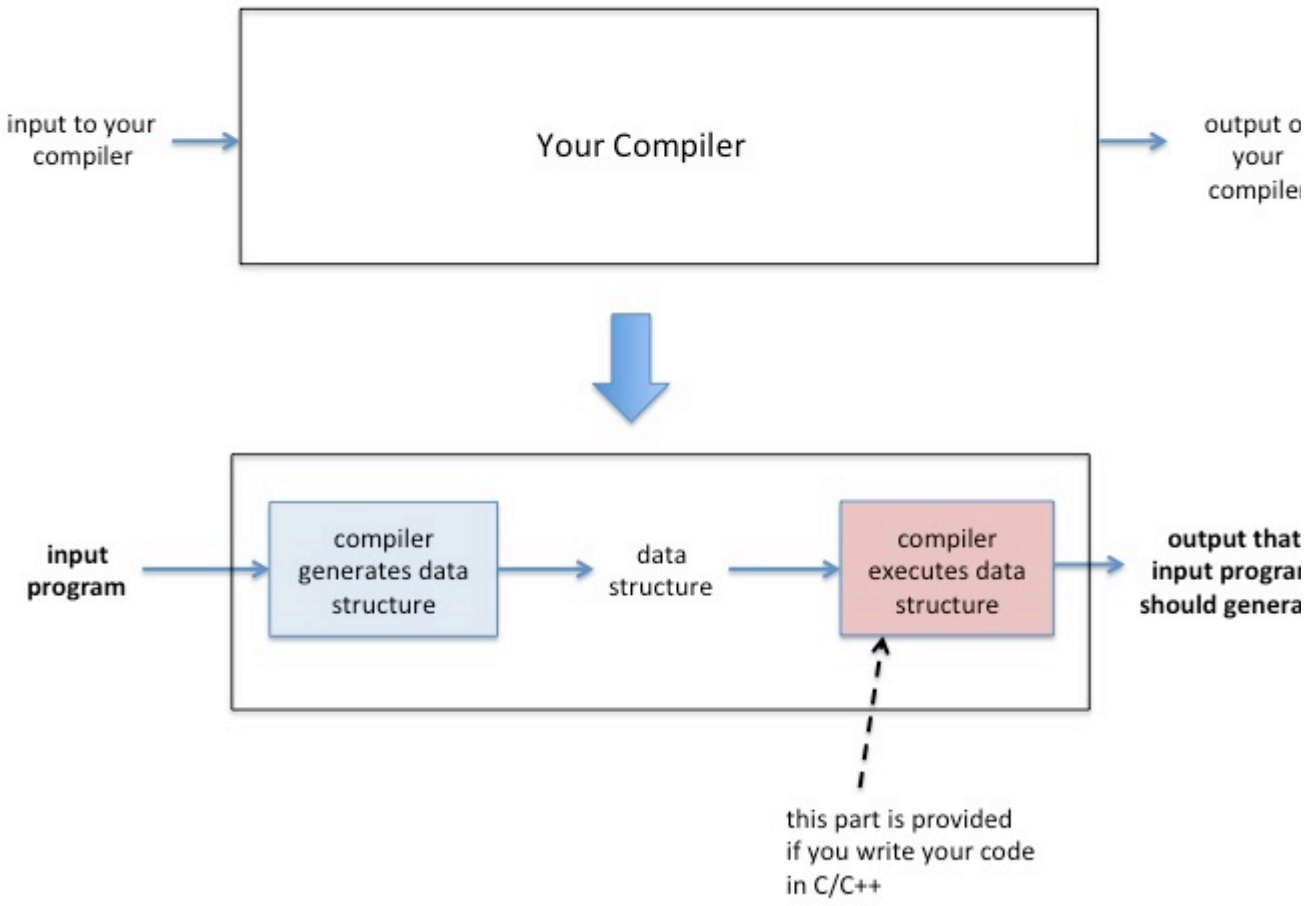
Introduction

You will write a small compiler that will read an input program and represents it in an internal data structure. The data structure will contain representation of 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 (interpret it). 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. These steps are illustrated in the following figure:



The remainder of this document is organized as follows:

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** You are only allowed to use the code we provide to execute the intermediate representation.
5. **Input/Output** Reminds you to only use standard input and output.
6. **Requirements** Lists the programming languages allowed and other requirements.
7. **Submission** Instructions for submitting your project.
8. **Grading** Describes the grading scheme.
9. **Bonus Project** Describes the requirements for the bonus project.

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
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
```

The tokens used in the grammar description are (note PRINT is not what you would expect):

```
SEMICOLON = ;
ID = letter(letter | digit)*
COMMA = ,
LBRACE = {
RBRACE = }
```

```
EQUAL = =
NUM = 0 | (digit digit*)
PLUS = +
MINUS = -
MULT = *
DIV = /
PRINT = print
WHILE = WHILE
IF = IF
GREATER = >
LESS = <
NOTEQUAL = <>
SWITCH = SWITCH
CASE = CASE
COLON = :
DEFAULT = DEFAULT
SEMICOLON = ;
ID = letter(letter | digit)*
COMMA = ,
LBRACE = {
RBRACE = }
LBRAC = [
RBRAC = ]
LPAREN = (
RPAREN = )
EQUAL = =
NUM = 0 | (digit digit*)
PLUS = +
MINUS = -
MULT = *
DIV = /
PRINT = print
WHILE = WHILE
IF = IF
GREATER = >
LESS = <
NOTEQUAL = <>
SWITCH = SWITCH
CASE = CASE
```

```
COLON = :
DEFAULT = DEFAULT
```

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 `]`.

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. A `print` statement is introduced. Note that the `PRINT` keyword is in lower 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)

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.

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.

If Statements

An `if_stmt` has the following (standard) semantics (note that our language does not have an `else` clause to the `if_stmt`):

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

These semantics apply recursively to nested `if_stmt`.

While Statements

`while_stmt` has the following (standard) semantics:

1. The condition is evaluated.
2. If the condition evaluates to `true`, the body of the `while_stmt` is executed, then goto step 1.
3. If the condition evaluates to `false`, then the next statement following the `while_stmt` 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 a `GotoStatement`which is used in conjunction with an `IfStatement` to represent `while` and `switch` statements.

Switch Statements

`switch_stmt` has the following (standard) semantics:

1. The value of the switch variable is checked against each case number in order.

2. If the value matches the case number, the body of the case is executed, then the next statement following the `switch_stmt` is executed.
3. If the value does not match the case number, the 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, then the next statement following the `switch_stmt` is executed.
5. If there is no default case and the value does not match any of the case numbers, then the next statement following the `switch_stmt`is executed.

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

```
SWITCH var {
    CASE n1 : { stmt_list_1 }
    ...
    CASE nk : { stmt_list_k }
}
```

is equivalent to:

```
IF var == n1 {
    stmt_list_1
    goto label
}
...
IF var == nk {
    stmt_list_k
    goto label
}
label:
```

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

```
SWITCH var {
    CASE n1 : { stmt_list_1 }
    ...
    CASE nk : { stmt_list_k }
    DEFAULT : { stmt_list_default }
}
```

is equivalent to:

```
IF var == n1 {
    stmt_list_1
    goto label
}
...
IF var == nk {
    stmt_list_k
    goto label
}
stmt_list_default
label:
```

Print statement

The statement:

```
print a;
```

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

How to generate the code

The intermediate code will be a data structure (a graph) that is easy to interpret and execute. All intermediate representation data structures are defined in `compiler.h`, and you are not allowed to change `compiler.h` or `compiler.c`, which are posted on the submission site.

You should become very familiar with the intermediate representation data structures in `compiler.h` and their usage to execute the program in the `execute_program` function in `compiler.c`.

We will start with describing the graph for assignments, then extend this to `while` statements. You should read this whole explanation.

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

This structure only accepts `ValueNode` as operands. To handle literal constants (`NUM`), you need to create a `ValueNode` for them and store them in the created `ValueNode` 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. Let's consider them one at a time.

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 {
    int 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 first check its `type`. If it is `PRINT_STMT`, we execute the `print_stmt` field, if it 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 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 pseudocode:

```
struct StatementNode* parse_body()
{
    struct StatementNode* stl;
    match LBRACE
    stl = parse_stmt_list();
    match RBRACE

    return stl;
}
```

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 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`.

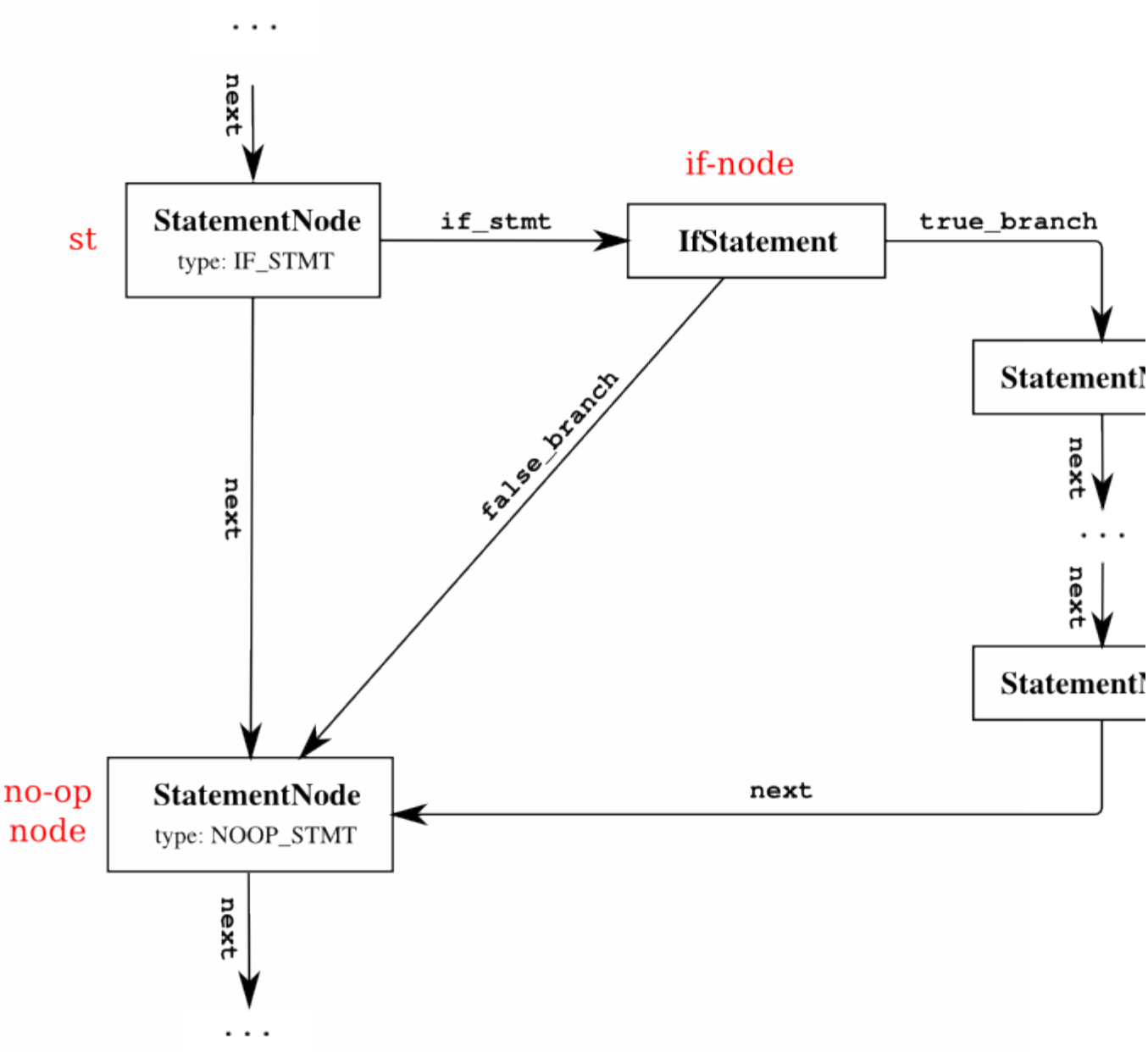
```
struct GotoStatement {
    struct StatementNode* target;
}
```

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 or 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();
```

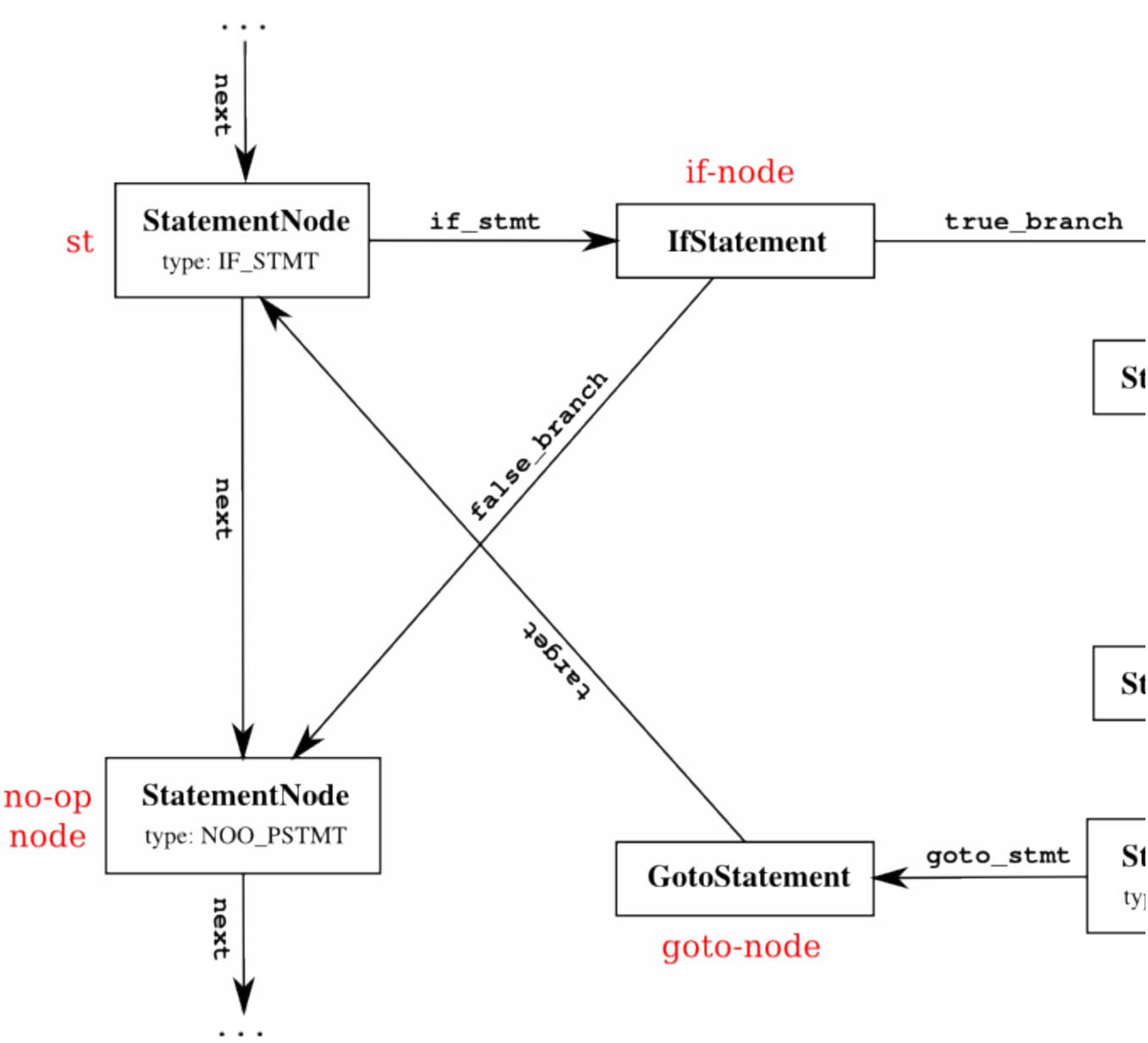
```
        if st->type == IF_STMT
        {
            append stl to the no-op node that follows st

            //         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:



Handling `switch` statement

You can handle the `switch` statement similarly to `while` and `if`. Use a combination of `IfStatement` and `GotoStatement` to support the semantics of the `switch` statement.

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
    }
}
```

Implementation

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.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_generate_intermediate_representation();
```

You need to implement this function.

The `main()` function is given in `compiler.c`:

```
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. 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.

Input/Output

As in the previous project, 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.

Requirements

1. Write a compiler that generates intermediate representation for the code.

2. **Language:** You can use C or C++ for this assignment.
3. **Platform:** As previous projects, the reference platform is CentOS
   a.
4. **You can assume that there are no syntax or semantic errors in the input program.**