

Programming Assignment 2 – Yacc

Return Day: Monday, December 4.

Teams: You are encouraged to form teams of 2 persons.

1 Overview

In this assignment you will use the parser generator **yacc** to construct an *interpreter* for the *Snail* programming language which we describe below. The interpreter executes the statements of a Snail program in sequence as they appear in the program.

In Section 2 we describe the Snail programming language. In Section 3 we give the grammar of the Snail language. In Section 4 we describe what your yacc code will do. In Section 5 are instructions to handin your assignment.

2 Snail Programming Language

Snail is a very simple programming language. The body of a Snail program consists from a sequence of statements. There are four kinds of statements: *assign*, *print*, *if*, and *while*. 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 two forms:

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

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

The “if-then” form means that if the expression value is not 0 then the sequence of statements between `then` and `endif` will be executed. The “if-then-else” form 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
```

- **while statement:**

A while statement has the following form:

```
while expression do
  statement
  statement
  ...
endwhile
```

The while statement implements a loop which executes the statements between `do` and `endwhile` for as long as the expression value is not 0. As an example the following while statement will iterate for five times:

```
v = 1;
while v <= 5 do
  print v;           // print the value of v
  print newline;
  v = v + 1;        // increase v by 1
endwhile
```

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 “//”. An simple example Snail program is the following:

```
v = 10;
i = 0;
```

```

while i <= v do
  print i*i;           // print the square of i
  if ( i == v/2)      // is i the half of v?
    print newline;    //yes
  else
    print '--';       //no
  endif
  i = i + 1;
endwhile
print 'end of execution';

```

The output of the program is:

```

0--1--4--9--16--25
36--49--64--81--100
end of execution

```

3 Snail Grammar

All the Snail programs can be described by the the context-free grammar of Figure 1. The start variable is `program`, the grammar variables are in small letters, and the terminals in capital letters. Notice that although this grammar is ambiguous in the `expr` variable, all the ambiguities can be removed using the precedence rules of yacc.

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 from the snail grammar. You will add actions to the grammar so that your interpreter does the following for any input Snail program:

1. builds the derivation tree of the program, and then
2. “executes” the derivation tree.

The derivation tree (see Chapter 5 in Book) has a node for each variable and terminal. At the root of the tree is the variable `program`. An example Snail program and derivation tree is shown in Figure 2.

To build the derivation tree you need a special routine, e.g. `build_tree`, which you will invoke at each production of your grammar. Your nodes of your tree must be general enough to accomodate all the different kinds of productions, variables and terminals in the grammar. You need a mechanism to distinguish between the various kinds of nodes. (for example, you can have a variable `kind` inside each node).

By “executing the tree” we mean that we traverse the tree recursively from the root to the leaves and execute the code that corresponds to each node of

```

program -> stmt_list

stmt_list -> stmt_list stmt
           | stmt

stmt -> assign_stmt
      | print_stmt
      | if_stmt
      | while_stmt

assign_stmt -> ID = expr ;

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

if_stmt -> IF expr THEN stmt_list ENDIF
         | IF expr THEN stmt_list ELSE stmt_list ENDIF

while_stmt -> WHILE expr DO stmt_list ENDWHILE

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

```

Figure 1: The snail grammar

Program: print 10+5;

derivation tree

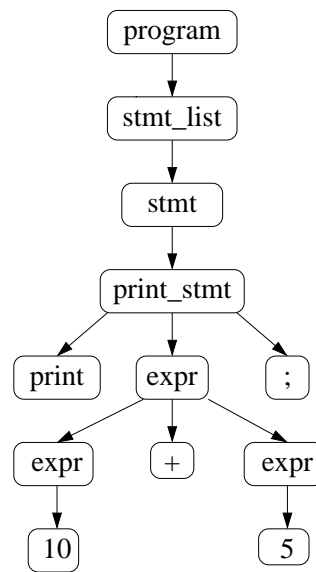


Figure 2: A small Snail program and its derivation tree

the tree. For this you will need to write a special routine, e.g. `execute_tree`. The main part of `execute_tree` is a big switch statement for the various kinds of nodes. The pseudocode for `execute_tree` is as follows:

```
execute_tree(tree) {  
  
    root = root(tree);  
    left_child  = root.left_child;  
    middle_child = root.middle_child  
    right_child = root.right_child;  
  
    switch (root.kind) {  
  
        case expr_plus:  execute_tree(left_child);  
                        execute_tree(right_child);  
                        root.value = left_child.value +  
                                right_child.value;  
  
        case expr_minus: .....  
        .....  
  
        case print_expr : execute_tree(middle_child);  
                        printf('%d', middle_child.value);  
  
        .....  
  
    }  
}
```

Each `expr` node must have a value variable which holds the current value of the expression. The `execute_tree` routine computes the `expr` values recursively, by computing the values of the children `expr` first. The value of the ID (identifier) can be stored in the symbol table.

For an `assign` node we update the value of the ID child in the symbol table to be equal to the value of the `expr` child.

For a `print` node we just print the contents (or value) of the middle child, which can be either a `STRING`, a `expr` or `NEWLINE`.

For an `if` node, we first execute the `expr` child and then if the value of `expr` is not 0 we execute the if-then `stmt_list` child. Otherwise, we execute the if-else `stmt_list`.

For a `while` node, we repeatedly do the following: first we execute the `expr` child and if the value of `expr` is not 0 we execute the `stmt_list` node. When the value of `expr` is zero the execution of the `while` node has finished.

Both routines `build_tree` and `execute_tree` will be invoked in the action part of the `program` variable.

Your yacc program will use the lexical analyzer of the first programming assignment and for this you need to modify appropriately the lex code.

5 HandIn

You should handin your lex and yacc code. Also you should handin the output for various Snail programs which will be given in the course web page.

In the course web page you can also find example yacc programs (together with lex programs) that may help you to get started with your assignment. For more information about yacc and lex visit the web page:

http://www.combo.org/lex_yacc_page/