CSE 340 Fall 2020 – Project 3

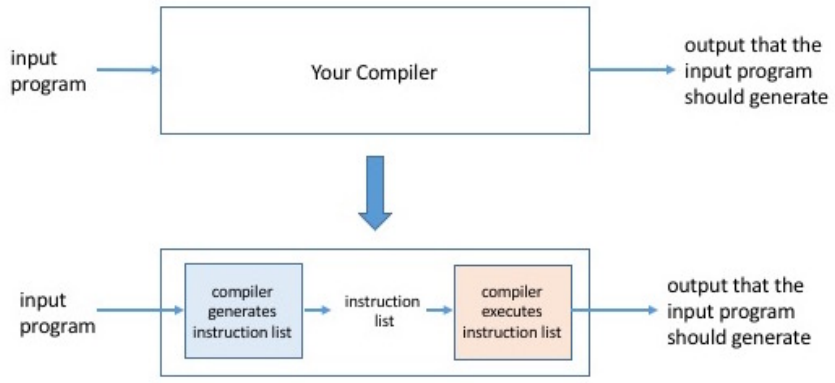
Due on **November 9 2020 by 11:59 pm**

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 a linked list. A node of the linked list represents one instruction. An instruction node specifies: (1) the type of the instructions, (2) the operand(s) of the instruction (if any) and, for jump instructions, the next instruction to be executed (the default is that the next instruction in the list is executed). After the list of instructions is generated by your compiler, your compiler will *execute* the generated list of instructions 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 `assignment`, `input`, `if`, `while`, `switch`, *for* and `output` statements.

3. **How to generate the linked list of instructions** Explains how to generate the intermediate representation (data structure). **You should read this sequentially and not skip around.**

4. **Requirements** Lists other requirements.

5. **Grading** Describes the grading scheme.

2 Grammar

The grammar for this project is the following:

program	→	*var_section body inputs*
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* | *while_stmt* | *if_stmt* | *switch_stmt* | *for_stmt*
stmt	→	*output_stmt* | *input_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
output_stmt	→	**output** ID SEMICOLON
input_stmt	→	**input** 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*
inputs	→	*num_list*
num_list	→	`NUM`
num_list	→	`NUM` *num_list*

Some highlights of the grammar:

1. Division is integer division and the result of the division of two integers is an integer.

2. Note that *if_stmt* does not have *else*.

3. Note that *for* has a very general syntax similar to that of the *for loop* in the C language

4. Note that the **input** and **output** keywords are lowercase, but other keywords are all uppercase.

5. *condition* has no parentheses.

6. There is no type specified for variables. All variables are `int` by default.

3 Variables and Locations

The *var_section* contains a list of all variable names that can be used by the program. For each variable name, we associate a unique locations that will hold the value of the variable. This association between a variable name and its location is assumed to be implemented with a function `location` that takes a variable name (`string`) as input and returns an integer value. The locations where variables will be stored is called `mem` which is an array of integers. Each variable in the program should have a unique entry (index) in the `mem` array. This association between variable names and locations can be implemented with a location table.

As your parser parses the input program, it *allocates* locations to variables that are listed in the *var_section*. You can assume that all variable names listed in the var section are unique. For each variable name, a new location needs to be associated with it and the mapping from the variable name to the location needs to be added to the location table. To associate a location with a variable, you can simply keep a counter that tells you how many locations have been used (associated to variable names). Initially the counter is 0. The first variable to be associated a location will get the location whose index is 0 (`mem`[0]) and the counter will be incremented to become 1. The next variable to be associated a location will get the location whose index is 1 and the counter will be incremented to become 2 and so on.

4 Inputs

The list of input values is called *inputs* and appears as the last section of the input to your compiler. This list must be read by your compiler and stored in an `inputs` array, which is simply a vector of integers.

5 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*, *switch_stmt*, and *for_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).

5.0.1 Input statements

Input statements get their input from the sequence of `inputs`. We refer to i'th value that appears in `inputs` as `i'th input`. The execution of the i'th input statement in the program of the form `input a` is equivalent to:

```
mem[location("a")] = inputs[input_index]
input_index = input_index + 1
```

where `location("a")` is an integer index value that is calculated at compile time as we have seen above. Note that the execution of an input statement advances an `input_index` which keeps track (at runtime) of the next value to read (like in project 1).

5.1 *Output* statement

The statement

```
output a;
```

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

5.2 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.

5.3 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.

5.4 Boolean Condition

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

5.5 *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.

5.6 *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
    }
```

Jump: In the code above, a `goto` statement is similar to the goto statement in the C language. 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 `jump` which is used in the implementation of *if*, *while*, *for*, and *switch* statements (`jump` is discussed later in this document).

5.7 *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; )
{
    output a;
}
```

is equivalent to:

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

5.8 *Switch* statement

switch_stmt has the following 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, 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:
```

The provided intermediate representation does not have a test for equality. You are supposed to implement the switch statement with the provided intermediate representation.

Note that the switch statement in the C language has different syntax and semantics. It is also dangerous!

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 AssignmentInstruction {
    int  left_hand_side_index;
    int  operand1_index;
    int  operand2_index;
    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 in the `mem[]` array mentioned above. In the statement, the index (address) of the location where the value of the variable or the constant is stored is given. The actual values in `mem[]` can be fetched or modified (for variables) at runtime.

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 AssignmentInstruction {
    int  left_hand_side_index;
    int  operand1_index;
    int  operand2_index;
    ArithmeticOperatorType op; // operator
    struct AssignmentStatement* next;
}
```

This structure only accepts indices (addresses) as operands. To handle literal constants (`NUM`), you need to store their values in `mem[]` at compile time and use the index of the constant as the operant.

This data structure will now allow us to execute a sequence of assignment statements represented in a linked-list of assignment instructions: 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;
}
1 2 3 4
```

The intermediate representation for this program will have have two assignment instructions: 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 (also, your program should read the inputs and store them in the inputs vector, but this is not the point of this example). 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 as well as the code in demo.cc. In demo.cc, a hardcoded data structure is shown for an example input program, which can be very useful in understanding what the data structure your program will generate 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.

6.2 Handling *output* statements

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

```
struct OutputInstruction
{
    int var_index;
}
```

where the operand is the index of the location of the variable to be printed.

Now, we ask: how can we execute a sequence of statements that are either *assign* or *output* statement (or other types of statements)? We need to put the instructions for both kinds of statements in a list. So, we introduce a new kind of node: an instruction node. The instruction node has a field that indicates which type of instruction it is. It also has fields to accommodate instructions for the remaining types of statements. It looks like this:

```
struct InstructionNode {
    InstructionType type;  // NOOP, ASSIGN, JMP, CJMP (conditional jump), IN, OUT

    union {
        struct {
                 int  left_hand_side_index;
                 int  operand1_index;
                 int  operand2_index;
                 ArithmeticOperatorType op;
        } assign_inst;
        struct {
               // details below
        } jmp_inst;
        struct  {
               // details below
        } cjmp_inst;
        struct {
               int var_index;
        } input_inst ;
        struct  {
               int var_index;
        } output_inst;
    };
    struct InstructionNode* next;
}
```

This way we can go through a list of instructions and execute one after the other or, if an instruction is a jump instruction, execute the target of the jump after the instruction. To execute a particular instruction node, we check its `type`. Depending on its type, we can access the appropriate fields in one of the structures of the union. If the type is `OUT` (output), for example, we access the field `var_index` in the `output_inst` struct to execute the instruction. Similarly for the `IN` (input) instruction. if the type is `ASSIGN`, we access the appropriate fields in the `assign_inst` struct to execute the instruction and so on.

With this combination of various instructions types in one `struct`, note how the `next` field is now part of the InstructionNode to line up all instructions in a sequence one after another.

This is all fine, but we do not yet know how to generate the list of instructions to execute later. The idea is to have the functions that parses non-terminals return the code that corresponds to the non-terminals, the code being a sequence of instructions. For example for a statement list, we have the following pseudocode (missing many checks):

```
struct InstructionNode* parse_stmt_list()
{
    struct InstructionNode* inst;   // instruction for one statement
    struct InstructionNode* instl;  // instruction list for statement list

    inst = parse_stmt();
    if (nextToken == start of a statement list)
    {
        instl = parse_stmt_list();
        append instl to inst;            // this is pseudocode
        return inst;
    }
```

```
    else
    {
        ungetToken();
        return inst;
    }
}
```

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

```
struct InstructionNode* parse_body()
{
    struct InstructionNode* instl;

    match LBRACE
    instl = parse_stmt_list();
    match RBRACE

    return instl;
}
```

6.3 Handling *if* and *while* statements

More complications occur with *if* and *while* statements. These statements would need to be implemented using the conditional jump (`CJMP`) and the jump (`JMP`) instructions. The conditional jump struct would have the following fields

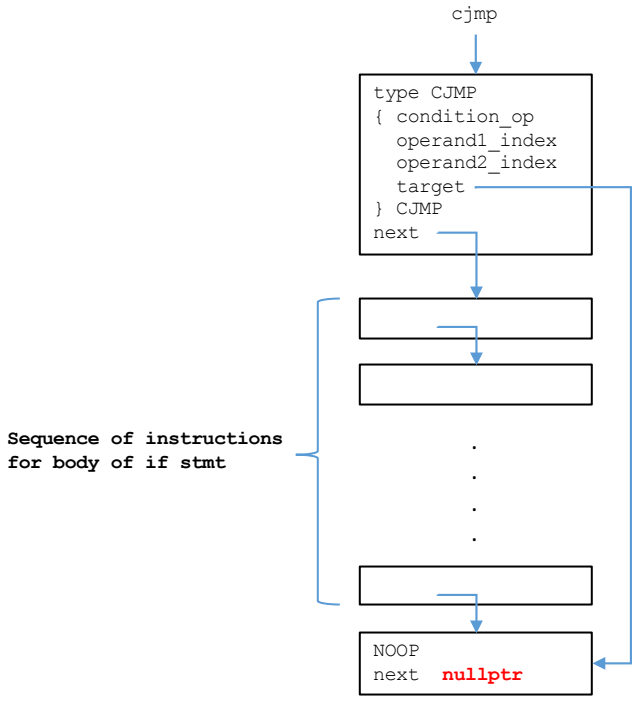
```
struct CJMP {
    ConditionalOperatorType condition_op;
    int operand1_index;
    int operand2_index;
    struct InstructionNode * target;
}
```

The `condition_op`, `operand1_index` and `operand2_index` fields are the operator and operands of the condition of the conditional jump (`CJMP`) instruction. The `target` field is the next instruction to execute if the condition evaluate to `false`. If the condition evaluates to `true`, the next instruction to execute will be the next instruction in the sequence of instructions.
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 InstructionNode* parse_stmt()
{
    ...

    InstructionNode * inst = new InstructionNode;
    if next token is IF
    {
        inst->type = CJMP;

        parse the condition and set inst->cjmp_inst.condition_op,
                                    inst->cjmp_inst.operand1_index and
                                    inst->cjmp_inst.operand2_index

        inst->next = parse_body();                      // parse_body returns a pointer to a sequence of instructions
```

```
        create no-op node                                   // this is a node that does not result
                                                            // in any action being taken.
                                                            // make sure to set the next field to nullptr

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

        ...

        return inst;

    } else ...
}
```

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



The *stmt_list* code should be modified because the code presented above for a *stmt_list* assumed that each statement is represented with one instruction but we have just seen that parsing an *if_list* returns a sequence of instructions. The modification is as follows:

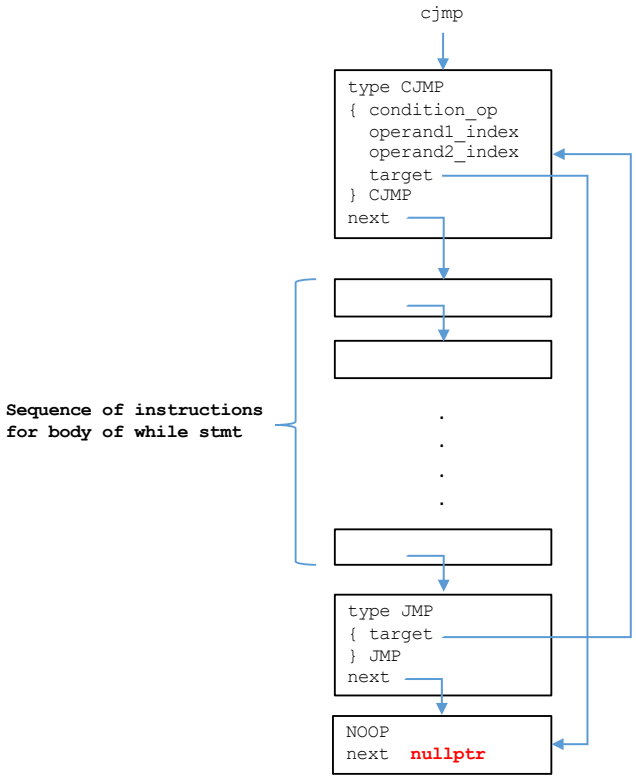
```
struct InstructionNode* parse_stmt_list()
{
    struct InstructionNode* instl1;   // instruction list for stmt
    struct InstructionNode* instl2;  // instruction list for stmt list

    instl1 = parse_stmt();
    if (nextToken == start of a statement list)
    {
```

```
        instl2 = parse_stmt_list();


        append instl2 to instl1

        //        instl1
        //         |
        //         V
        //         .
        //         .
        //         .
        //     last node in
        //     sequence staring
        //     with instl1
        //         |
        //         V
        //       instl2

        return instl1;
    }
    else
    {
        ungetToken();
        return instl1;
    }
}
```

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

```
...

create instruction node inst
if next token is WHILE
{
    inst->type = CJMP;                              // handling WHILE using if and goto nodes

    parse the condition and set inst->cjmp_inst.condition_op, inst->cjmp_inst.operand1 and inst->cjmp_inst.condition_operand2

    inst->next = parse_body();                      // when condition is true the next instruction
                                                    // is the first instruction of the body of while
    create jmp node of type JMP                     // do not forget to set next field to nullptr
    set jmp->jmp_inst.target to inst
    append jmp node to end of body of while
    create no-op node and attach it to the list of instruction after the jmp node
    set inst->cjmp_target.target to point to no-op node

    return inst;

}

...
```

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



6.4 Handling *switch* and *for* statements

You can handle the *switch* and *for* statements similarly, but you should figure that yourself. Use a combination of `JMP` and `CJMP` to support the semantics of the *switch* and *for* statements. See sections 5.8 and 5.7 for the semantics of the *switch* and *for* statements.

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 != nullptr)
{
    switch (pc->type)
    {
       case ASSIGN:      // code to execute pc->assign_stmt ...
                         pc = pc->next

       case CJMP:        // code to evaluate condition ...
                         // depending on the result
                         //   pc = pc->cjmp_inst.target (if condition is false)
```

```
                        // or
                        //   pc = pc->next (if condition is true)

        case NOOP:      pc = pc->next

        case JMP:       pc = pc->jmp_inst.target

        case OUT:       // code to print mem[pc->output_inst.var_index] ...
                        pc = pc->next

        case IN:        // code to read next input value into
                        // mem[pc->input_inst.var_index] and updating
                        // counter for how many values have been read
                        pc = pc->next
    }
}
```

We have provided you with the data structures and the code to execute the graph and **you must use it.** When you submit your code, you will not submit compiler.cc and compiler.h, we will provide them automatically for your submission, so if you modify them, your submission will not compile and run.. You should include `compiler.h` in your code. The entry point of your code is a function declared in `compiler.h`:

```
struct InstructionNode* parse_generate_intermediate_representation();
```

You need to implement this function. In the file demo.cc that we provide, we show a hardcoded example of the function `parse_generate_intermediate_representation()` for a given example input program. I strongly recommend that you draw the data structure that is generated by this hardcoded function to gain a better understanding. If you come to office hours for help, I expect that you will have a drawing of that data structure.

The `main()` function is provided in `compiler.cc`:

```
int main()
{
    struct InstructionNode * 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.

8 Requirements

1. Write a compiler that generates intermediate representation for the code. The interpreter (execute function) is provided.

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

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

9 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. Also, remember that the provided test cases are only provided as examples and they are not meant to be exhaustive in any way.

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: 20
- For statement: 10
- All statements: 10
- **Total:** 100