CSE 340 - Fall 2012
Project 4

Assigned: November 18, 2012
Due: **December, 11, 2012**

Abstract

The goal of this project is to give you some hands-on experience with implementing a compiler. You will write a compiler for a simple language. You will not be generating low level code. Instead, you will generate an intermediate representation (a data structure that represents the program). The execution of the program will be done after compilation by *interpreting* the generated intermediate representation.

1 Introduction

You will write a compiler that will read an input program and represents it in an internal data structure. The data structure will contain instructions to be executed as well as a part that represents the memory of the program (space for variables). Then your compiler will execute the data structure. This means that the program will traverse the data structure and at every node it visits, it will execute the node by changing appropriate memory locations and deciding what is the next instruction to execute (program counter). The output of your compiler is the output that the input program should produce.

2 Grammar

The grammar for this project is a simplified form of the grammar from the previous project, but there are a couple extensions.

program	$\mapsto$	*var_section body*
id_list	$\mapsto$	ID COMMA *id_list*
	|	ID
var_section	$\mapsto$	VAR *var_decl*
var_decl	$\mapsto$	*id_list* SEMICOLON
body	$\mapsto$	LBRACE *stmt_list* RBRACE
stmt_list	$\mapsto$	*stmt stmt_list*
	|	stmt
stmt	$\mapsto$	ID = *expr* SEMICOLON
	|	*while_stmt*
	|	*if_stmt*
	|	*print_stmt*
expr	$\mapsto$	*primary op primary*
primary	$\mapsto$	ID
	|	NUM

op	$\mapsto$	PLUS | MINUS | MULT | DIV
print_stmt	$\mapsto$	**print** ID SEMICOLON
while_stmt	$\mapsto$	WHILE *condition body*
if_stmt	$\mapsto$	IF *condition body*
condition	$\mapsto$	*primary relop primary*
relop	$\mapsto$	GREATER | LESS | NOTEQUAL

Some highlights of the grammar:

1. Expressions are greatly simplified and are not recursive.
2. There is no type declaration section.
3. Division is integer division and the result of the division of two integers is an integer.
4. *if* statement are introduced. Note that *if_stmt* does not have *else*.
5. A print statement is introduced. Note that the **print** keyword is in lower case and not upper case.
6. There is no variable declaration list. There is only one *id_list* in the global scope and that contains all the variables.
7. There is no type specified for variables. All variables are INT by default.
8. All terminals are written in capital in the grammar and are as defined in the previous projects (except the **print** keyword)

3 Boolean Condition

A boolean condition takes two operands as parameters and returns a boolean value. It is used to control the execution of *while* statements and *if* statements.

4 Execution Semantics

All statements in a statement list are executed sequentially according to the order in which they appear. Exception is made for body of *while_stmt* as explained below.

5 while statement

while_stmt has the standard semantics:

1. The condition is evaluated.
2. If the condition evaluates to **true**, the body of the *while_stmt* is executed.

3. If the condition evaluates to **false**, the statement following the *while_stmt* in the *stmt_list* is executed

These semantics apply recursively to nested *while_stmt*.

6 if statement

if_stmt has the standard semantics:

1. The condition is evaluated.
2. If the condition evaluates to **true**, the body of the *if_stmt* is executed, then the next statement following the *if* is executed.
3. If the condition evaluates to **false**, the statement following the *if* in the *stmt_list* is executed

These semantics apply recursively to nested *if_stmt*.

7 Print

The statement

```
print a;
```

prints the value of variable a at the time of the execution of the print statement.

8 How to generate the code

The intermediate code will be a data structure that is easy to interpret (and execute). I will start by describing how this graph looks for simple assignments then I will explain how to deal with *while* statements.

Note that in the explanation below I start with incomplete data structures then I explain what is missing and make them more complete. You should read the whole explanation.

8.1 handling simple assignments

A simple assignment is fully determined by: the operator (if any), the id on the left-hand side, and the operand(s). A simple assignment can be represented as a node

```
struct assignmentNode {
  struct varNode* lvalue;
  struct varNode* op1;
  struct varNode* op2;
  int operator;
}
```

To execute an assignment, you can pass the assignment node to a function that will assign the value of right-hand side to the lvalue. For literals (NUM), the value is the value of the number. For variables, the value is the last value stored in the variable. **Initially, all variables are initialized to 0**.

Multiple assignments are executed one after another. So, we need to allow multiple nodes to be linked to each other. This can be achieved as follows:

```
struct assignmentNode {
  struct varNode* lvalue;
  struct varNode* op1;
  struct varNode* op2;
  int operator;
  struct assignmentNode* next;
}
```

This will now allow us to execute a sequence of assignment statements represented in a linked list: we start with the head of the list, then we execute every assignment in the list one after the other. This is simple enough, but does not help with executing other kinds of statements. We consider them one at a time.

8.2 handling print statements

The *print* statement is straightforward. It can be represented as

```
struct printStatement {
  struct varNode* id;
}
```

Now, we ask, how we can execute a sequence of statements that are either assignment or print statement (or other types of statements). We need to put both kinds of statements in a list and not just the assignment statements as we did above. So, we introduce a new kind of node, a statement node:

```
#define PRINTSTMT 0
#define ASSIGNSTMT 1

struct statementNode {
  int stmtType;                         // NOOPSTMT, GOTOSTMT, ASSIGNSTMT
                                        // WHILESTMT, IFSTMT
  struct assignmentNode* assign;
  struct printStatement* print;
  struct whileStatement* while;
  struct ifStatement* if;
  struct gotoSTatement goto;
  struct statementNode* next;
}
```

This way we can go through a list of statements and execute one after the other. To execute a particular node, we check its stmtType. If it is PRINTSTMT, we execute the print field, if it is ASSIGNSTMT, we execute the assign field and so on. With this modification, we do not need a next field in the assignmentNode structure.

This is all fine, but we do not yet know how to generate the list to execute later. The idea is to have function that parse non-terminals return the code for the non-terminals. For example for a statement list, we have the following pseudecode (missing many checks):

```
struct statementNode* stmt_list()
{ struct statementNode* st;   // statement
  struct statementNode* stl;  // statement list

  st = stmt();
  if (nextToken == start of a statement list)
  {
     stl = stmt_list();
     append stl to st;            // this is pseudecode
     return st;
  }
  else
  {  ungetToken();
     return st;
  }
}
```

8.3 If and While

More complications occur with *if* and *while* statements. The structure for an *if* statement can be as follows:

```
struct ifStatement {
   struct conditionNode* condition;     // condition of the if
   struct statementNode* stmt_list;     // body of the if statement
}
```

To generate the node for an *if* statement, we need to put together the *condition*, and *stmt_list* that are generated in the parsing of the *if* statement.

To generate code for a while statement, we can use the following representation:

```
struct whileStatement {
   struct conditionNode* condition;     // condition of the while
   struct statementNode* stmt_list;     // body of the while statement
}
```

Finally, we need to define the node for a condition:

```
struct conditionNode {
   int operator;
   struct varNode* op1;
   struct varNode* op2;
   struct statementNode* trueBranch;
   struct statementNode* falseBranch;
}
```

The *trueBranch* and *falseBranch* fields are crucial to the execution of the *while* and if statements. If the condition evaluates to true then the statement specified in *trueBranch* is executed otherwise the one specified in *falseBranch* is execute.

We need one more type of node to allow loop back for while statements. This is a gotoNode.

```
struct gotoNode {
  struct statementNode* target;
}
```

To generate code for the while statement and if statements, we need to put a few things together. The outline given above for *stmt_list* needs to be modified as follows (this is missing details and shows only the main steps):

```
struct statementNode* stmt_list()
{ struct statementNode* st;   // statement
  struct statementNode* stl;  // statement list

  st = stmt();
  if (nextToken == start of a statement list)
  {
     stl = stmt_list();

     if st is an IFSTMT
     {
           create no-op node                                   // this is a node that doe not result
                                                               // in any action being taken
           append no-op node to st
           fill the falseBranch of the condition
                 of st->if to point to the no-op node          //
                                                               //
           append no-op node to st->stmt_list                  // this will make the end of the body of the
                                                               // if point to the no-op node. This is the "fall-through"
                                                               // of the body of the if
                                                               // note that no-op is appended to two lists:
                                                               //                - ifstmt
                                                               //                - ifstmt->if->stmt_list


           append stl to st                                    // st is at the front


           //                             field
           //        st-------->if-node -----------> condition ----------
           //        |             |                    |               |
           //        |             |                    | true branch   |
           //        |             |    field           V               |
           //        |             |---------------> stmt_list          |
           //    next |                                 |               | false branch
           //        |  --------------------------------| next          |
           //        |  |                                               |
           //        V  V                                               |
           //       no-op  <------------------------------------------|
           //        |
           //        | next
           //        V
           //       stl

     } else
```

```
     if st is a WHILESTMT
     {
           create a new goto node gt and make the goto target be st   //
           append the goto node to st->stmt_list                      // append the go to the body of the while

           create  a no-op stmt and append the no-op stmt  to st      //
           fill the falseBranch of the condition of st->while         //
                  to point to no-op stmt                              //
           append stl to the list obtained by appending no-op to st   // at this point gt points to the
                                                                      // while statement
                                                                      // the falseBranch points to no-op
                                                                      // and the trueBranch points to the
                                                                      // body of the while


           //                                   field
           //          st   ----->while-node -----------> condition ----------
           //         |  /\              |                    |               |
           //         |   |              |                    | true branch   |
           //         |   |              |    field           V               |
           //         |   |              |---------------> stmt_list          |
           //    next |   |                                   |               | false branch
           //         |   |                                   | next          |
           //         |   |       target                      V               |
           //         |   -----------------------------------goto             |
           //         V                                                       |
           //       no-op <---------------------------------------------------|
           //         |
           //         | next
           //         V
           //        stl
     }
     append stl to st;            // st is at the front
     return st;
  }
  else
  {  ungetToken();
     return st;
  }
}
```

9 Executing the intermediate representation

After the graph data structure is built, it needs to be executed. Execution starts with the first node in the list. Depending on the type of the node, the next node to execute is determined. The general form for execution is illustrated in the following pseudo-code.

```
pc = first node
while (pc != NULL)
{
   case (node->stmtType)
     ASSIGNSTMT: // code to execute node->assign ...
                 pc = node->next

     IFSTMT:     // condition = node->if->condition
                 // code to evaluate condition ....
                 // depending on the result
                 //   pc = condition->trueBranch
```

```
                // or
                //   pc = condition->falseBranch

    NOOPSTMT:   pc = node->next

    GOTSTMT:    pc = node->goto->target

    WHILESTMT:  // ....
}
```

Executing the graph should be done non-recursively and without any function calls. This is a requirement that will be checked by inspecting your code.

10 Input/Output

The input will be read from standard input. We will test your programs by redirecting the standard input to an input file. You should not specify a file name from which to read the input. Output should be written to standard output.

11 Requirements

1. Write a compiler that generates intermediate representation for the code and write an interpreter to execute the intermediate representation. **You can assume that there are no syntax or semantic errors in the input program.**
2. **Platform:** You can use Java, C, or C++ for this assignment.
3. **Any language other than Java, C or C++ is not allowed for this project**.
4. You should execute and test any code you develop on the general machines. If you are used to programming in a Java IDE and you want to implement your solution using Java, you should familiarize with the `javac` compiler on general early on.
5. Do not wait until the last day to attempt porting to general. It is not straightforward and might take some time if you are using some features specific to your environment.

12 Bonus: replaces project 2 or project 3

Support the following grammar

$$
\begin{array}{lcl}
\textit{program} & \mapsto & \textit{var\_section body} \\
\textit{id\_list} & \mapsto & \text{ID COMMA } \textit{id\_list} \\
 & | & \text{ID} \\
\textit{var\_section} & \mapsto & \text{VAR } \textit{int\_var\_decl array\_var\_decl} \\
\textit{int\_var\_decl} & \mapsto & \textit{id\_list} \text{ SEMICOLON} \\
\textit{array\_var\_decl} & \mapsto & \textit{id\_list} \text{ COLON ARRAY[NUM] SEMICOLON} \\
\textit{body} & \mapsto & \text{LBRACE } \textit{stmt\_list} \text{ RBRACE} \\
\textit{stmt\_list} & \mapsto & \textit{stmt stmt\_list} \\
 & | & \text{stmt} \\
\textit{stmt} & \mapsto & \text{ID} = \textit{expr} \text{ SEMICOLON} \\
 & | & \textit{while\_stmt} \\
 & | & \textit{if\_stmt} \\
 & | & \textit{print\_stmt} \\
\textit{expr} & \mapsto & \textit{term} \text{ PLUS } \textit{expr} \\
\textit{expr} & \mapsto & \textit{term} \text{ MINUS } \textit{expr} \\
\textit{expr} & \mapsto & \textit{term} \\
\textit{term} & \mapsto & \textit{factor} \text{ MULT } \textit{term} \\
\textit{term} & \mapsto & \textit{factor} \text{ DIV } \textit{term} \\
\textit{term} & \mapsto & \textit{factor} \\
\textit{factor} & \mapsto & \text{LPAREN } \textit{expr} \text{ RPAREN} \\
\textit{factor} & \mapsto & \text{NUM} \\
\textit{factor} & \mapsto & \text{ID} \\
\textit{factor} & \mapsto & \text{ID}[\textit{expr}] \\
\textit{print\_stmt} & \mapsto & \textbf{print} \text{ ID SEMICOLON} \\
\textit{while\_stmt} & \mapsto & \text{WHILE } \textit{condition body} \\
\textit{if\_stmt} & \mapsto & \text{IF } \textit{condition body} \\
\textit{condition} & \mapsto & \textit{primary relop primary} \\
\textit{relop} & \mapsto & \text{GREATER} \mid \text{LESS} \mid \text{NOTEQUAL}
\end{array}
$$

Assume that all arrays are integer arrays.

Submission

1. **You should submit all your code on blackboard by 11:59:59 pm on December, 11, 2012**.
2. You should submit one **.zip** file that includes all your source files and provide instructions on how to compile and execute your code.
3. Make sure your submission is correctly uploaded.
4. Make sure you follow the naming conventions for your files.