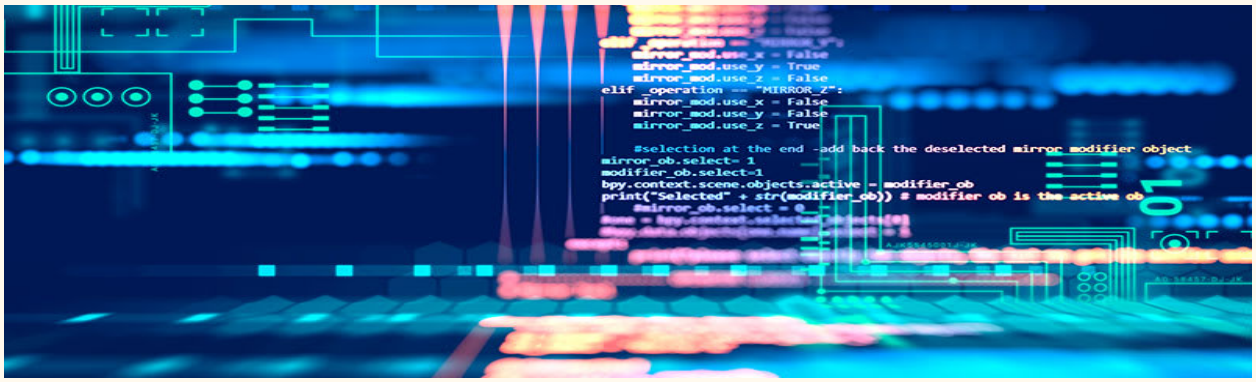
Computer and System Engineering Department

Programming languages and compiler

2022, 2023 semester 1



Phase - 1

Due Date: 7/12/2022

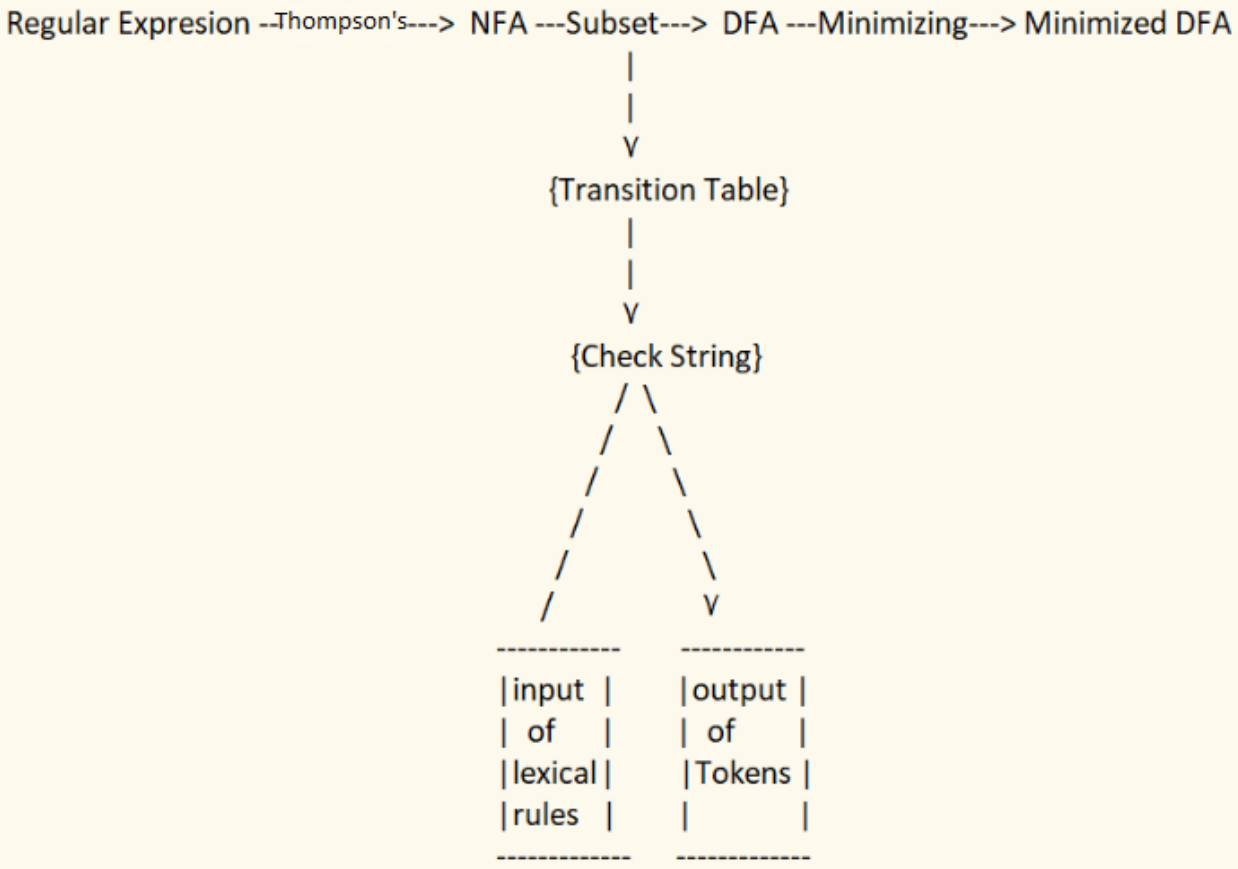
Overview

This phase of the assignment aims to practice techniques for building automatic lexical analyzer generator tools.

Team

Name	ID
Amr Ahmed Fouad Mohamed Mostafa Al-Dhahabi	18011174
Amr Ayman Ibrahim Momtaz Ibrahim Momtaz	18011178
Mohannad Tarek Abd El-Khalek Mahmoud	18011889
Momen Ibrahim Fawzy Hassan	18011896

Phase I: Overview



1. Thompson's Algorithm {Regular Expression ---Thompson's Algorithm---> NFA}
2. Subset {NFA ---Subset---> DFA}
3. Minimizing {DFA ---Minimizing---> Minimized DFA}
4. Check String {Input file ------> Output file of Tokens}

1-Parsing grammar and Thompson's algorithm

In this phase, We parsed the regular grammar file containing the regular expressions and the regular definitions and we used Thompson's construction algorithm to create the NFA.

Assumptions: The only assumption taken in this part concerns the precedence of operations. The order should be parentheses, kleene closure, positive closure, concatenation then union. However, the order in the program is the same except that it doesn't give priority of concatenation over union. Instead it uses from left to right. The problem to be fixed requires the creation of an unambiguous grammar which enforces this order and we didn't learn something like this before. So we used extra parentheses only in the num regular definition.

Data Structures and modules:

1. **Transition Table:** The transition is the data structure which is used to to represent the transition table and it consists of two maps. One map contains the rows and columns of the table which is created using an unordered_map where the key is int which is the state number and the column is another map which maps each input character to the incoming the next states it can go for under that input. The other map contains the accepting states where each key is the accepting state number and the value is the accepting state name.
2. **Grammar IO module:** This module handles all the interaction done with the grammar input file.Functionalities are the following:
    a. It reads and parses all the regular definitions and expressions
    b. It determines the priority of the accepting state giving highest priority to keywords and punctuations then the priority is assigned by order of declaration.
    c. It cleans a given token by removing the \ and handling lambda case.

3. **Nfa Node:** It represents a node in the graph of the NFA. It contains only an ID which is the state number and a map which contains its children where the key is the character which makes the transition to another child node.
4. **NFA graph:** This module handles all the graph operations discussed by Thompson's algorithm (Union, Concatenation, Kleene and positive closure). And has a map which contains all the regular expressions where the key is the regular expression name and the value is a vector of characters representing that regular expression. And it's responsible for creating new states and representing certain token to its corresponding automata.
5. **NFA:** This module handles the creation of the NFA. It has three instances of Grammar IO, NFA Graph and Transition Table modules. It gets all the regular expressions and definitions using the Grammar IO module. It scans the regular definitions and sends the corresponding operations to the NFA Graph module which performs the operation. It performs the recursive calls when parentheses appear. Finally, it combines all the patterns starting nodes with node zero and it traverses the final graph using dfs (to reduce memory) and it fills the Transition Table while traversing the tree. The NFA module acts as the control unit of the NFA.

2- Subset

In this phase, We traversed the graph/table produced by the NFA module to create the DFA by gathering all the states that we're at currently using epsilon closure and the input that was given and creating a new state for them to create a graph/table where we're at one node at a time, then adding the newly created node in a queue and we loop keep looping over the queue and popping and performing the operation mentioned above until all nodes are computed.

Assumptions: During the creation of the graph/table we didn't create a "dead state" as it is implicitly understood by the absence of the input character in the table as we're using maps.

Data Structures and Methods:

1. **NFA:** A reference to the NFA generated by reading the grammar to get the graph/table and methods that can help in determining the correct acceptor if we need to break a tie.
2. **table:** The DFA transition table created from the given NFA.
3. **accepting_states:** The new accepting states table after applying breaking ties by checking the priorities.
4. **get_or_create_state**: Method that either creates a new state out of a collection of states or returns the singular state that represents the collection if they've been processed before
5. **add_accepting_state**: Method that creates an accepting state given a collection of states by checking whether the states included are accepting states, if more than one state exists then we perform tie breaking by using priorities.
6. **compute_column**: Method that given a collection of states, gets the epsilon closure of the states and assigns a new state for them if they weren't computed before and adds the state to the accepting states if it wasn't added before.
7. **get_states_vector_from_set**: Sorts and returns the states that were stored in a set.
8. **get_epsilon_closure**: Given a collection of states it returns a set containing the states as well as their epsilon closure.

3-Minimizing

Assumptions:

We are assuming the absence of the dead state. It's only included in the process of minimization, but not in the transition table.

Method:

First partition the states into 2 groups, (non final states) and (final states).

Then for each state check if it's equivalent to its neighboring states in the same partition. Two states are equivalent if under every transition both go to 2 states that belong to the same partition. If they are final states they should accept the same pattern.

A partition is divided such that every subpartition contains equivalent states.

The previous process is repeated until no more partitions are created.

Finally use those partitions to form new states and create a new transition table.

Data Structures and modules:

DFA_Minimized: This module performs the previous process and returns the new minimized transition table. For representing the (**partitions)** we used list<list<int>> where each list is a partition that contains the ids of states. It is more efficient than vectors (dynamic array) as the most performed operation is deletion (when 2 states are equivalent). We also defined an array (**indices)** that contains the partition index of each state. Therefore checking if 2 states are in the same partition or not is O(1). Checking if 2 states are equivalent takes O(d) where d is the degree of both states. The dead state is added in calculation just for completeness, but not added to the final table.

4-Check String

In this phase, the lexical analyzer is generated and has to read its input one character at a time, until it finds the longest prefix of the input, which matches one of the given regular expressions. It should create a symbol table and insert each identifier in the table. If more than one regular expression matches some longest prefix of the input, the lexical analyzer should break the tie in favor of the regular expression listed first in the regular specifications. If a match exists, the lexical analyzer should produce the token class and the attribute value. If none of the regular expressions matches any input prefix, an error recovery routine is to be called to print an error message and to continue looking for tokens.

The output of our program:

The input file: file

Input file example for the above lexical rules:

letter = a-z | A-Z
digit = 0 - 9
*id: letter (letter|digit)**
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits (\L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: =
{ if else while }
[; , \(\) { }]
addop: \+ | -
*mulop: * | /*

If the input file is: file

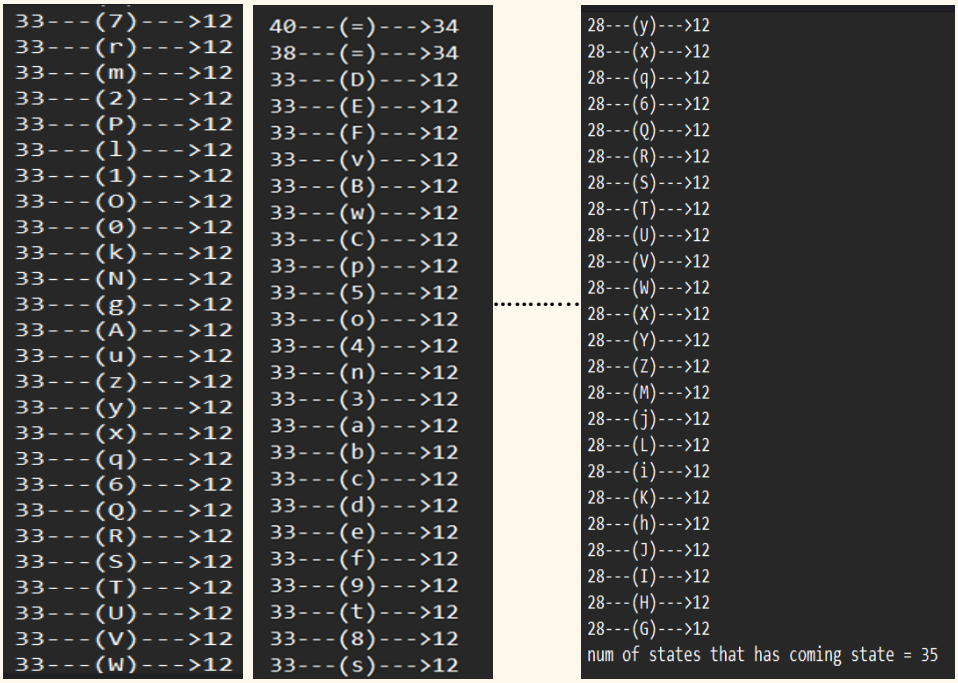
Test Program

```
int sum , count , pass , mnt; while (pass != 10)
{
        pass = pass + 1 ;
}
```

The output file is: file

```
int : int
sum : id
, : ,
count : id
, : ,
pass : id
, : ,
mnt : id
; : ;
while : while
( : (
pass : id
!= : relop
10 : num
) : )
{ : {
pass : id
= : assign
pass : id
+ : addop
1 : num
; : ;
} : }
 : Not Defined!
```

And the minimal DFA is: file



And we construct also the symbol table of the given identifiers:

```
-------- Symbol Table --------
count|id
mnt|id
pass|id
sum|id
```

Testing other program:

We tested our program using question 5 in sheet 2: file

Construct nondeterministic finite state acceptor, a combined NFA and then a minimal DFA for the following regular definitions:

if → if
then → then
else → else
relop → < | <= | = | <> | > | >=
id → letter (letter | digit)*
num → digit digit*

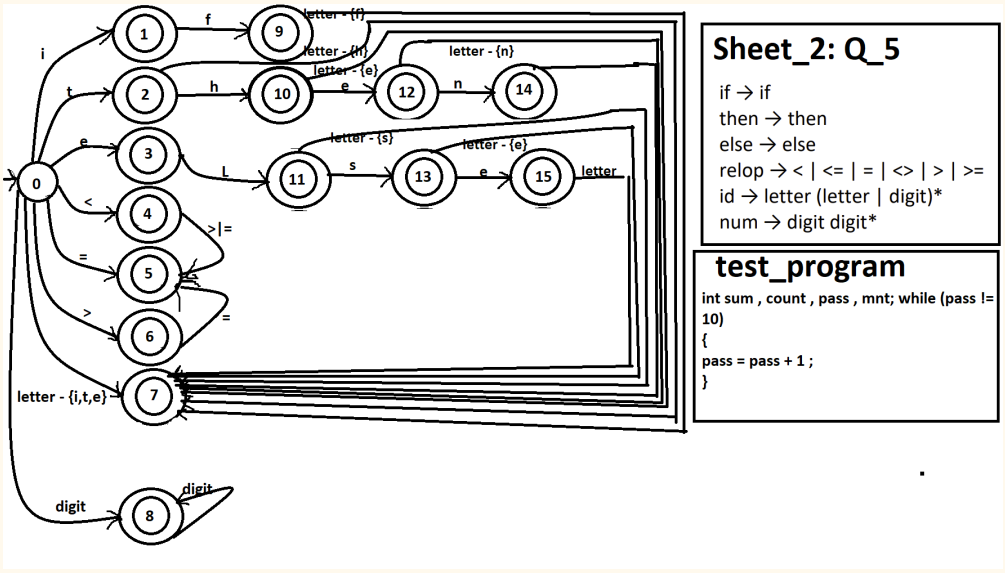
We got that minimal DFA is: file

```
16---(>)--->12
16---(=)--->12
15---(0)--->15
15---(1)--->15
15---(2)--->15
15---(3)--->15
15---(4)--->15
15---(5)--->15
15---(6)--->15
15---(7)--->15
15---(8)--->15
15---(9)--->15
0---(J)--->7
0---(h)--->7
0---(K)--->7
0---(L)--->7
0---(j)--->7
0---(M)--->7
0---(l)--->7
0---(1)--->15
0---(=)--->12
0---(x)--->7
0---(5)--->15
0---(p)--->7
0---(i)--->11
```

```
0---(6)--->15
0---(q)--->7
0---(7)--->15
0---(r)--->7
0---(t)--->8
0---(9)--->15
0---(8)--->15
0---(s)--->7
0---(D)--->7
0---(E)--->7
0---(y)--->7
0---(>)--->14
0---(F)--->7
0---(z)--->7
0---(G)--->7
0---(e)--->9
0---(H)--->7
0---(I)--->7
0---(u)--->7
0---(A)--->7
0---(B)--->7
0---(C)--->7
0---(3)--->15
0---(n)--->7
0---(v)--->7
0---(<)--->16
```

```
11---(z)--->7
11---(y)--->7
11---(x)--->7
11---(Q)--->7
11---(R)--->7
11---(S)--->7
11---(T)--->7
11---(U)--->7
11---(V)--->7
11---(W)--->7
11---(X)--->7
11---(Y)--->7
11---(Z)--->7
11---(M)--->7
11---(j)--->7
11---(L)--->7
11---(i)--->7
11---(K)--->7
11---(h)--->7
11---(J)--->7
11---(I)--->7
11---(H)--->7
11---(G)--->7
14---(=)--->12
num of states that has coming state = 15
```

Which represents the following graph:



If we used the same test program: file

We got the following output: file

```
int : id
sum : id
, : Not Defined!
count : id
, : Not Defined!
pass : id
, : Not Defined!
mnt : id
; : Not Defined!
while : id
( : Not Defined!
pass : id
! : Not Defined!
= : relop
10 : num
) : Not Defined!
{ : Not Defined!
pass : id
```

```
= : relop
pass : id
+ : Not Defined!
1 : num
; : Not Defined!
} : Not Defined!
◆ : Not Defined!
```

And we construct also the symbol table of the given identifiers:

```
-------- Symbol Table --------
count|id
int|id
mnt|id
pass|id
sum|id
while|id
```

Bonus:

We used Flex to automatically generate a lexical analyzer for the given regular expressions.

The expressions: file

```
%{
%}

%option noyywrap

DIGIT [0-9]
LETTER [a-zA-Z]
DIGITS {DIGIT}+
%%
boolean|int|float|if|else|while {printf("%s\n", yytext);}
{LETTER}({LETTER}|{DIGIT})* {printf("id\n");}
(({DIGITS})|({DIGITS}"."{DIGITS}))(("E"{DIGITS})?) printf("num\n");
"=="|"!="|">"|">="|"<"|"<=" {printf("relop\n");}
"=" {printf("assign\n");}
;|, {printf("%s\n", yytext);}
"{" printf("{\n");
```

```
"}" printf("}\n");
"(" printf("(\n");
")" printf(")\n");
"+"|"-" {printf("addop\n");}
"*"|"/" {printf("mulop\n");}
[ \t\n\r]+
. printf("Not Defined!\n");
%%
```

And when we tested it on the given program test we got that output: executable

Input: `int sum , count , pass , mnt; while (pass !=10){pass = pass + 1 ;}`

```
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```

✅***Which is similar to our output.***

And for the question 5 in sheet 2:

The expressions: file

```
%{
%}

%option noyywrap

DIGIT [0-9]
LETTER [a-zA-Z]
DIGITS {DIGIT}+
%%
if|else|then {printf("%s\n", yytext);}
"="|"!="|">"|">="|"<"|"<=" {printf("relop\n");}
(({DIGITS})|({DIGITS}"."{DIGITS}))(("E"{DIGITS})?) printf("num\n");
{LETTER}({LETTER}|{DIGIT})* {printf("id\n");}
[ \t\n\r]+
. printf("Not Defined!\n");
%%
```

And when we tested it on the given program test we got that output: executable

Input: `int sum , count , pass , mnt; while (pass !=10){pass = pass + 1 ;}`

```
id
id
Not Defined!
id
Not Defined!
id
Not Defined!
id
Not Defined!
id
Not Defined!
id
Not Defined!
relop
num
Not Defined!
Not Defined!
```

```
id
relop
id
Not Defined!
num
Not Defined!
Not Defined!
```

✅***Which is similar to our output.***
