CSCI 2400 – Models of Computation

Project 2

Interpreter for Snail

1 Overview

In this assignment you will use the parser generator **yacc** to construct an *interpreter* for a language called *Snail* containing the set of statements listed below. The interpreter executes the statements in a Snail program, one after the other in the order which they appear in the Snail program.

In Section 2 we describe the various statements. In Section 3 we give the grammar of the *Simple* language. In Section 4 we describe what your yacc code will do. In Section 5 are instructions to hand-in your assignment.

2 The Snail Programming Language

Snail is a very simple programming language. The body of a *Snail* program consists of a sequence of statements. There are three kinds of statements: *assign*, *print*, and *if*. A basic component for all kinds of statements is the *expression*. The expression and the statements are described below.

- **expression**

  An expression is any mathematical expression made from identifiers, integers, parenthesis, the arithmetic operators

  ```
  +   -   *   /
  ```

  and the comparison operators

  ```
  <  >  <=  >=  ==  !=
  ```

  For example, this is a valid expression:

  ```
  10 + 20 * (10 < 3)
  ```

  The *value* of an expression is obtained by executing all the arithmetic operations in the expression. The result of a comparison operation is 1 if the comparison result is true, and 0 otherwise. For example, the above expression has value 10 (since, $10 < 3 = 0$).

  The value of an identifier is the last value assigned to it in an assign statement. An identifier which hasn't been assigned a value before cannot be used inside an expression and in this case you should report an error message.

- **assign statement**

  The assign statement has the form:

  ```
  identifier = expression;
  ```

  For example, this is a valid assign statement:

  ```
  var1 = 20 - 3*2;
  ```

  In the assign statement the identifier gets the value of the expression. As an example, in the above assign statement the new value of variable `var1` is 14.

- **print statement**

  The print statement print messages on the screen. The print statement has one of following forms:

  ```
  print ‘‘string’’;     //prints the string
  print newline;        //prints a newline character
  print expression;     //prints the expression value
  ```

  For example, the execution of the following statements

  ```
  print ‘‘The value of 10*5 is ’’;
  print 10*5;
  ```

  produces the output:

  ```
  The value of 10*5 is 50
  ```

- **if statement**

  An if statement has one form:

  ```
  if expression then         //if-then-else statement
    statement
    ...                      //more statements
  else
    statement
    ...                      //more statements
  endif
  ```

  The “if-then-else” statement means that if the expression value is not 0 then the statements between `then` and `else` will be executed, and otherwise, if the expression value is 0, the statements between `else` and `endif` will be executed. For example, the following is a valid if statement:

```
    if (x < 10) then
      print ``x is smaller than ten'';
      x = x - y + 20;
    else
      x = 10* y;
    endif
```

A Snail program is a sequence of statements and has the following general form:

```
statement
statement
  ...
statement
```

We can have comments in a Snail program right after "//" (as in a C++ program). An simple example Snail program is the following:

```
x = 10;
y = 20;
print ``the value of x is ''; print x; print newline;
print ``the value of y is ''; print y; print newline;
print ``the sum of x and y is ''; print x + y; print newline;

if x < y then                            //test x < y
  print ``x is smaller than y'';         // x is smaller
else
  print ``x is bigger or equal than y''  // x is not smaller
endif
```

The output of the program is:

```
the value of x is 10
the value of y is 20
the sum of x and y is 30;
x is smaller than y;
```

3 Snail Grammar

All the Snail programs can be described by the context-free grammar given below. The start variable is `program`, the grammar variables are in small letters, and the terminals in capital letters. Notice that this grammar is ambiguous in the `expr` variable; however, all the ambiguities can be removed using the precedence rules of yacc.

```
program -> stmt_list
```

```
stmt_list -> stmt_list stmt
          |  stmt

stmt -> assign_stmt
     |  print_stmt
     |  if_stmt

assign_stmt -> ID = expr ;

print_stmt -> PRINT expr ;
           |  PRINT STRING ;
           |  PRINT NEWLINE ;

if_stmt ->  IF expr THEN stmt_list ELSE stmt_list ENDIF

expr -> ( expr )
     |  expr + expr
     |  expr - expr
     |  expr * expr
     |  expr / expr
     |  expr < expr
     |  expr > expr
     |  expr <= expr
     |  expr >= expr
     |  expr == expr
     |  expr != expr
     |  - expr
     |  INT
     |  ID
```

4 Yacc Code

You will write a yacc code which implements the interpreter for Snail programs. The main part of your yacc code will consist of the snail grammar augmented with actions.

- **expression**

  Whenever you find an expression, you need to evaluate the expression. This can be done in a bottom up fashion with actions such as:

  ```
  expr : expr '+' expr {$$ = $1 + $3;}
  ```

- **assign statement**

  The value of an `ID` (identifier) node is stored in the symbol table. For an `assign` node you need to update the value of the variable to be equal to the `expr` of the right hand side of the assign statement.

- **print statement**

  For a `print` node you just print on the output the contents of the `STRING` stripped from quotes, or the value of the `expr`, or a newline character.

- **if statement**

  For the if statement you first evaluate the expr. If the value of expr is true (not zero), then you execute the if part of the statement, otherwise you execute the else part of the statement. You need to be careful not to execute the part of the if statement you don't want to execute. In the following example, when expression is true then only statement1 will be executed, while when expression is false then only statement2 will be executed.

  ```
  if expression then
    statement1
  else
    statement2
  endif
  ```

  In order to achieve this, you need to have a stack which will control whether or not to execute particular statements. We push true for the part of the if which will be executed; for the other part we push false. The stack is needed in order to handle nested if statements. The code to handle this is as follows:

  ```
  if_stmt :  IF expr
             THEN {push($2 != 0);} stmt_list {pop();}
             ELSE {push($2 == 0);} stmt_list {pop();} ENDIF
  ```

  If the top of the stack contains true then statements can be executed and expressions can be evaluated (these statements and expressions are inside the part of an if statement that will be executed). For example, the actions of the plus expression should be modified as:

  ```
  expr : expr '+' expr {if (top() == 1) $$ = $1 + $3;}
  ```

  In order to handle statements and expressions in the main body, outside from if statements, you need to push true at the beginning of the execution of the interpreter.

If you find a syntax error in the input program then report an error message with the line number where you found the error and abort the program.

Your yacc program will use parts of the the lexical analyzer you built in the first project and for this you need to modify appropriately the lex code.

5 Hand-In

You need to submit your lex and yacc program through cs web submission. In the course web page you can also find example yacc programs (together with lex programs) that will help you to get started with your project.