# CSC4180 Macro Compiler Construction Assignment 1

# Student ID: 120090712

## Project structure

```
The structure of my workspace:
.
├── csc4180_A1_120090712.pdf
├── testcases
└── src
    ├── Makefile
    ├── ir_generator.cpp
    ├── ir_generator.hpp
    ├── node.cpp
    ├── node.hpp
    ├── scanner.I
    ├── parser.y
    ├── run_compiler.sh
    └── main.cpp
```

## How to execute the compiler

Uplaod the whole file to the cluster and use the following commands:

```
cd path/to/project
cd src
bash run_compiler.sh
```

The expected output will be:

```
test0
export parse tree filename: ./cst/test0.dot
export parse tree filename: ./ast/test0.dot
|--Pass
test1
export parse tree filename: ./cst/test1.dot
export parse tree filename: ./ast/test1.dot
|--Pass
test2
export parse tree filename: ./cst/test2.dot
export parse tree filename: ./ast/test2.dot
|--Pass
test3
```

```
export parse tree filename: ./cst/test3.dot
export parse tree filename: ./ast/test3.dot
|--Pass
......
```

Then you can check the result in /project/testcases/ast|cst...

## How did I design the Scanner

In designing the scanner, I have utilized Flex (Fast Lexical Analyzer Generator) to define a set of rules that describe how the input text should be tokenized. Here's a breakdown of my design:

### Regular Expressions and Rules:

```
In this section, I define regular expressions for various tokens and specify the
corresponding actions to be taken when a match is found.
In addition to the tokens mentioned in the PDF introduction file, I have also
added additional tokens to skip:
```

```
SPACES  (\t|\0|\r|\n|\ )+ // ignore
COMMENT --.*\n
BEGIN_ "begin"
END "end"
READ "read"
WRITE "write"
LPAREN "("
RPAREN ")"
SEMICOLON ";"
COMMA ","
ASSIGNOP ":="
PLUSOP "+"
MINUSOP "-"
ID [a-zA-Z][a-zA-Z0-9_]{0,31}
INTLITERAL  -?[0-9]+
```

### Rules for Token Recognition and Printing:

```
Here, I define the rules for recognizing different tokens and specify actions to
be performed when a token is found. If scan_only is set, it prints the token
information in the format <token-class, lexeme>.

An example:
```

```
{BEGIN_} {
    if(scan_only == 1) {
        printf("<BEGIN_, %s>\n", yytext);
        return 1; // Keep the while looping
    }
    return TOK_BEGIN; // Token pass to parser
}
{END} {
    // Similar actions for other tokens...
}
```

The scanning result looks like:

```
<BEGIN_, begin>
<ID, A>
<ASSIGNOP, :=>
<INTLITERAL, 10>
<SEMICOLON, ;>
<ID, B>
<ASSIGNOP, :=>
<ID, A>
<PLUSOP, +>
<INTLITERAL, 20>
<SEMICOLON, ;>
<WRITE, write>
<LPAREN, (>
<ID, B>
<RPAREN, )>
<SEMICOLON, ;>
<END, end>
<SCANEOF>
```

## EOF Handling:

```
I handle the end-of-file condition (<<EOF>>) separately. If scan_only is set, it
prints <SCANEOF> and returns 0 to stop the scanning process(while looping in main
function):
```

```
<<EOF>> {
    if(scan_only == 1) {
        printf("<SCANEOF>\n");
        // stop the while loop
        return 0;
    }
    return TOK_SCANEOF;
}
```

## How did I design the Parser

### About yydata type

```
In this parser, I've used Bison syntax to define the grammar rules. The %union
section specifies the possible types for semantic values. The %token section lists
the terminal tokens, and %type specifies the types for non-terminal symbols.
```

```
%union {
    int intval;
    const char* strval;
    struct Node* nodeval;
}

// Define terminal symbols with %token.
%token <intval> TOK_INTLITERAL
%token <strval> TOK_BEGIN TOK_END TOK_READ TOK_WRITE TOK_LPAREN TOK_RPAREN
TOK_SEMICOLON TOK_COMMA TOK_ASSIGNOP TOK_PLUSOP TOK_MINUSOP TOK_ID TOK_SCANEOF

// Start Symbol
%start start

// Define Non-Terminal Symbols with %type.
%type <nodeval> program statement_list statement id_list expression_list
expression primary
```

### About CFG Rules

```
Similar to designing a tree data structure, I only need to connect the following
rules together using nodes:
```

```
1. <start> → <program> SCANEOF
2. <program> → BEGIN <statement list> END
3. <statement list> → <statement> {<statement>}
4. <statement> → ID ASSIGNOP <expression>;
5. <statement> → READ LPAREN <id list> RPAREN;
6. <statement> → WRITE LPAREN<expr list> RPAREN;
7. <id list > → ID {COMMA ID}
8. <expr list > → <expression> {COMMA <expression>}
9. <expression> → <primary> {<add op> <primary>}
10. <primary> → LPAREN <expression> RPAREN
11. <primary> → ID
12. <primary> → INTLITERAL
```

```
13. <add op> → PLUSOP
14. <add op> → MINUSOP
```

```
In this process, I need to pay attention to removing some unnecessary nodes and
tokens, like this:
```

```
statement_list  : statement { // First kind of
                    $$ = new Node(SymbolClass::STATEMENT_LIST);
                    $$ ->append_child($1);
                }
                | statement_list statement  {
                    if (cst_only == 1){
                        // cst
                        $$ = new Node(SymbolClass::STATEMENT_LIST);
                        $$->append_child($1);
                        $$->append_child($2);
                    } else {
                        // ast
                        $1->append_child($2);
                        $$ = $1;
                    }
                };
```

## How did I design the IR generator

As I already have a complete AST from the parser, I just need to translate the nodes into the LLVM IR. So, I design the following functions to help me finish it:

```
    void export_ast_to_llvm_ir(Node* node){
        /* Recieve the root node and start the process */
    }

    void gen_llvm_ir(Node* node){
        /* Distingush the Symbol class of the nodes and apply proper functions,
also, to make sure every node will be considered, it will call itself, which means
it is recursive */
    }

    void gen_read_llvm_ir(Node* node){
        /* A function which is designed to handle the <read> node, I have wirte
some string format to get the desired string output */
    }

    void gen_write_llvm_ir(Node* node){
        /* Similar with "gen_read_llvm_ir(Node* node)", it is designed to handle
the <write> node */
```

```
    }

    vector<string> tmp_register and std::string find_tmp_register(){
        /* Working together, whenever a data or variable needs to get a tempoary
storage, this function would return the available number of the register(Small
priority) */
    }

    std::string reference(Node* node){
        /* This function is working for the rvalue or called variables, it will
make sure the proper string is "translate" into IR*/
    }

    std::string combine(Node* node){
        /* Similar with reference, but this function is working for operators like
+ or -. According to the AST structure, to make sure all the value and variables
are processed, it is recursive designed */
    }

    void gen_assignop_llvm_ir(Node* node){
        /* The assign operator is the closest to the root node, basicly, it will
call the other functions to finish the work */
    }
```