Programming Languages Translation
Front End Compiler

Name: Mariam Boules **ID:** 4764

Roles:

- Reading of input files for 3 phases
- DFA minimization
- Computing First and Follow sets
- Building Parsing Table
- Building Annotated Parse Tree

Name: Meeral Maged **ID:** 4847

Roles:

- Converting regular expressions to NFA
- Using minimum DFA in lexical analyzer
- Using parse table to parse and generate parse tree
- Adding semantic rules to context free grammar
- Backpatching
- Semantic Errors Detection

Name: Monique Ehab **ID:** 4928

Roles:

- Converting NFA to DFA
- FLEX (Phase 1 Bonus)
- Removing Left Recursion and Left Factoring (Phase 2 Bonus)
- Bytecode generation
- Report finalization

1. Abstract and objective

A Compiler generator is a tool that creates a lexical analyzer, parser, interpreter and generator from some form of formal description of a language. The compiler generator is the starting point for many systems that translate programs into another formalism. For program verification, it can translate programs into verification conditions. For efficient compilation, it can translate programs into intermediate code, which a separate program could use to generate optimized code. The compiler generator can provide compatible compilers on different machines.

2. Literature and review

2.1 Introduction

Compilers and the electronic computer have been in existence for a short period compared to other technologies. Compilers have influenced society dramatically. It is hard to imagine a world without computers. Think of the traffic light that regulates traffic to avoid an accident, think of the satellites that monitor our weather and can predict disasters before they have even happened, think of the interconnectivity and how you can talk to any place in the world in a fraction of a second. Computers make this possible and the compiler's job is to make the programmer's job easier and cleaner. Without compilers and assembler's programmers would be writing in machine code and it is unlikely we would ever be at the stage we are without them. Writing in machine code is a tedious process and something as simple as adding a new instruction may lead to many instruction offsets requiring to be changed manually. This literature review aims to show the history of compilers.

2.2 History of Compilers and Compiler Generators

The word *compiler* is often attributed to Grace Murray Hopper, who visualized the implementation of a high-level language, quite accurately at the time, as a "compilation of a sequence of subroutines from a library." The first efforts at implementing natural-language-like programming languages were done this way.

The first translator-like compilers were written in the late 1950's. Credit is given to FORTRAN as the first successfully compiled language. It took 18 person-years to develop the FORTRAN compiler because the language was being designed at the same time that it was being implemented, and since this was a first all around, the translation process was not well understood.

We can now design and implement compilers faster and better because (1) languages are better understood, (2) tools now exist for some of the phases of a compiler, and (3) data structures and algorithms are well-known for various compiler tasks.

By the late 1950's, computers and computer languages were proliferating. To create a compiler for N languages on M machines would require $N * M$ programs if each were to be created separately. However, if the front-end of a compiler were to translate the source to a common intermediate language, and the back end of a compiler were to translate the source to a common intermediate language, and the back end of a compiler were to translate this intermediate language to the language of the machine, then (ideally) only $N + M$ programs would be needed.

This was the philosophy behind UNCOL Sammet,1969), one of the first attempts at automating the process of compiler creation. UNCOL stood for Universal Computer-Oriented Language and was actually an intermediate language, not a compiler generator.

It proved difficult to design an intermediate language which was able to "capture" the variable aspects of different programming languages and different machines. This difficulty is still an issue in today's compiler tools. Front-end compiler generators began to appear in the early 1960's, and new generators continue to be created today.

Perhaps the best-known compiler tool is YACC, Yet Another Compiler-Compiler, written by Steve Johnson (1975). YACC was first created for the UNIX operating system and is associated with another tool called LEX (Lesk and Schmidt, 1975) which generates a scanner, although it is not necessary to use a scanner generated by LEX to use YACC. Users can write their own scanners. LEX/YACC have been ported to other machines and operating systems.

Compiler tools whose metalanguages include attribute grammars began to appear in the late 1970's and 1980's. GAG (Kastens et al., 1982) are based upon attribute grammar descriptions.

The PQCC (Wulf et al., 1975) project and others have undertaken the difficult task of generating the back end of a compiler.

3. Description of the problem

Compiler tools aid in the generation of compilers by producing some of the phases automatically. Such tools are also called *translator writing systems, compiler compilers,* or *compiler generators.* We will mention it as a *compiler generator* to emphasize that it generates a program or part of a program.

Like compilers themselves, compiler generators are frequently separated into phases. For the front end of a compiler, the phases are often termed lexical analyzer generator, syntax analyzer generator (or parser generator) and semantic analyzer generator.

4. Phase 1: Lexical Analyzer Generator

4.1. Reading The Input File

- **Data Structures Used:**
- An input file class taking in the input file name including punctuation, keywords, regular expressions and regular definition lists.

- **Algorithms and Techniques Used:**
- First of all the read file method reads in the file and splits it by spaces.

- We then check for the '=', ':', '{' or '[' and call the appropriate method accordingly.

- For the regular definition : we search for (r'\w\s*-\s*\w) (ex:a-z or A-Z) that is expanded to all alphabet separated by or (|) and then assigned to the LHS, if not found the RHS is directly assigned to the LHS and appended to the RD list.

- For the regular expression : we check for the presence of any of the regular definitions and it is replaced by its expansion and assigned to the LHS, else the RHS is assigned to the LHS and appended to the RE list.

- For the punctuation and keywords they're just appended to the punctuation and keywords lists.

4.2. Regex to NFA

- **Data Structures Used:**
- Node Class which includes node_id, boolean state -accepting or not- and adjacency list which contains all the adjacent nodes and their corresponding transition functions. The class methods facilitate adding transition functions from that node to others.
- NFA Class contains states represented as nodes as well as symbols, transition functions, starting state and accepting states. The class methods facilitate the conversion to DFA.
- The accepting states in the NFA - as well as the accepting states in the DFA and minimum DFA - are represented in a dictionary where the key is the state id and the value is the label for every final state.

- StateGroup Class contains references to the start and end nodes of each group which will be used later in Thompson’s Algorithm.

- A dictionary to assign priorities for regular expression symbols.

- A stack is used twice. First to convert regular expressions from infix to postfix, and then to process the postfix expressions.

- A list of nodes is used to keep track of all the states present in the transition graph.

● **Algorithms and Techniques Used:**

- Thompson’s Algorithm is used to convert from regex to NFA.

- First of all, the regex is converted from infix to postfix. We reserved a symbol for concatenation (ex: infix expression: ‘ab’ -> infix expression after adding our reserved symbol: ‘a^b’ -> postfix expression: ‘ab^’). Each regex symbol is assigned a priority, then they are added to the stack ensuring that the symbol pushed to the stack is less than the top of the stack. Otherwise the stack is popped and the symbols are added to the output until that condition is satisfied. Other elements in the regular expression that are not regex symbols are added to the output stream. The result of this function in a postfix regular expression.

- The next step is to process this regex. As we go through the expression, elements are now pushed to the stack after converting them to a state group (Method 1 in Thompson’s Algorithm). The state group in this case consists of a start node and an end node with an edge between them with a transition function of value equal to the element in the regular expression. The state group is then pushed to the stack

- If a reserved regex symbol is found, the appropriate number of state groups is popped from the stack to perform the corresponding regex operation according to Methods 2 and 3 in Thompson’s Algorithm. Then the resultant graph is represented as a state group - with references to the start and end node - then it’s pushed to the stack again. The process is repeated till the end of the regular expression. Throughout this process every created node is added to the list of nodes and every deleted one is removed from that list as well. The NFA is now represented as a list of nodes and a StateGroup with start and end nodes where the end node is the only accepting state.

- This algorithm returns an NFA for every label (or regular expression). The next step is merging all those NFAs into one by transitioning from the new start state to all the start states using epsilon.

4.3. NFA to DFA

- **Data Structures Used:**

- Class DFA that contains the start state, accepting states dictionary, symbols list and transition functions list

- **Algorithms and Techniques Used:**

- Subset Construction Algorithm is used to convert NFA to DFA

- The first step is to combine the NFA transitions which is done by combining all accepting states of the same (start state, input).

- The second step is to convert NFA transitions to DFA transitions, by passing through every state and passing every (start state, input), and the output is considered either one of the already defined states or None or a new state that is added.

- The third step is to take the new list of states and combine their accepting states at every input.

- Print the DFA before minimization with the transition table, states (start and final).

4.4. DFA Minimization

- **Data Structures Used:**

- Class DFAmin with states, terminals, start state, transitions and final states.

- Transitions are read in a dictionary in the form (start state , input):(final state).

- Start states defined in a stack.

- Reachable_states as a set.

- Class DisjointSet provides some functions as union of equivalent states, find items and find set.

- **Algorithms and Techniques Used:**

- The first step is to define the states and assign them to self. States as well as assigning values to self.final_states and self.transitions.

- The second step is to define the transitions that have a final state.

- Pass through all combinations of states and determine which are in our set of transitions.

- Pass through the transitions and examine the end state at different inputs and if they give the same output, they are united together.

- Rename states after minimization and append the new final states to the new united states.

- Define the new transitions and the new final states.

4.5. Lexical Analyzer

- **Data Structures Used:**

- Lexical Analyzer class includes the minimized DFA to be used for scanning.

- **Algorithms and Techniques Used:**

- The class is initialized by calling the DFA generator which generates a list of labeled regular expressions from an input file then passes it to the NFA generator. The resultant NFA is converted to a DFA then a minimized DFA.

- Using the minimized DFA, the scanner uses Maximal Munch Algorithm which traverses the DFA and returns the label corresponding to the longest found lexeme.

- The Algorithm continues until the end of file.

4.6. FLEX (Fast Lexical Analyzer Generator) :

- t is a computer program that generates lexical analyzers (also known as "scanners" or "lexers")
- Flex was written in C around 1987 by Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson.
- Format of input file :

definitions
%%
rules

```
%%
user code
```

*The *definitions section* contains declarations of simple *name* definitions to simplify the scanner specification, and declarations of *start conditions*, which are explained in a later section.

Name definitions have the form:

```
name definition
```

*The *rules* section of the flex input contains a series of rules of the form:

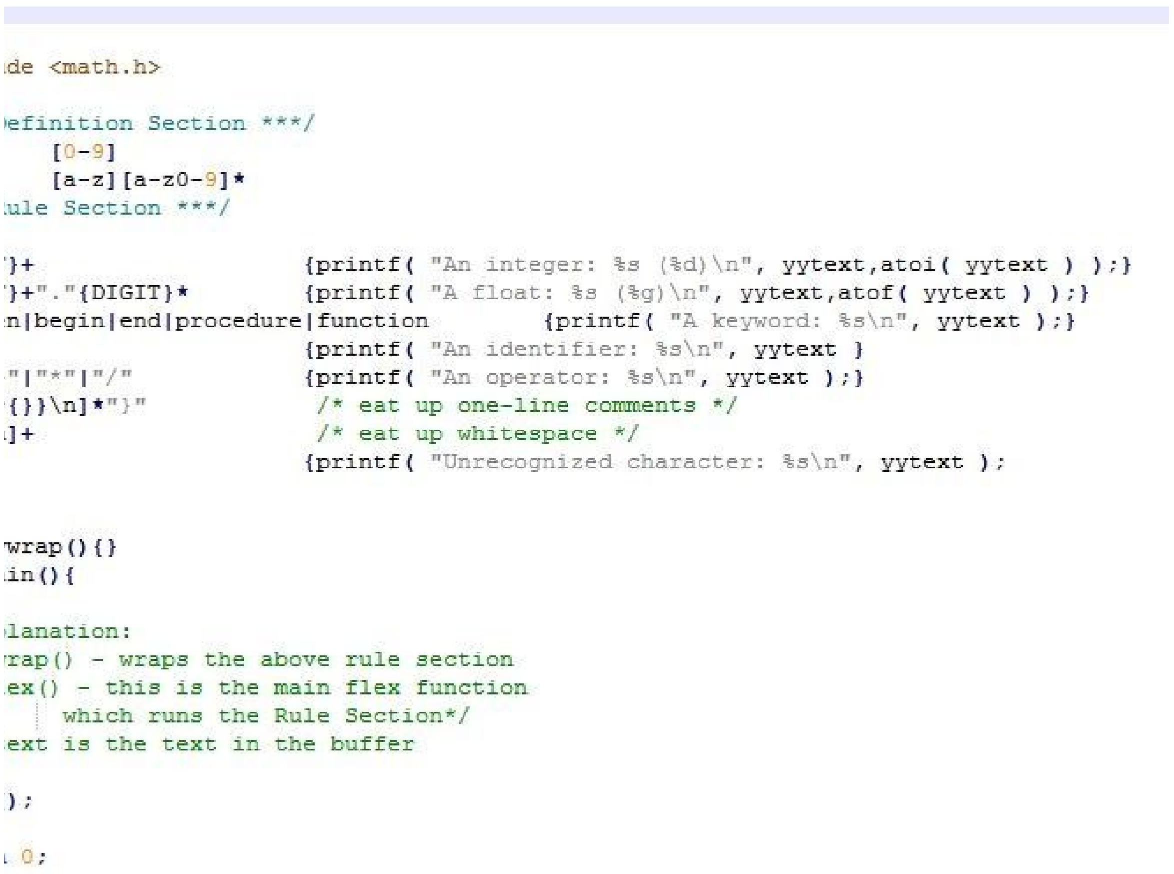
```
pattern   action
```

where the pattern must be unindented and the action must begin on the same line

- Example of format :

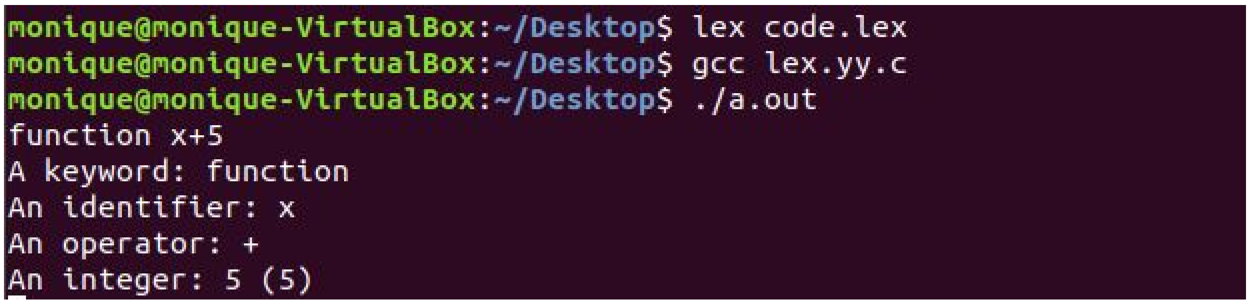
```
%{
/* code block */
%}

/* Definitions Section */
%x STATE_X

%%
   /* Rules Section */
ruleA   /* after regex */ { /* code block */ } /* after code block */
      /* Rules Section (indented) */
<STATE_X>{
ruleC   ECHO;
ruleD   ECHO;
%{
/* code block */
%}
}
%%
/* User Code Section */
```

**Sample input file to flex :



```
de <math.h>

efinition Section ***/
    [0-9]
    [a-z][a-z0-9]*
ule Section ***/

}+                        {printf( "An integer: %s (%d)\n", yytext,atoi( yytext ) );}
}+"."{DIGIT}*             {printf( "A float: %s (%g)\n", yytext,atof( yytext ) );}
n|begin|end|procedure|function          {printf( "A keyword: %s\n", yytext );}
                          {printf( "An identifier: %s\n", yytext }
"|"*"|"/"                 {printf( "An operator: %s\n", yytext );}
{}}\n]*"}"                 /* eat up one-line comments */
]+                         /* eat up whitespace */
                          {printf( "Unrecognized character: %s\n", yytext );


wrap(){}
in(){

lanation:
rap() - wraps the above rule section
ex() - this is the main flex function
    which runs the Rule Section*/
ext is the text in the buffer

);

 0;
```

** OUTPUT :



```
monique@monique-VirtualBox:~/Desktop$ lex code.lex
monique@monique-VirtualBox:~/Desktop$ gcc lex.yy.c
monique@monique-VirtualBox:~/Desktop$ ./a.out
function x+5
A keyword: function
An identifier: x
An operator: +
An integer: 5 (5)
```

5. Phase 2: Parser Generator

5.1. Reading The Input File:

- **Data Structures Used:**

- String class was used that had a list of RHS productions, the LHS non-terminal, and a boolean value whether this LHS non-terminal is a start state or not. LHS class has a name, a list of first and a list of follow.

- **Algorithms and Techniques Used:**
- In save_to_list method, I make sure every line passed is the one which starts with '#' and that the line passed is in the form of a string. This line is then passed to get_production where the LHS is identified and each production is saved in a list then all productions saved in the form of a list of lists called productions which is then passed to the next part.

5.2. Computing First and Follow Sets

- **Data Structures Used:**
- Lists of all LHS objects and LHS names are used to ease dealing with them then 2 dictionaries are used one for the first list and the other for the follow, where the LHS is the key and the first/follow list is the value.
- **Algorithms and Techniques Used:**
- Both the first and follow operations are distributed among 2 methods each. The get_first1 method adds epsilon to the first list if it's found in RHS and adds the first production if it's terminal. The get_first2 method adds the first element of the first list of the first production if it's a non-terminal,adds the first element of the first list of a non-terminal if the previous non-terminal has an epsilon, and adds epsilon to the first list if all productions are non-terminal and has epsilon in their first lists.
- The get_follow1 method accepts the whole list of lines of the cfg, it adds '$' to the start LHS follow set, and adds first lists of following non-terminals in all productions. The get_follow2 method adds the follow sets of the starting non-terminals to final non-terminals or those who are followed by going-to-epsilon non-terminals.

5.3. Building The Parsing Table

- **Data Structures Used:**
- The parsing table is a dictionary whose key is a tuple (the variable and the terminal) and its value is the corresponding rule which the variable goes to.
- The first and follow dictionaries were used in this algorithm as well as the list of grammar rules.

- **Algorithms and Techniques Used:**
- For every variable in the grammar, the parser generator iterates over its rules.
- For every rule, the leftmost symbol is extracted -if it's a terminal- or the First of this symbol if it's a variable and put in a list of terminals.

- If the list of terminals contains epsilon, it is replaced with the Follow of that leftmost symbol.
- For every terminal in this list, the dictionary value that corresponds to the key (variable, terminal) is inspected. If it's already filled then this grammar is ambiguous and the parser terminates. Else, the rule is assigned to this key in the table.
- After that variable is completely processed with all its rules, we extract the Follow terminals of that variable.
- For every key in the parsing table (variable, Follow terminal), if it's empty, it is replaced with the mark 'sync' to be used in panic-mode recovery.

5.4. The Parser

- **Data Structures Used:**

- A stack is used for the variables during parsing.
- A parsing tree to be passed to the intermediate code generator represented as the head treenode
- A treenode which contains a list of children, a list of parents and a label

- **Algorithms and Techniques Used:**

- The lexical analyzer passes the tokens from the input code to the parser. The parser adds a '\$' sign at the end of the list of tokens to mark the end of the code.
- The parser creates a stack and pushes to it a '\$' then the start symbol.
- While the stack isn't empty, the parser pops a symbol from the stack and compares it with the current token.
    - If the symbol is a terminal that is equal to the token then it's a match and the parser continues to process the next token.
    - If the symbol is a terminal that is not equal to the token, an error message is displayed then the symbol is discarded as part of the panic-mode recovery technique.
    - If the symbol is a variable, the parser checks the parsing table for the key (symbol, token). If it is found, the value is pushed to the stack.
    - If the value corresponding to (symbol, token) is 'sync', that means an error occurred. The symbol is discarded as part of the panic-mode error recovery technique after an error message is displayed.
    - If there is no value corresponding to (symbol, token), an error message is displayed and the token is discarded as part of the panic-mode error recovery technique.
    - The parser continues until both the stack and the tokens list only contain the '\$' sign.

- Every time elements are pushed into the stack, they are also added as children to the current treenode in the parse tree, and the node corresponding to the top of the stack is assigned to the current treenode. If a terminal is added to the parse tree, it is added as a tuple of label and value, to be used later in the bytecode generator.

5.5. Left recursion elimination

- **Data Structures Used:**
- Grammar presented as a dictionary

- **Algorithms and Techniques Used:**
- Initialize every non terminal as “A#” where # is a number.
- Loop the grammar and replace every non terminal with its corresponding “A#” leaving the terminals as they are.
- Check for left recursion, if a non terminal is at the left of the recursion then a new non terminal is added (non-terminal_2) applying the rules of LR.
- Then we replace the symbols (A#) with their corresponding non terminals in the grammar.

5.6. Left factoring

- **Data Structures Used:**
- Lists for rules and common strings

- **Algorithms and Techniques Used:**
- Split the production at “=” , then split at “|”
- Group the initials of the rules in common list
- For each of the common strings initialize a new non terminal (non_terminal_2).
- Keep the common part in the original production and for the new production of (non_terminal_2) put the extra parts that are not common

6. Phase 3: Intermediate Code Generator

6.1. Preparing The Input File (Semantic Rules)

- **Technique:**
- The semantic rule for every production rule gives instructions to build the annotated parsing tree. Every semantic rule adds to the parent’s attributes through its children (synthesized attributes). The attributes which each node has are type (int or float) and code. Some methods are called to facilitate the decision making process including type checking and getting the opcode corresponding to certain operations.

6.2. Reading The Input File (Semantic Rules)

- **Data Structures Used:**

- A dictionary whose key is the production rule represented in a tuple of the LHS and the RHS, and its value is the corresponding semantic rule written in Python code.

- **Algorithms and Techniques Used:**

- The LHS and RHS are separated by ':=' token and then the semantic rule is extracted from the RHS by finding the separate '{' then extracting all the coming rule until the '}'

6.3. Building The Annotated Parse Tree

- **Data Structures Used:**

- Annotated parse tree which is referred to using the head tree node.
- The tree node which consists of a list of children, a list of parents, a label, and a type (whether it's a grammar rule or a semantic rule).
- The tree node also includes a list of attributes which is a dictionary of the attribute name and value. This is later used when executing the semantic rules.
- Symbol table which is a dictionary whose key is the variable id and value is a tuple of variable type and variable id in the locals stack.

- **Algorithms and Techniques Used:**

- The parse tree is traversed depth first. For every (parent, children) tuple, the corresponding semantic rule from the semantic rules dictionary is added as a child to that parent.

6.4. Bytecode Generation

- **Data Structures Used:**

- Annotated Parse Tree from previous step

- **Algorithms and Techniques Used:**

- The annotated parse tree is traversed depth first. For each semantic rule found it is executed in Python. By the end of the traversel the head of the annotated parse tree will have an attribute 'code' which contains the bytecode of the whole method body.
- Afterwards, each code line is labeled with its line number. The branching instructions and initially relative to the current instruction. It is then transformed to the actual label using backpatching.

7. Sample Run:

7.1. Input code:

```
int sum;int count;int pass;
int mnt;pass=0;while(pass<10)
{
   pass = pass + 1;
}
```

7.2. Output:

- **Minimum DFA:**
  Found in textfile: ./Ouput/min_dfa0.txt
- **Tokens:**

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

- **Parse Table:**

```
{('METHOD_BODY', 'int'): ['STATEMENT_LIST'], ('METHOD_BODY', 'float'): ['STATEMENT_LIST'], ('METHOD_BODY',
'if'): ['STATEMENT_LIST'], ('METHOD_BODY', 'while'): ['STATEMENT_LIST'], ('METHOD_BODY', 'id'):
['STATEMENT_LIST'], ('METHOD_BODY', '$'): ['sync'], ('STATEMENT_LIST', 'int'): ['STATEMENT',
'STATEMENT_LIST_2'], ('STATEMENT_LIST', 'float'): ['STATEMENT', 'STATEMENT_LIST_2'], ('STATEMENT_LIST',
'if'): ['STATEMENT', 'STATEMENT_LIST_2'], ('STATEMENT_LIST', 'while'): ['STATEMENT', 'STATEMENT_LIST_2'],
('STATEMENT_LIST', 'id'): ['STATEMENT', 'STATEMENT_LIST_2'], ('STATEMENT_LIST', '$'): ['sync'],
('STATEMENT', 'int'): ['DECLARATION'], ('STATEMENT', 'float'): ['DECLARATION'], ('STATEMENT', 'if'):
['IF'], ('STATEMENT', 'while'): ['WHILE'], ('STATEMENT', 'id'): ['ASSIGNMENT'], ('STATEMENT', '}'):
['sync'], ('STATEMENT', '$'): ['sync'], ('DECLARATION', 'int'): ['PRIMITIVE_TYPE', 'id', ';'],
('DECLARATION', 'float'): ['PRIMITIVE_TYPE', 'id', ';'], ('DECLARATION', '}'): ['sync'], ('DECLARATION',
'$'): ['sync'], ('PRIMITIVE_TYPE', 'int'): ['int'], ('PRIMITIVE_TYPE', 'float'): ['float'],
('PRIMITIVE_TYPE', 'id'): ['sync'], ('IF', 'if'): ['if', '(', 'EXPRESSION', ')', '{', 'STATEMENT', '}',
'else', '{', 'STATEMENT', '}'], ('IF', '}'): ['sync'], ('IF', '$'): ['sync'], ('WHILE', 'while'):
['while', '(', 'EXPRESSION', ')', '{', 'STATEMENT', '}'], ('WHILE', '}'): ['sync'], ('WHILE', '$'):
['sync'], ('ASSIGNMENT', 'id'): ['id', 'assign', 'EXPRESSION', ';'], ('ASSIGNMENT', '}'): ['sync'],
('ASSIGNMENT', '$'): ['sync'], ('EXPRESSION', 'id'): ['SIMPLE_EXPRESSION', 'EXPRESSION_2'], ('EXPRESSION',
'num'): ['SIMPLE_EXPRESSION', 'EXPRESSION_2'], ('EXPRESSION', '('): ['SIMPLE_EXPRESSION', 'EXPRESSION_2'],
('EXPRESSION', 'addop'): ['SIMPLE_EXPRESSION', 'EXPRESSION_2'], ('EXPRESSION', ')'): ['sync'],
('EXPRESSION', ';'): ['sync'], ('SIMPLE_EXPRESSION', 'id'): ['TERM', 'SIMPLE_EXPRESSION_2'],
('SIMPLE_EXPRESSION', 'num'): ['TERM', 'SIMPLE_EXPRESSION_2'], ('SIMPLE_EXPRESSION', '('): ['TERM',
'SIMPLE_EXPRESSION_2'], ('SIMPLE_EXPRESSION', 'addop'): ['SIGN', 'TERM', 'SIMPLE_EXPRESSION_2'],
('SIMPLE_EXPRESSION', 'relop'): ['sync'], ('SIMPLE_EXPRESSION', ')'): ['sync'], ('SIMPLE_EXPRESSION',
';'): ['sync'], ('TERM', 'id'): ['FACTOR', 'TERM_2'], ('TERM', 'num'): ['FACTOR', 'TERM_2'], ('TERM',
'('): ['FACTOR', 'TERM_2'], ('TERM', 'addop'): ['sync'], ('TERM', 'relop'): ['sync'], ('TERM', ')'):
['sync'], ('TERM', ';'): ['sync'], ('FACTOR', 'id'): ['id'], ('FACTOR', 'num'): ['num'], ('FACTOR', '('):
['(', 'EXPRESSION', ')'], ('FACTOR', 'mulop'): ['sync'], ('FACTOR', 'addop'): ['sync'], ('FACTOR',
'relop'): ['sync'], ('FACTOR', ')'): ['sync'], ('FACTOR', ';'): ['sync'], ('SIGN', 'addop'): ['addop'],
('EXPRESSION_2', ')'): ['\\L'], ('EXPRESSION_2', ';'): ['\\L'], ('EXPRESSION_2', 'relop'): ['relop',
'SIMPLE_EXPRESSION'], ('STATEMENT_LIST_2', 'int'): ['STATEMENT', 'STATEMENT_LIST_2'], ('STATEMENT_LIST_2',
'float'): ['STATEMENT', 'STATEMENT_LIST_2'], ('STATEMENT_LIST_2', 'if'): ['STATEMENT',
'STATEMENT_LIST_2'], ('STATEMENT_LIST_2', 'while'): ['STATEMENT', 'STATEMENT_LIST_2'],
('STATEMENT_LIST_2', 'id'): ['STATEMENT', 'STATEMENT_LIST_2'], ('STATEMENT_LIST_2', '$'): ['\\L'],
('SIMPLE_EXPRESSION_2', 'addop'): ['addop', 'TERM', 'SIMPLE_EXPRESSION_2'], ('SIMPLE_EXPRESSION_2',
'relop'): ['\\L'], ('SIMPLE_EXPRESSION_2', ')'): ['\\L'], ('SIMPLE_EXPRESSION_2', ';'): ['\\L'],
('TERM_2', 'mulop'): ['mulop', 'FACTOR', 'TERM_2'], ('TERM_2', 'addop'): ['\\L'], ('TERM_2', 'relop'):
['\\L'], ('TERM_2', ')'): ['\\L'], ('TERM_2', ';'): ['\\L']}
```

- **Parse Tree (DFA Traversal)**

```
METHOD_BODY
STATEMENT_LIST
STATEMENT
DECLARATION
PRIMITIVE_TYPE
('int', 'int')
('id', 'sum')
(';', ';')
STATEMENT_LIST_2
STATEMENT
DECLARATION
PRIMITIVE_TYPE
('int', 'int')
('id', 'count')
(';', ';')
STATEMENT_LIST_2
STATEMENT
DECLARATION
PRIMITIVE_TYPE
('int', 'int')
('id', 'pass')
(';', ';')
STATEMENT_LIST_2
STATEMENT
DECLARATION
PRIMITIVE_TYPE
('int', 'int')
('id', 'mnt')
(';', ';')
STATEMENT_LIST_2
STATEMENT
ASSIGNMENT
('id', 'pass')
('assign', '=')
EXPRESSION
SIMPLE_EXPRESSION
TERM
FACTOR
('num', '0')
TERM_2
SIMPLE_EXPRESSION_2
EXPRESSION_2
(';', ';')
STATEMENT_LIST_2
STATEMENT
WHILE
('while', 'while')
```

```
('(', '(')
EXPRESSION
SIMPLE_EXPRESSION
TERM
FACTOR
('id', 'pass')
TERM_2
SIMPLE_EXPRESSION_2
EXPRESSION_2
('relop', '<')
SIMPLE_EXPRESSION
TERM
FACTOR
('num', '10')
TERM_2
SIMPLE_EXPRESSION_2
(')', ')')
('{', '{')
STATEMENT
ASSIGNMENT
('id', 'pass')
('assign', '=')
EXPRESSION
SIMPLE_EXPRESSION
TERM
FACTOR
('id', 'pass')
TERM_2
SIMPLE_EXPRESSION_2
('addop', '+')
TERM
FACTOR
('num', '1')
TERM_2
SIMPLE_EXPRESSION_2
EXPRESSION_2
(';', ';')
('}', '}')
STATEMENT_LIST_2
```

- **Parsing Stack Output:**

```
['$', 'METHOD_BODY']
['$', 'STATEMENT_LIST']
['$', 'STATEMENT_LIST_2', 'STATEMENT']
['$', 'STATEMENT_LIST_2', 'DECLARATION']
['$', 'STATEMENT_LIST_2', ';', 'id', 'PRIMITIVE_TYPE']
['$', 'STATEMENT_LIST_2', ';', 'id', 'int']
['$', 'STATEMENT_LIST_2', ';', 'id']
['$', 'STATEMENT_LIST_2', ';']
['$', 'STATEMENT_LIST_2']
['$', 'STATEMENT_LIST_2', 'STATEMENT']
['$', 'STATEMENT_LIST_2', 'DECLARATION']
['$', 'STATEMENT_LIST_2', ';', 'id', 'PRIMITIVE_TYPE']
['$', 'STATEMENT_LIST_2', ';', 'id', 'int']
['$', 'STATEMENT_LIST_2', ';', 'id']
['$', 'STATEMENT_LIST_2', ';']
['$', 'STATEMENT_LIST_2']
['$', 'STATEMENT_LIST_2', 'STATEMENT']
['$', 'STATEMENT_LIST_2', 'DECLARATION']
['$', 'STATEMENT_LIST_2', ';', 'id', 'PRIMITIVE_TYPE']
['$', 'STATEMENT_LIST_2', ';', 'id', 'int']
['$', 'STATEMENT_LIST_2', ';', 'id']
['$', 'STATEMENT_LIST_2', ';']
['$', 'STATEMENT_LIST_2']
['$', 'STATEMENT_LIST_2', 'STATEMENT']
['$', 'STATEMENT_LIST_2', 'DECLARATION']
```

```
['$', 'STATEMENT_LIST_2', ';', 'id', 'PRIMITIVE_TYPE']
['$', 'STATEMENT_LIST_2', ';', 'id', 'int']
['$', 'STATEMENT_LIST_2', ';', 'id']
['$', 'STATEMENT_LIST_2', ';']
['$', 'STATEMENT_LIST_2']
['$', 'STATEMENT_LIST_2', 'STATEMENT']
['$', 'STATEMENT_LIST_2', 'ASSIGNMENT']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION', 'assign', 'id']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION', 'assign']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'FACTOR']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'num']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2']
['$', 'STATEMENT_LIST_2', ';', 'EXPRESSION_2']
['$', 'STATEMENT_LIST_2', ';']
['$', 'STATEMENT_LIST_2']
['$', 'STATEMENT_LIST_2', 'STATEMENT']
['$', 'STATEMENT_LIST_2', 'WHILE']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION', '(', 'while']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION', '(']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION_2', 'SIMPLE_EXPRESSION']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2',
'FACTOR']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2',
'id']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'EXPRESSION_2']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'SIMPLE_EXPRESSION', 'relop']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'SIMPLE_EXPRESSION']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'SIMPLE_EXPRESSION_2', 'TERM']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'FACTOR']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'num']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'SIMPLE_EXPRESSION_2', 'TERM_2']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')', 'SIMPLE_EXPRESSION_2']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{', ')']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT', '{']
['$', 'STATEMENT_LIST_2', '}', 'STATEMENT']
['$', 'STATEMENT_LIST_2', '}', 'ASSIGNMENT']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION', 'assign', 'id']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION', 'assign']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'FACTOR']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'id']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM', 'addop']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'FACTOR']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2', 'num']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2', 'TERM_2']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2', 'SIMPLE_EXPRESSION_2']
['$', 'STATEMENT_LIST_2', '}', ';', 'EXPRESSION_2']
['$', 'STATEMENT_LIST_2', '}', ';']
['$', 'STATEMENT_LIST_2', '}']
['$', 'STATEMENT_LIST_2']
['$']
Success
```

- **Bytecode Output:**

```
  .limit stack 2
  .limit locals 5
1: ldc 0
2: istore_3
3: iload_3
4: ldc 10
5: if_icmplt 7
6: goto 12
7: iload_3
8: ldc 1
9: iadd
10:istore_3
11:goto 3
12:return
```

8. Conclusion:

To conclude, our front-end compiler generator takes lexical rules as input, from which it generates an NFA, a DFA, then a minimized DFA. This DFA is used to convert the input file into a stream of tokens This part of the compiler is responsible for syntax errors. The parser takes a context free grammar as an input, and uses them to compute first and follow sets which are then used to build a parse table. This table along with the stream of tokens together create a parse tree. The semantic rules are used to annotate that parse tree, through which bytecode is generated. Throughout this project, we have learned so much about how compilers work. We definitely appreciate them more now that we understand what’s happening under the hood.