# CSC4180 Macro Compiler Construction Assignment 2

# Student ID: 120090712

## Project structure

```
The structure of my workspace:
.
├── csc4180_A2_120090712.pdf
├── testcases
└── src
    ├── Makefile
    ├── scanner.cpp
    ├── scanner.hpp
    ├── tokens.cpp
    ├── tokens.hpp
    ├── lexer.l
    ├── compile_test.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 compile_test.sh
```

Then you will see 10 files in the `./A2` directory

## How did I design and implement this assignment

Scanner by Flex:

**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|\ )+
NUL                 "null"
TRUE                "true"
FALSE               "false"
TVOID               "void"
TINT                "int"
TSTRING             "string"
TBOOL               "bool"
IF                  "if"
ELSE                "else"
WHILE               "while"
FOR                 "for"
RETURN              "return"
NEW                 "new"
VAR                 "var"
GLOBAL              "global"
LPAREN              "("
RPAREN              ")"
LBRACE              "{"
RBRACE              "}"
LBRACKET            "["
RBRACKET            "]"
NOT                 "!"
TILDE               "~"
ASSIGN              "="
SEMICOLON           ";"
COMMA               ","
STAR                "*"
PLUS                "+"
MINUS               "-"
LSHIFT              "<<"
RLSHIFT             ">>"
RASHIFT             ">>>"
LESS                "<"
LESSEQ              "<="
GREAT               ">"
GREATEQ             ">="
EQ                  "=="
NEQ                 "!="
LAND                "&"
LOR                 "|"
BAND                "[&]"
BOR                 "[|]"
INTEGER             [0-9]+
ID                  [a-zA-Z][a-zA-Z0-9_]*
COMMENT             \/\*([^*]|\*+[^*/])*\*+\/
STRING              \"(.)*\"
```

**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:
```

```
{SPACES} { /* Skip */ }
{COMMENT} { /* Skip */ }
{NUL} {
  printf("NUL %s\n",yytext);
}
{TRUE} {
  printf("TRUE %s\n",yytext);
}
{FALSE} {
  printf("FALSE %s\n",yytext);
}
// Similar actions for other tokens...
```

The scanning result looks like:

```
TINT int
ID main
LPAREN (
RPAREN )
LBRACE {
VAR var
ID str
ASSIGN =
STRINGLITERAL "hello world!"
SEMICOLON ;
ID print_string
LPAREN (
ID str
RPAREN )
SEMICOLON ;
RETURN return
INTLITERAL 0
SEMICOLON ;
RBRACE }
```

## Scanner by Hand:

While implementing the scanner, I first create some operations between NFAs, like:

```
void concat(NFA* from);

void set_union(NFA* from);

void set_union(std::set<NFA*> set);

void kleene_star();
```

Then I finish the functions of specific regular expression, which are used to setting some token forms:

```
static NFA* from_string(std::string str);

static NFA* from_letter();

static NFA* from_digit();

static NFA* from_any_char();

// All of these functions above is serving for the following functions:
void add_token(std::string token_str, TokenClass token_class, unsigned int
precedence = 100);

void add_identifier_token(TokenClass token_class, unsigned int precedence = 50);

void add_integer_token(TokenClass token_class, unsigned int precedence = 50);

void add_string_token(TokenClass token_class, unsigned int precedence = 50);

void add_comment_token(TokenClass token_class, unsigned int precedence = 50);
```

The last part is the method of translating NFA to DFA using the subset construction:

**Algorithm 1** Subset Construtions

Initially, $\epsilon - closure(s_0)$ is the only state in $Dstates$ and it is unmarked;
**while** there is an unmarked state $T$ in $Dstates$ **do**
  mark $T$;
  **for** each input symbol a in alphabet **do**
    $U := \epsilon - closure(move(T, a))$
    **if** $U$ is not in $Dstates$ **then**
      add $U$ as an unmarked state to $Dstates$;
    **end if**
    $Dtran[T, a] := U$
  **end for**
**end while**

Here, I use BFS(implemented by queue) to go through the states:

```
/**
 * Determinize NFA to DFA by subset construction
 * @return DFA
 */
```

```
DFA* NFA::to_DFA() {
    // Create a DFA object
    DFA* dfa = new DFA();


    // Saving the processes states(NFA)
    std::map<std::set<State*>, DFA::State*> states_dfa_map;

    // Initializationn including ε-closure
    std::set<State*> initial_closure = epsilon_closure(start);
    auto init_state = new DFA::State();
    states_dfa_map[initial_closure] = init_state;
    dfa->states.push_back(init_state);

    // Using queue BFS
    std::queue<std::set<State*>> state_queue;
    state_queue.push(initial_closure);

    // Begin BFS
    while (!state_queue.empty()) {
        std::set<State*> current_closure = state_queue.front();
        state_queue.pop();

        // Go through every possible char (ASCII 128)
        for (char c = 0; c < 127; ++c) {
            // Get the closure if using char `c`
            std::set<State*> neighbor_states = move(current_closure, c);
            std::set<State*> next_closure;
            for (auto in_state : neighbor_states){
                std::set<State*> temp = epsilon_closure(in_state);
                next_closure.insert(temp.begin(),temp.end());
            }

            // If the arrivable next closure is not empty
            if (!neighbor_states.empty()) {
                // If this closure is not in the `states_dfa_map`
                if (states_dfa_map.find(next_closure) == states_dfa_map.end()) {
                    // Create and put a new DFA state int to the map
                    auto dfa_state= new DFA::State();
                    states_dfa_map[next_closure] = dfa_state;
                    dfa->states.push_back(dfa_state);
                    state_queue.push(next_closure);
                }
                // Build connection between current_closure and next_closure
                states_dfa_map[current_closure]->transition[c] =
states_dfa_map[next_closure];
            }
        }
    }

    // Set the DFA `accepted` value
    for (auto &t : states_dfa_map) {
        for (auto nfa_state : t.first) {
            if (nfa_state->accepted) {
```

```
                t.second->accepted = true;
                t.second->token_class = nfa_state->token_class;
                break;
            }
        }
    }
    return dfa;
}
```

## Why we choose regular expression to represent lexical specification

Regular expressions are chosen to represent lexical specifications for several reasons:

1. **Expressiveness**: Regular expressions provide a concise and expressive way to describe patterns of characters or strings. They allow defining complex patterns with a relatively simple syntax.
2. **Flexibility**: Regular expressions support a wide range of operations such as alternation, concatenation, repetition, and grouping, allowing for the specification of various patterns effectively.
3. **Widespread Adoption**: Regular expressions are widely used and supported in many programming languages, tools, and libraries. This makes them a common choice for specifying lexical structures in compilers, parsers, text processing utilities, and more.
4. **Efficiency**: Regular expressions can be efficiently processed using algorithms like finite automata, which can quickly determine whether a given input string matches a specified pattern. This efficiency is crucial for lexical analysis, which is often a performance-critical part of a compiler or text processing system.
5. **Ease of Understanding**: Regular expressions provide a compact and intuitive way to represent patterns, making it easier for developers to understand and maintain lexical specifications.

## Why NFA is more suitable than DFA to recognize regular expression, and how did I enable NFA to recognize different kinds of regular expression?

Nondeterministic Finite Automata (NFA) are often considered more suitable than Deterministic Finite Automata (DFA) for recognizing regular expressions due to their greater flexibility and simplicity in construction. Here's why NFA is preferred:

1. **Flexibility**: NFAs allow multiple transitions from a single state on the same input symbol or even on epsilon transitions (ε-transitions). This flexibility makes it easier to handle certain aspects of regular expressions, such as optional parts, repetitions, and alternatives.
2. **Simplicity in Construction**: NFAs can be constructed directly from regular expressions using a systematic approach. Each element of the regular expression (e.g., characters, concatenation, alternation, repetition) can be mapped to corresponding transitions in the NFA, resulting in a straightforward construction process.
3. **Ease of Understanding**: NFAs often closely resemble the structure of the regular expressions from which they are derived. This similarity makes it easier for developers to understand and reason about the behavior of the NFA in relation to the original regular expression.

To enable an NFA to recognize different kinds of regular expressions, I typically construct the NFA based on the structure of the regular expression. Each component of the regular expression (e.g., characters, alternations, repetitions) would be translated into corresponding transitions and states in the NFA.

For example, to recognize the regular expression `[a-zA-Z]`, I would create an NFA with states representing the characters from 'a' to 'z' and from 'A' to 'Z', and transitions between these states on the corresponding characters.

```
/**
 * RegExp: [a-zA-Z]
 * @return
 */
NFA* NFA::from_letter() {
    // Create start and end
    State* start = new State();
    State* end = new State();

    // Create transition
    for (char c = 'a'; c <= 'z'; ++c) {
        start->transition[c].insert(end);
    }
    for (char c = 'A'; c <= 'Z'; ++c) {
        start->transition[c].insert(end);
    }

    // Create nfa
    NFA* nfa = new NFA();
    nfa->start = start;
    nfa->end = end;

    return nfa;
}
```

Similarly, for more complex regular expressions involving alternations, concatenations, and repetitions, I construct the NFA accordingly, ensuring that the NFA can transition through the states according to the rules specified by the regular expression:

```
void Scanner::add_identifier_token(TokenClass token_class, unsigned int
precedence) {
    // start with letter
    NFA* identifier_nfa = NFA::from_letter();
    // following with Letter, digit, underscore
    NFA* letter_digit_underscore_nfa = NFA::from_letter();
    letter_digit_underscore_nfa->set_union(NFA::from_digit());
    letter_digit_underscore_nfa->set_union(NFA::from_string("_"));

    letter_digit_underscore_nfa->kleene_star(); // repeat
    identifier_nfa->concat(letter_digit_underscore_nfa);

    // Set precedence
```

```
    identifier_nfa->set_token_class_for_end_state(token_class, precedence);

    // Union with primary nfa
    nfa->set_union(identifier_nfa);
}
```

## How did I enable my scanner to always recognize the longest match and the most precedent match?

Here, in my scanner, the code will stop scanning a token when there is not state for a char `c` to go further, which means it has reached the longest way it can possible go. But considering some special cases(like 'comments' and 'string'), the code my not able to stop. So every time the code will go further, it will check whether it reach a `comments` or `string` token class and do something special. Here is the detials:

```
    for (char c : source_code) {
        if (pointer == start_dfa_state){
            if (pointer->transition.find(c) == pointer->transition.end()){
                // Ignored
                continue;
            }
        }
        auto index = pointer->transition.find(c);
        // If it can continue
        if (index != pointer->transition.end()) {
            if (pointer->accepted && (pointer->token_class == STRINGLITERAL ||
pointer->token_class == COMMENT)){
                // Don't continue
            } else {
                temp += c;
                pointer = pointer->transition[c];
                continue;
            }
        }

        // Find the token class
        if (pointer->accepted){
            if (pointer->token_class != COMMENT) printf("%s
%s\n",token_class_to_str(pointer->token_class).c_str(),temp.c_str());
            temp = "";
        } else {
            // Something wrong!
            printf("Unkown %s\n",temp.c_str());
            temp = "";
        }

        pointer = start_dfa_state;
        index = pointer->transition.find(c);
        if (index != pointer->transition.end()) {
            temp += c;
            pointer = pointer->transition[c];
            continue;
```

```
        }
    }
```

## Why do we still need to convert NFA into DFA for lexical analysis in most cases

While Nondeterministic Finite Automata (NFA) are more flexible and easier to construct from regular expressions, Deterministic Finite Automata (DFA) are often preferred for lexical analysis in most cases for several reasons:

1. **Determinism**: DFAs have deterministic behavior, meaning that for any given input symbol, there is only one possible transition from each state. This determinism simplifies the implementation and analysis of lexical analyzers.

2. **Efficiency**: DFAs are generally more efficient in terms of memory and processing time compared to NFAs. DFAs have a fixed number of transitions per state, making them more predictable and easier to optimize for performance.

3. **Deterministic Lookup**: DFAs can be implemented using lookup tables, where the next state is determined by the current state and the input symbol. This allows for constant-time transitions, making lexical analysis faster, especially for large input streams.

Overall, while NFAs are more expressive and easier to construct from regular expressions, DFAs are preferred for lexical analysis due to their deterministic nature, efficiency, and ease of implementation and optimization. Therefore, it is common practice to convert NFAs into DFAs as part of the lexical analysis process in order to achieve better performance and reliability.