CSE 340 Fall 2019 – Project 4

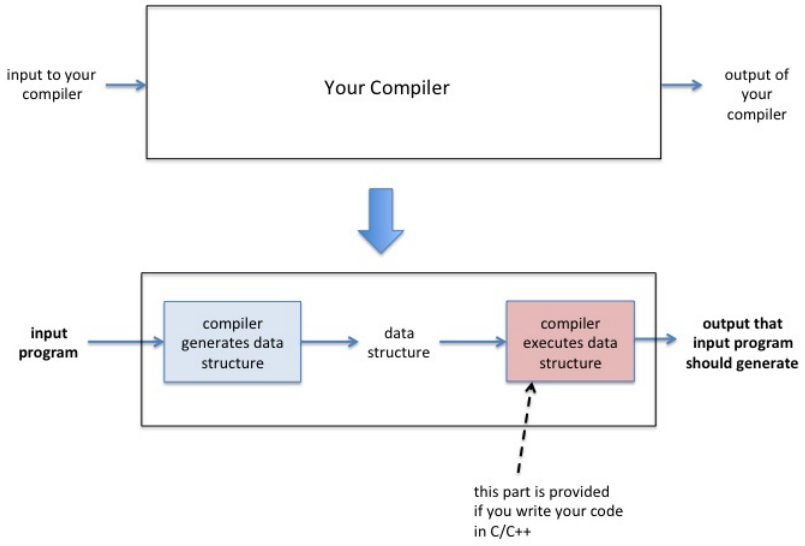
Due on **December 4 2019 by 11:59 pm**

Abstract

The goal of this project is to give you some hands-on experience with implementing a small compiler. You will write a compiler for a simple language. You will not be generating assembly 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 small compiler that will read an input program and represent it in an internal data structure. The data structure consists of two parts: (1) a representation of instructions to be executed and (2) a representation of the memory of the program (locations for variables). Instructions are represented by a data structure that includes the operand(s) of the instruction (if any) and that specify the next instruction to be executed. After the data structures are generated by your compiler, your compiler will *execute* the generated instructions representation by interpreting it. This means that the program will traverse the data structure and at every node it visits, it will "execute" the node by changing the content of memory locations corresponding to operands 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. These steps are illustrated in the following figure



The remainder of this document is organized into the following sections:

1. **Grammar** Defines the programming language syntax including grammar.
2. **Execution Semantics** Describe statement semantics for `if`, `while`, `switch` and `print` statements.
3. **How to generate the intermediate representation** Explains step by step how to generate the intermediate representation (data structure). **You should read this sequentially and not skip around.**
4. **Executing the intermediate representation** This section specifies that you should use the code we provide to execute the intermediate representation.
5. **Requirements** Lists other requirements.
6. **Grading** Describes the grading scheme.
7. **Second Chance Project** Describes the requirements for a second-chance project.

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* | *for_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
for_stmt	→	FOR LPAREN assign_stmt condition SEMICOLON assign_stmt RPAREN *body*
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*.
5. *for* statement is introduced. Note that *for* has a very general syntax similar to that of the *for loop* in the C language
6. A *print* statement is introduced. Note that the **print** keyword is in lower case, but other keywords are all upper-case letters.
7. *condition* has no parentheses.
8. There is no variable declaration list. There is only one *id_list* in the global scope and that contains all the variables.
9. There is no type specified for variables. All variables are `int` by default.

3 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. In what follows, I will assume that all values of variables as well as constants are stored in locations. This assumption is used by the execution procedure that we provide. This is not a restrictive assumption. For variables, you will have locations associated with them. For constants, you can reserve a location in which you store the constant (this is like having an unnamed immutable variable).

3.1 Assignment Statement

To execute an assignment statement, the expression on the righthand side of the equal sign is evaluated and the result is stored in the location associated with the lefthand side of the expression.

3.2 Expression

To evaluate an expression, the values in the locations associated with the two operands are obtained and the expression operator is applied to these values resulting in a value for the expression.

3.3 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. To evaluate a condition, the values in the locations associated with the operands are obtained and the relational operator is applied to these values resulting in a true or false value. For example, if the values of the two operands a and b are 3 and 4 respectively, `a < b` evaluates to **true**.

3.4 *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 (if any) following the *if_stmt* in the *stmt_list* is executed.
3. If the condition evaluates to **false**, the statement following the *if_stmt* in the *stmt_list* is executed

3.5 *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. The next statement to execute is the *while_stmt* itself.
3. If the condition evaluates to **false**, the body of the *while_stmt* is not executed. The next statement to execute is the next statement (if any) following the *while_stmt* in the *stmt_list*.

The code block:

```
WHILE condition
{
    stmt_list
}
```

is equivalent to:

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

Note that **goto** statements are not part of the grammar and cannot appear in a program (input to your compiler), but our intermediate representation includes `GotoStatement` which is used in conjunction with `IfStatement` to represent *while* and *switch* statements.

3.6 *For* statement

The *for_stmt* is very similar to the for statement in the C language. The semantics are defined by giving an equivalent construct.

```
FOR ( assign_stmt_1 condition ; assign_stmt_2 )
{
    stmt_list
}
```

is equivalent to:

```
assign_stmt_1
WHILE condition
{
    stmt_list
    assign_stmt_2
}
```

For example, the following snippet of code:

```
FOR ( a = 0; a < 10; a = a + 1; )
{
    print a;
}
```

is equivalent to:

```
a = 0;
WHILE a < 10
{
    print a;
    a = a + 1;
}
```

3.7 *Switch* statement

switch_stmt has the following semantics[1]:

[1]The switch statement in the C language has different syntax and semantics. It is also dangerous!

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, the next case number is checked.
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.

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:
```

3.8 *Print* statement

The statement

```
print a;
```

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

4 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.

4.1 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 ValueNode* left_hand_side;
    struct ValueNode* operand1;
    struct ValueNode* operand2;
    ArithmeticOperatorType op; // operator
}
```

For assignment without an operator on the right-hand side, the operator is set to `OPERATOR_NONE` and there is only one operand. To execute an assignment, you need calculate the value of the right-hand-side and assign it to the left-hand-side. If there is an operator, the value of the right-hand-side is calculated by applying the operator to the values of the operands. If there is no operator, the value of the right-hand-side is the value of the single operand: for literals (NUM), the value is the value of the number; for variables, the value is the last value stored in the location associated with

the variable. **Initially, all variables are initialized to 0**. In this representation, the locations associated with variables as well as the locations in which constants are stored are value nodes (`struct valueNode`), a node that contains a value. The assignment statement contains pointers to the value nodes of the left-hand side and the operand(s).

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 ValueNode* left_hand_side;
    struct ValueNode* operand1;
    struct ValueNode* operand2;
    ArithmeticOperatorType op;  // operator
    struct AssignmentStatement* next;
}
```

This structure only accepts `ValueNode` as operands. To handle literal constants (`NUM`), you need to create `ValueNode` for them and initialize the value stored in these value nodes to the constant value. This initialization, as well as the initialization of values in locations associated with variable is done 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.

Begin Note It is important to distinguish between compile-time initialization and runtime execution. For example, consider the program

```
a b;
{
    a = 3;
    b = 5;
}
```

The intermediate representation for this program will have have two assignment statements: one to copy the value in the location that contains the value 3 to the location associated with a and one to copy the value in the location that contains the value 5 to the location associated with b. The values 3 and 5 will not be copied to the locations of a and b at compile-time. The values 3 and 5 will be copied during execution by the interpreter that we provided. I highly recommend that you read the code of the interpreter that we provided. In particular, there is a sample hardcoded example to show how the generated code will look like.**End Note**

This is simple enough, but does not help with executing other kinds of statements. We consider them one at a time.

4.2 Handling *print* statements

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

```
struct PrintStatement
{
```

```
    struct ValueNode* 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 {
    StatementType type;  // NOOP_STMT, GOTO_STMT, ASSIGN_STMT, IF_STMT, PRINT_STMT

    union {
        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 `type`. If the type is `PRINT_STMT`, we execute the `print_stmt` field, if the type is `ASSIGN_STMT`, we execute the `assign_stmt` field and so on. With this modification, we do not need a `next` field in the `AssignmentStatement` structure (as we explained above), instead, we put the `next` field in the statement node.

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 parses non-terminals return the code that corresponds to 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;
}
```

4.3 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 {
    ConditionalOperatorType condition_op;
    struct ValueNode* condition_operand1;
    struct ValueNode* condition_operand2;

    struct StatementNode* true_branch;
    struct StatementNode* false_branch;
}
```

The `condition_op`, `condition_operand1` and `condition_operand2` 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.

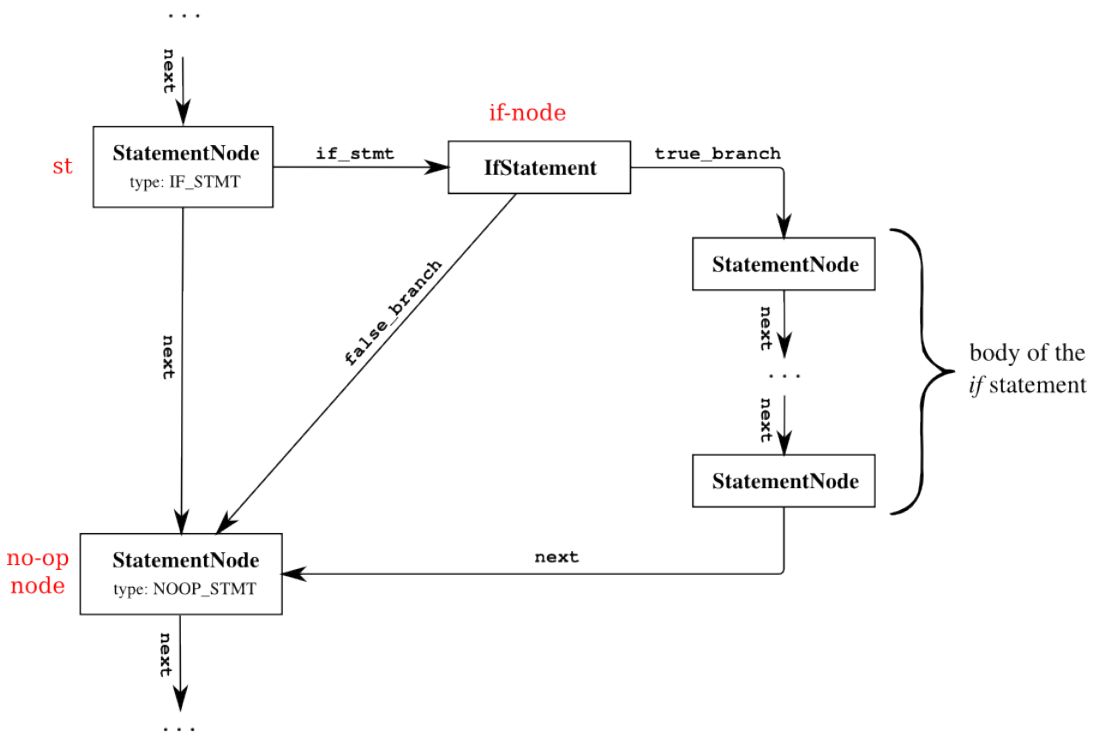
The `true_branch` and `false_branch` fields are crucial to the execution of the *if* statement. If the condition evaluates to true then the statement specified in `true_branch` is executed otherwise the one specified in `false_branch` is executed. We need one more type of node to allow loop back for *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->type = IF_STMT;
        create if-node;                                  // note that if-node is pseudecode and is not
                                                         // a valid identifier in C, C++
        st->if_stmt = if-node;

        parse the condition and set if-node->condition_op, if-node->condition_operand1 and if-node->condition_operand2

        if-node->true_branch = 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->true_branch 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->false_branch 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:

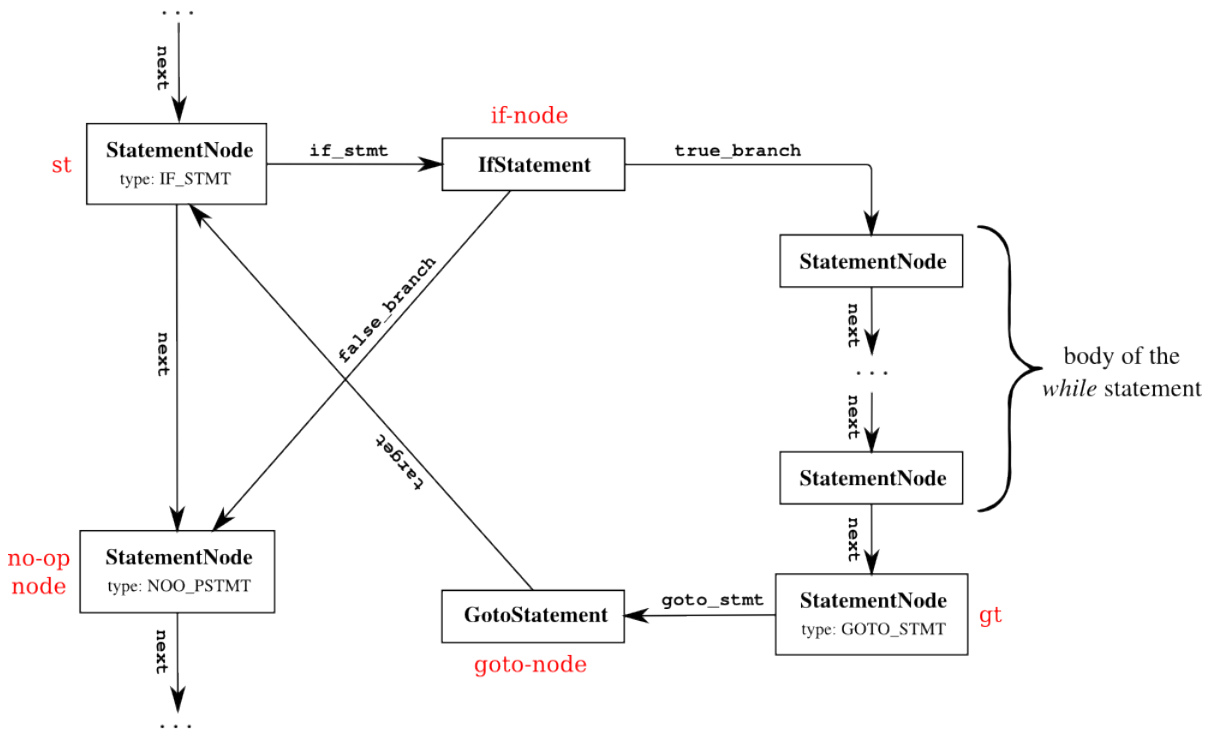
```
struct StatementNode* parse_stmt_list()
{
    struct StatementNode* st;   // statement
    struct StatementNode* stl;  // statement list

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

        if st->type == IF_STMT
        {
            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->type = IF_STMT;                              // 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->condition_op, if-node->condition_operand1 and if-node->condition_operand2

    if-node->true_branch = parse_body();

    create a new statement node gt                   // This is of type StatementNode
    gt->type = GOTO_STMT;
    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->false_branch 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:



4.4 Handling *switch* and *for* statements

You can handle the *switch* and *for* statements similarly, but you should figure that yourself. Use a combination of `IfStatement` and `GotoStatement` to support the semantics of the *switch* and *for* statements. See sections 3.7 and 3.6 for the semantics of the *switch* and *for* statements.

5 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->type)
    {
       case ASSIGN_STMT: // code to execute pc->assign_stmt ...
                         pc = pc->next

       case IF_STMT:     // code to evaluate condition ...
                         // depending on the result
                         //   pc = pc->if_stmt->true_branch
                         // or
                         //   pc = pc->if_stmt->false_branch

       case NOOP_STMT:   pc = pc->next

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

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

We have provided you with the data structures and the code to execute the graph and **you must use it**. There are two files `compiler.h` and `compiler.cc`, 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_generate_intermediate_representation();
```

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

```
int main()
{
    struct StatementNode * program;
    program = parse_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, you should not submit compiler.cc and compiler.h; we will provide them when you submit your code.

6 Requirements

1. Write a compiler that generates intermediate representation for the code and write an interpreter to execute the intermediate representation. The interpreter is provided.

2. **Language:** You can only use C, or C++ for this assignment.

3. **You can assume that there are no syntax or semantic errors in the input program.**

7 Grading

The test cases provided with the assignment, do not contain any test case for *switch* and *for* statements. However, test cases with *switch* and *for* statements will be added for grading the project. Make sure you test your code extensively with input programs that contain *switch* and *for* statements.

The test cases (there will be multiple test cases in each category, each with equal weight) will be broken down in the following way (out of 100 points):

- Assignment statements: 20
- If statements: 20
- While statements: 30
- Switch: 15
- All statements not including For statement: 10
- For statement: 5
- **Total:** 100

Note that all test cases depend on successful implementation of print statements (otherwise your program cannot generate correct output). Also, assignment statements are needed for *if*, *while*, *switch*, statement test cases all depend on assignment statements.

8 Second Chance Project

For this option you are given another chance to submit project 2. The grade you obtain will be reduced by 35% and, if it is greater than the grade you previously obtained on project 2, it will replace it.