts an integer zero to the boolean value 0 and all non-zero values to a boolean value 1.

4. Final Report

In addition to sending me your compiler source and build environment that I can run on my linux workstation (you are responsible for ensuring that it will build on a standard linux box; if you build it on some exotic system like Haiku, we can discuss a demo on that platform), you must also turn in a one page report documenting your compiler. It should document your software, its structure, the build process, and highlight any unique features you have implemented in the system.