

Compilers

Phase 1

Lexical Analyzer Generator

Febronia Ashraf Tawfik (31)

Menna Tallah Mohamed Ahmed Osman (55)

Merna Mustafa El-Refaie (57)

Neveen Samir Nagy (61)

Overview

It is required to design and implement a lexical analyzer generator tool.

The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens. The tool is required to construct a nondeterministic finite automata (NFA) for the given regular expressions, combine these NFAs together with a new starting state, convert the resulting NFA to a DFA, minimize it and emit the transition table for the reduced DFA together with a lexical analyzer program that simulates the resulting DFA machine.

The generated lexical analyzer 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 lexical analyzer generator is required to be tested using the given lexical rules of tokens of a small subset of Java. Use the given simple program to test the generated lexical analyzer.

Used data structures

- **Struct**

struct transition → We use a struct whose name is transition which consists of two integers, the number of the state (from) and the number of the state (to), and the symbol of the transition between those two states.

struct DFAelement → is used in functions that convert NFA to DFA.

- Vector

vector<transition> transitions → of struct transition which is used to store all the possible transitions in the finite automata.

vector<int> final_To_DFA → store all the final states of the nondeterministic finite automata.

vector<int> vertices → store the states of graphs of finite Automata.

vector<FA> all → store all the nondeterministic finite automata that will be union.

- Stack

stack<char> operators → is used to store the operators that will be performed between these NFAs.

stack<FA> finite_automata → that is used when the input file is read to store all the possible NFA.

- Map

map<string, FA> regular_Definitions → is used to store the name of the regular definitions and their NFA as the value in the map.

map<string, FA> regular_Expression → is used to store the name of the regular expressions and their NFA as the value in the map.

map<string, FA> key_words → is used to store the name of the keywords and their NFA as the value in the map.

map<char, FA> punctuations → is used to store the name of the punctuations and their NFA as the value in the map.

map<int,string> nfa → is used to store all the nfa of regular expressions, keywords and punctuations.

map<int,string> finalStatesMap → is used to store all the dfa of regular expressions, keywords and punctuations.

`map<int,vector<string>> Mini` → is used to store all the minimal dfa of regular expressions, keywords and punctuations.

Explanation of all algorithms and techniques used

- **Void read_inputFile(const char* input_file):**

First this function is used to read input files that include language, and fill a map of regular definitions, regular expression, keywords and punctuations.

- **FA language() :**

This function is called after that to union all the NFA of language, union keywords, punctuations and regular expressions.

- **FA NFA to DFA(FA a):**

After calling the above functions, this function is called to convert NFA to DFA depending on the concept of the Thompson algorithm.

- **FA minimizedTable (map<int, vector<int>> partitions,FA DFA):**

After converting NFA to DFA, this function is called to minimize the resulting DFA depending on the concept of subset construction algorithm.

- **read_testProgram(const char* input_file,FA mini):**

Finally this function is called to read a test program file to check every word in it and get the corresponding regular expression, keywords, punctuations to it in language, if not found then this means that the input word is incorrect not included in language, and construct output file.

Explanation of all Functions used

- **Class FA** → that is used to construct the finite automata, the deterministic and the nondeterministic, it holds the number of the states in the FA, the start state, final state

and a vector of the final state if the finite automata has multiple final states, the functions that are used in this class are set and get calls as to set the number of the states and get it if is needed, one of its functions is set transition which set all the transitions in the finite automata in vector of struct.

- **Function concatenation** → that takes two FA (class) and concatenate them by build another FA that is the result, its number of the states is the sum of the number of states in the two FA and its start state is the start state of one of the two FA and its final state is the final state of the second FA and the final state of the first FA will be the start state of the second FA.
- **Function kleene closure** → it takes FA as input and build the result FA which their number of states equal the number of the states of input FA plus two, the start state of the result goes to start state of the input FA with absalon transition and the same for final state as the final state of the input FA goes to final state of the result with absalon transition and the final state of the input FA goes to start state of the input FA by absalon transition and the same with start state of the result as goes to final state of the result by absalon transition.
- **Function positive closure** → the same as kleene closure but without the transition between start state and final state of the result FA.
- **Function union** → as takes a vector of FAs that will be union together, build a result FA that has a number of states equal the sum of all number of states of input FAs plus 2 as the start state of result FA will go to all start states of the input FAs by absalon transition and all the final states of the input FAs will go to result's final state by absalon transition.
- **Function keywords** → it builds the NFA of the input keywords, it takes a string as input which consists of keywords and builds the NFA to them, then puts the result in its map.
- **Function punctuations** → it takes a string as input which consists of symbols and builds the NFAs to them then puts the result in its map.
- **Function return FA** → takes string as input and checks if it has a NFA built before, if exists then returns its NFA if not then builds its NFA and returns it.
- **Function range** → when the input expression has (-) operator which is considered a range between two values so, this function takes all possible values between these two values and builds the FA for all these values then union them together.
- **Function expression to NFA** → it is used to build the FA for regular expressions and regular definitions, it takes string -the name of the regular- and its expression, a for loop

is used to check every character in expression string, if it is a character then it is kept in a string, if it is a space then calculate the NFA for the string before it, if it is a special character then it checked, if it is (\) operators then takes the next character whatever it is type, if it is (or(l) range(-) (l concatenation) then push it to operators stack, if it is (l) operator then execute all the operations that are inside this operators until we arrive to (operators, after that if there were operators that haven't been executed yet, then it is executed then push the final NFA to its corresponding map depending on the third input string that determines the type of the input string.

- **Function read file** → reads the input file and call functions keywords, punctuation, expression to NFA depending on input line in the file.
- **Function language** → it combines all NFAs of regular expressions, keywords and punctuations in one map then calls function union NFA that union them.
- **Function union NFA** → it takes a vector of all NFA that are built during reading the input file and union it by make the result FA's start state goes to all start state of input FAs by absalon transition, and result FA has a vector of final states that are all final states of the input FAs.
- **Function getNonFinalStates** → it takes a vector of all vertices as first parameter and another vector of final states as second parameter and just take the difference of both to get non final states
- **Function getInputs** → it takes one parameter which is all the transitions of DFA and gets all inputs that make these transitions.
- **Function NFAtoDFA** → it takes NFA and transforms it to DFA. It gets s0 closure and makes it the start state then gets the next state and its equivalent for every input symbol. Before adding any state to the table it checks if this state is added before or not and then checks if it's a final state or not.
- **Function findNextStats** → it takes four parameters ,integer represent current vertex, vector of all the transitions of DFA, vector of all the inputs and map contains vertex as key and its partition as value. For each input we check every transition if source (vertex_from) of the transition equals the current vertex and input that make a transition equals input of vector then we store the destination (vertex_to) in string which is the output of the function.
- **Function updateMapValues** → it takes three parameters integer which represents the new value of partition name , vector contains vertices of the new partition and map

contains vertex as key and its partition as value . For each vertex in the vector we change the value of the map with key equals that vertex.

- **Function minimization**→ it takes one parameter class DFA and use the last four Function above
 1. it uses updateMapValues to identify the final and non final states partition name then store them in a map called keyOfStates which its key is vertex and its value is partition name
 2. Add the final and non final states in another map called partitions which its key is partition name and its value is vector of all vertices in each the vector with key equal zero used to know which partitions must be split.
 3. Another vector called mySS contains partition names that must be checked for minimization.
 4. For each vertex in vector currntVector - at first contains non final states- we get its next states using findNextStats function and use the output string as key in pre_result map this allow us to know which vertices have the same next states for minimization because all vertices with the same next states are in the vector represents the value of pre_result map with next states as key
 5. When we check all vertices in currntVector if pre_result have more than one elements that means that currntVector must be split because it has states with different next states
 6. Deleting currntVector name from mySS and check if it is not empty ,change currntVector to the next partition and return to point 4
 7. If mySS is empty and partition vector value for key zero is not empty ,delete all partitions with names exist in the vector of key zero and use updateMapValues function to change vertices partition names to the new names of new partitions the put all partition names again in mySS finally ,change currntVector to the next partition and return to point 4
 8. if mySS and partition vector value for key zero are empty stop looping and return partition map as output for the function
- **Function minimizedTable**→ it takes output of minimization function and DFA class. this function makes class with new vertices , transitions and final states after minimization and partitioning

- **Function read test Program**→ it is used to read the test program, as it divides it into its words then sends this word to another function to check if it is included in language or not and convert it into its corresponding regular expression, keywords or punctuation.
- **Function construct output**→ it is called by function read test program. it takes two inputs a word and the minimal finite automata. It walked through the transitions of the FA and checked every character of this word with the symbol of the transitions. Finally, it checks if the states we hold are final and using priority rule, it chooses the corresponding word.

The resultant transition table for the minimal DFA

As the inputs may be quiet big so output of minimization could not be represented in a table so we show only transition like that

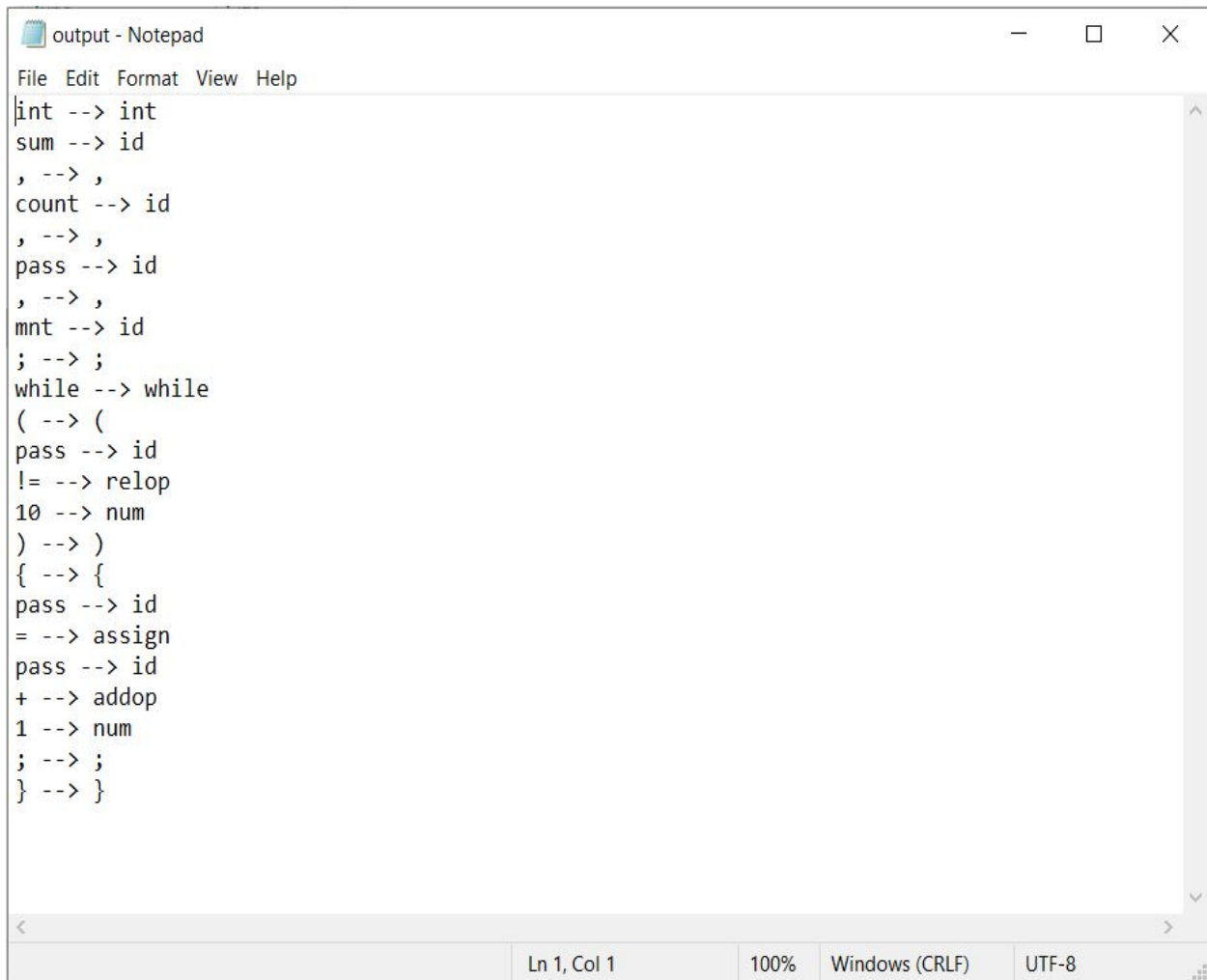
(Source vertex) → (destination vertex) : Symbol is (input for this transition). For simple example with inputs 0,1:

```

q0 --> q1 : Symbol is 0
q0 --> q8 : Symbol is 1
q1 --> q2 : Symbol is 0
q1 --> q3 : Symbol is 1
q2 --> q4 : Symbol is 0
q2 --> q2 : Symbol is 1
q3 --> q5 : Symbol is 0
q3 --> q2 : Symbol is 1
q4 --> q8 : Symbol is 0
q4 --> q3 : Symbol is 1
q5 --> q7 : Symbol is 0
q5 --> q3 : Symbol is 1
q7 --> q4 : Symbol is 0
q7 --> q3 : Symbol is 1
q8 --> q8 : Symbol is 0
q8 --> q8 : Symbol is 1

```


The resultant stream of tokens for the example test program



```
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
; --> ;
} --> }
```

Assumptions

Regular expression : represent epsilon symbol as “^” in the map instead of “L” to avoid confusion with letter “L”

Minimization : every kind of final state joins in one partition at the first step in partitioning .

Test file : all words in the same line are separated from the next one with space.

Bonus Task (FLEX)

FLEX (fast lexical analyzer generator) is a tool for programming that recognizes lexical patterns in the input with the help of flex specifications. An input file describes the lexical analyzer to be generated (named with the extension .l) is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c. And the C compiler compiles the lex.yy.c file into an executable file (output file) called a.exe that takes a stream of input characters and produces a stream of tokens.

Structure of a Flex program (input file) :

Definitions

%%

Rules

%%

User code section

1- Definition Section

It contains the declaration of variables and regular definitions.

2- Rules Section

It contains the pattern and corresponding action. The pattern part contains a regular expression of the lexical analyzer and the action part is a C code, which will be executed when a pattern matches with the input. The rule section is enclosed in "%% %%".

3- User Code Section

It contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer. Here we call The function `yylex()` which is the main flex function that runs the Rule Section

Example

```
digit  [0-9]
digits [0-9]+
letter [a-zA-Z]
%%
"boolean"|"int"|"float"|"if"|"else"|"while"      {printf("KeyWord\n");}
{digit}+|{digit}+\.?{digits}([E]{digits})?        {printf("Number\n");}
{letter}({digit}|{letter})*                        {printf("ID\n");}
";"|","|"|"("|"{"|"}|")"                        {printf("Punctuation\n");}
"=="|"!="|">"|">="|"<"|"<="                        {printf("Relation operation\n");}
"="                                                {printf("Assign\n");}
"+"|"-"                                            {printf("Addition operation\n");}
"*"|"/"                                            {printf("Multiplication operation\n");}
%%
int yywrap(){return 1;}
int main(){
printf("Enter your input\n");
yylex();
return 0;
}
```

```
C:\Users\BLU-Ray>cd Desktop\flex_folder
C:\Users\BLU-Ray\Desktop\flex_folder>flex flex_program.1
C:\Users\BLU-Ray\Desktop\flex_folder>gcc lex.yy.c
C:\Users\BLU-Ray\Desktop\flex_folder>a.exe
Enter your input
sum
ID

int
Keyword

15
Number

15.15
Number

+
Addition operation

/
Multiplication operation

=
Assign

<
Relation operation
```