CSE 340 Fall 2014 – Project 4

Due on **Dec. 5, 2014 by 11:59 pm**

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	→	*var_section* *body*
var_section	→	*id_list* SEMICOLON
id_list	→	ID COMMA *id_list* | ID
body	→	LBRACE *stmt_list* RBRACE
stmt_list	→	*stmt* *stmt_list* | stmt
stmt	→	*assign_stmt* | *print_stmt* | *while_stmt* | *if_stmt* | *switch_stmt*
assign_stmt	→	ID EQUAL *primary* SEMICOLON
assign_stmt	→	ID EQUAL *expr* SEMICOLON
expr	→	*primary* *op* *primary*
primary	→	ID | NUM
op	→	PLUS | MINUS | MULT | DIV
print_stmt	→	**print** ID SEMICOLON
while_stmt	→	WHILE *condition* *body*
if_stmt	→	IF *condition* *body*
condition	→	*primary* *relop* *primary*
relop	→	GREATER | LESS | NOTEQUAL

switch_stmt	→	SWITCH ID LBRACE *case_list* RBRACE
switch_stmt	→	SWITCH ID LBRACE *case_list* *default_case* RBRACE
case_list	→	*case* *case_list* | *case*
case	→	CASE NUM COLON *body*
default_case	→	DEFAULT COLON *body*

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 is introduced. Note that *if_stmt* does not have *else*. Also, there is no SEMICOLON after the *if* statement.
5. *switch* statement is introduced. Note that there is no SEMICOLON after the *switch* statement.
6. A *print* statement is introduced. Note that the **print** keyword is in lower case.
7. There is no variable declaration list. There is only one *id_list* in the global scope and that contains all the variables.
8. There is no type specified for variables. All variables are INT by default.
9. 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* 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 *if_stmt*, *while_stmt* and *switch_stmt* as explained below.

4.1 *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*.

4.2 *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, then the condition is evaluated again and the process repeats.
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*. The code block:

```
WHILE condition
{
    stmt_list
}
```

is equivalent to:

```
label: IF condition
    {
        stmt_list
        goto label
    }
```

Note that **goto** statements do not appear in the input program, but our intermediate representation includes `gotoStatement` which is used in conjunction with `ifStatement` to represent *while* and *switch* statements.

4.3 *Switch* statement

switch_stmt has the standard semantics:

1. The value of the switch variable is checked against each case number in order.
2. If the value matches the number, the body of the case is executed, then the statement following the *switch_stmt* in the *stmt_list* is executed.
3. If the value does not match the number, next case is evaluated.
4. If a default case is provided and the value does not match any of the case numbers, then the body of the default case is executed and then the statement following the *switch_stmt* in the *stmt_list* is executed.
5. If there is no default case and the value does not match any of the case numbers, then the statement following the *switch_stmt* in the *stmt_list* is executed.

These semantics apply recursively to nested *switch_stmt*. The code block:

```
    SWITCH var {
        CASE n_1 : { stmt_list_1 }
        ...
        CASE n_k : { stmt_list_k }
    }
```

is equivalent to:

```
    IF var == n_1 {
        stmt_list_1
        goto label
    }
    ...
    IF var == n_k {
        stmt_list_k
        goto label
    }
label:
```

And for switch statements with default case, the code block:

```
    SWITCH var {
        CASE n_1 : { stmt_list_1 }
        ...
        CASE n_k : { stmt_list_k }
        DEFAULT : { stmt_list_default }
    }
```

is equivalent to:

```
    IF var == n_1 {
        stmt_list_1
        goto label
    }
    ...
    IF var == n_k {
        stmt_list_k
        goto label
    }
    stmt_list_default
label:
```

5 *Print* statement

The statement

```
print a;
```

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

6 How to generate the code

The intermediate code will be a data structure (a graph) 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.

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

For assignment without an expression on the right-hand side, the operator is set to 0 and there is only one operand. To execute an assignment, you need the values of the operand(s), apply the operator, if any, to the operands and assign the resulting value of the 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 assignment nodes to be linked to each other. This can be achieved as follows:

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

This structure only accepts `varNode` as operands. To handle literal constants (`NUM`), you need to store them in `varNode` while parsing.

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.

6.2 Handling *print* statements

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

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

Now, we ask: how can we 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. The statement node has a field that indicates which type of statement it is. It also has fields to accommodate the remaining types of statements. It looks like this:

```
struct statementNode {
    int stmtType;      // NOOPSTMT, GOTOSTMT, ASSIGNSTMT, IFSTMT, PRINTSTMT

    struct assignmentStatement* assign_stmt;
    struct printStatement* print_stmt;
    struct ifStatement* if_stmt;
    struct gotoStatement* goto_stmt;

    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_stmt` field, if it is `ASSIGNSTMT`, we execute the `assign_stmt` field and so on. With this modification, we do not need a next field in the `assignmentStatement` structure.

This is all fine, but we do not yet know how to generate the list to execute later. The idea is to have the functions 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* parse_stmt_list()
{
    struct statementNode* st;   // statement
    struct statementNode* stl;  // statement list

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

And to parse *body* we have the following pseudecode:

```
struct statementNode* parse_body()
{
    struct statementNode* stl;

    match LBRACE
    stl = parse_stmt_list();
    match RBRACE

    return stl;
}
```

6.3 Handling *if* and *while* statements

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

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

The `operator`, `op1` and `op2` fields are the operator and operands of the condition 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.

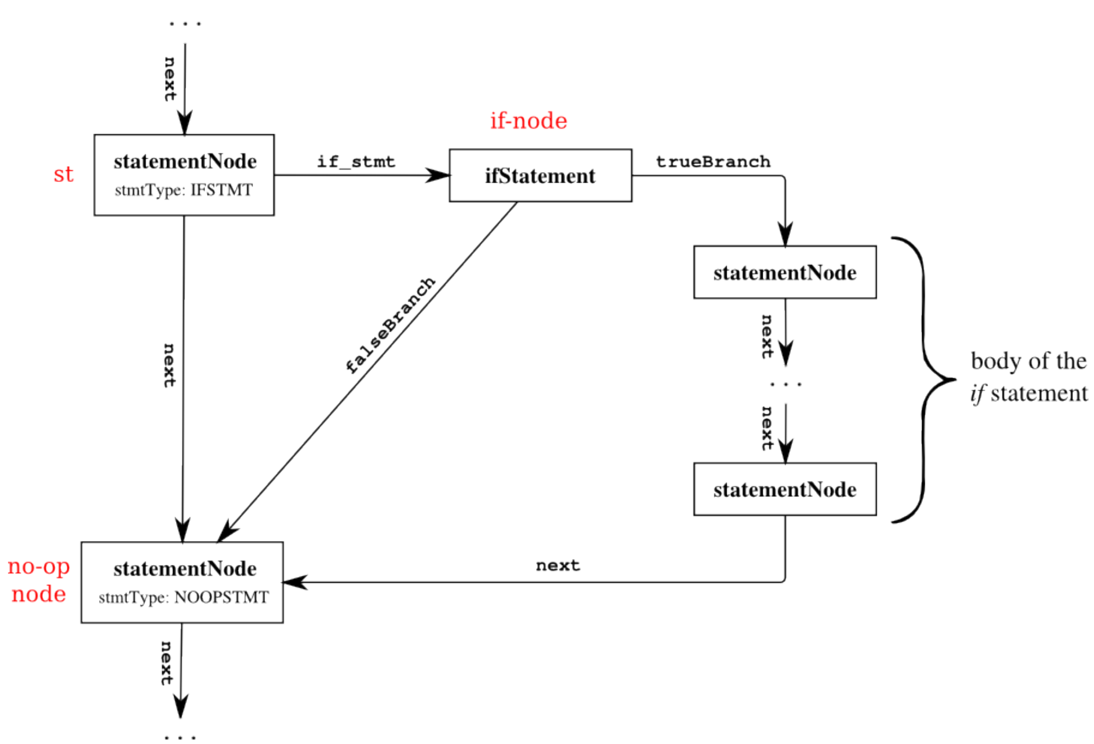
The `trueBranch` and `falseBranch` fields are crucial to the execution of the *if* statement. If the condition evaluates to true then the statement specified in `trueBranch` is executed otherwise the one specified in `falseBranch` is executed. We need one more type of node to allow loop back for *while* statements. This is a `gotoStatement`.

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

To generate code for the *while* 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* parse_stmt()
{
    ...

    create statement node st
    if next token is IF
    {
        st->stmtType = IFSTMT;
        create if-node;                               // note that if-node is pseudecode and is not
                                                      // a valid identifier in C, C++ or Java
        st->if_stmt = if-node;

        parse the condition and set if-node->operator, if-node->op1 and if-node->op2

        if-node->trueBranch = parse_body();           // parse_body returns a pointer to a list of statements

        create no-op node                             // this is a node that does not result
                                                      // in any action being taken

        append no-op node to the body of the if       // this requires a loop to get to the end of
                                                      // if-node->trueBranch by following the next field
                                                      // you know you reached the end when next is NULL
                                                      // it is very important that you always appropriately
                                                      // initialize fields of any data structures
                                                      // do not use uninitialized pointers
        set if-node->falseBranch to point to no-op node
        set st->next to point to no-op node

        ...

    } else ...
}
```

The following diagram shows the desired structure for the *if* statement:



The *stmt_list* code should be modified because of the extra no-op node:

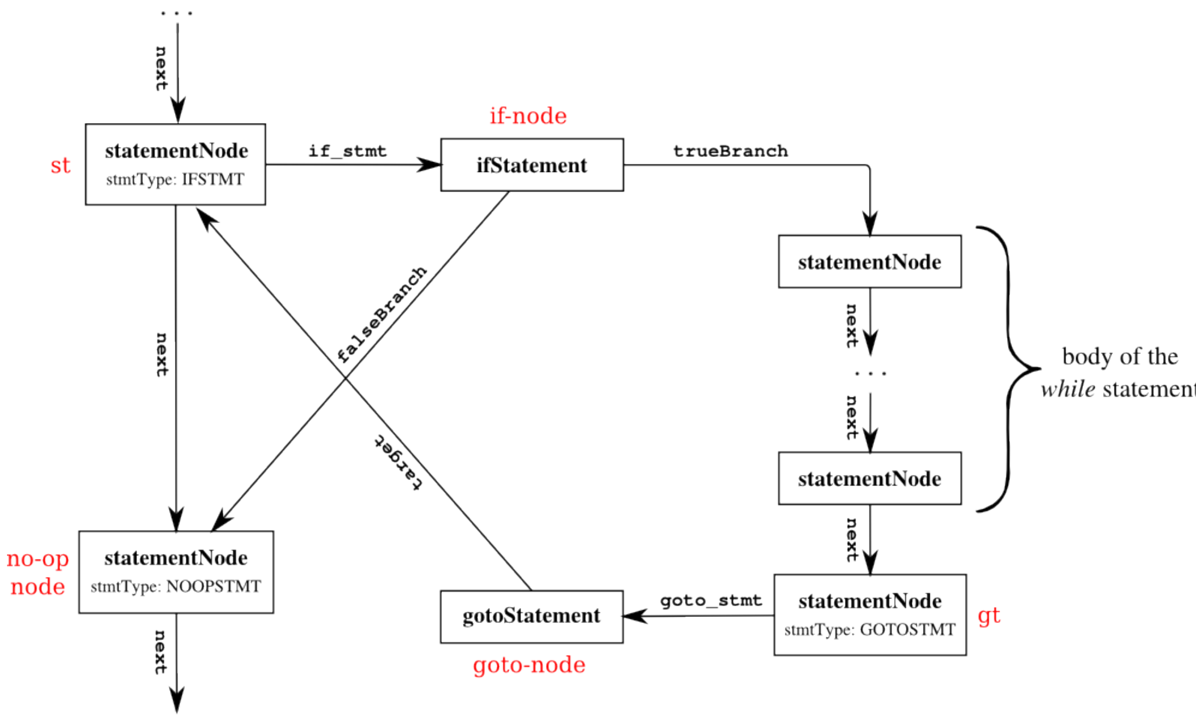
```
struct statementNode* parse_stmt_list()
{
    struct statementNode* st;   // statement
    struct statementNode* stl;  // statement list

    st = parse_stmt();
    if (nextToken == start of a statement list)
    {
        stl = parse_stmt_list();

        if st->stmtType == IFSTMT
        {
            append stl to the no-op node that follows st

            //         st
            //         |
            //         V
            //       no-op
            //         |
            //         V
            //        stl
        }
        else
        {
            append stl to st;

            //         st
            //         |
            //         V
            //        stl
        }
        return st;
    }
    else
    {
        ungetToken();
        return st;
    }
}
```

Handling *while* statement is similar. Here is the outline for parsing a *while* statement and creating the data structure for it:

```
...

create statement node st
if next token is WHILE
{
    st->stmtType = IFSTMT;                          // handling WHILE using if and goto nodes
    create if-node                                  // if-node is not a valid identifier see
                                                    // corresponding comment above
    st->if_stmt = if-node

    parse the condition and set if-node->operator, if-node->op1 and if-node->op2

    if-node->trueBranch = parse_body();

    create a new statement node gt                  // This is of type statementNode
    gt->stmtType = GOTOSTMT;
    create goto-node                                // This is of type gotoStatement
    gt->goto_stmt = goto-node;
    goto-node->target = st;                         // to jump to the if statement after
                                                    // executing the body
```

```
        append gt to the body of the while              // append gt to the body of the while
                                                        // this requires a loop. check the comment
                                                        // for the if above.

        create no-op node
        set if-node->falseBranch to point to no-op node
        set st->next to point to no-op node
    }

    ...
```

The following diagram shows the desired structure for the *while* statement:



6.4 Handling *switch* statement

You can handle the *switch* statement similarly. Use a combination of `ifStatement` and `gotoStatement` to support the semantics of the *switch* statement. See section 4.3 for more information.

7 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)
{
    switch (pc->stmtType)
    {
       case ASSIGNSTMT: // code to execute pc->assign_stmt ...
                        pc = pc->next

       case IFSTMT:     // code to evaluate condition ...
                        // depending on the result
                        //   pc = pc->if_stmt->trueBranch
                        // or
                        //   pc = pc->if_stmt->falseBranch

       case NOOPSTMT:   pc = pc->next

       case GOTSTMT:    pc = pc->goto_stmt->target

       case PRINTSTMT:  // code to execute pc->print_stmt ...
                        pc = pc->next
    }
}
```

Executing the graph should be done non-recursively and without any function calls. Even helper functions are not allowed for the execution of the graph. This is a requirement that will be checked by inspecting your code. Little credit will be assigned if this requirement is not met.

The data structures and the execution function are provided. If you are developing in C or C++, we have provided you with the code to execute the graph and you should use it. There are two files `compiler.h` and `compiler.c`, you need to write your code in separate file(s) and include `compiler.h`. The entry point of your code is a function declared in `compiler.h`:

```
struct statementNode *  parse_program_and_generate_intermediate_representation();
```

You need to implement this function.
The `main()` function is given in `compiler.c`:

```
int main()
{
  struct statementNode * program = parse_program_and_generate_intermediate_representation();
  execute_program(program);
  return 0;
}
```

It calls the function that you will implement which is supposed to parse the program and generate the intermediate representation, then it calls the `execute_program` function to execute the program. You should not modify any of the given code. In fact if you write your program in C or C++, you should only submit the file(s) that contain your own code and we will add the given part and compile the code before testing. If you write your program in Java, you should strictly follow the guidelines for executing the intermediate representation.

8 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.

9 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. **Language:** 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. If you use C or C++ for this project, you should use the provided code and only implement the required functions.
5. If you use Java, you will need to write everything yourself but the requirements on the execute function will be checked manually when grading.
6. **Platform:** As previous projects, the reference platform is CentOS 6.5

10 Submission

1. Submit your code on the course website by the deadline. Submission by email or other forms are NOT accepted.
2. You should submit the bonus separately from the main submission.
3. As always, input is from standard input and output is to standard output.
4. **If you use C/C++** then only submit *your own code*. **Do NOT** submit `compiler.h` and `compiler.c`. These files are automatically added to your submission (this does not apply to the bonus project or Java submissions).

11 Grading

The test cases provided with the assignment as well as those posted on the course website, do not contain any test case for *switch* statement. However, test cases with *switch* statements will be added for grading the project. The additional test cases will account for 10% of the assignment grade. Make sure you test your code extensively with input programs that contain switch statements.

12 Bonus: replaces any project grade for projects 1 or 2

Support the following grammar:

program	→	*var_section body*
var_section	→	VAR *int_var_decl array_var_decl*
int_var_decl	→	*id_list* SEMICOLON
array_var_decl	→	*id_list* COLON ARRAY LBRAC NUM RBRAC SEMICOLON
id_list	→	ID COMMA *id_list* | ID
body	→	LBRACE *stmt_list* RBRACE
stmt_list	→	*stmt stmt_list* | stmt
stmt	→	*assign_stmt* | *print_stmt* | *while_stmt* | *if_stmt* | *switch_stmt*
assign_stmt	→	*var_access* EQUAL *expr* SEMICOLON
var_access	→	ID | ID LBRAC *expr* RBRAC
expr	→	*term* PLUS *expr*
expr	→	*term*
term	→	*factor* MULT *term*
term	→	*factor*
factor	→	LPAREN *expr* RPAREN
factor	→	NUM
factor	→	*var_access*
print_stmt	→	**print** *var_access* SEMICOLON
while_stmt	→	WHILE *condition body*
if_stmt	→	IF *condition body*
condition	→	*expr relop expr*
relop	→	GREATER | LESS | NOTEQUAL
switch_stmt	→	SWITCH *var_access* LBRACE *case_list* RBRACE
switch_stmt	→	SWITCH *var_access* LBRACE *case_list default_case* RBRACE
case_list	→	*case case_list* | *case*
case	→	CASE NUM COLON *body*
default_case	→	DEFAULT COLON *body*

Note that LBRAC is `"["` and LBRACE is `"{"`. The former is used for arrays and the latter is used for body. Assume that all arrays are integer arrays and are indexed from 0 to *size* - 1, where *size* is the size of the array specified in the *var_section* after the ARRAY keyword and between `"["` and `"]"`.

The data structures and code that we have provided for the regular assignment will not be enough for the bonus, you will need to modify those to support arrays. Submit *all* code files for the bonus project (including the modified `compiler.h` and `compiler.c`).

All restrictions imposed on the execution of the intermediate representation for the regular project apply to the bonus project as well. You are not allowed to call any functions while executing the intermediate representation. You are not allowed to execute the program recursively.