Team

Bishoy Nader Fathy Beshara	22
Amr Mohamed Nasr Eldine Elsayed	45
Marc Magdi Kamel Ayoub	52
Michael Raafat Mikhail Zaki	54

Compiler Phase I : Lexical Analyzer

24th March 2018

USED DATA STRUCTURES

- **We defined the following Model Classes:**
    1. Node.h : represents the nfa node in the NFA state machine and contains a list of edges that the state has.
    2. Edge.h : represents the transitions in NFA node where each edge represents a range of input letters that on given to the Node transfers it to another Node.
    3. Nfa.h : is the wrapper of a single Nfa with pointers to its start and end states.
    4. NfaToken.h : represents a top token in an Nfa with all its minitokens.
    5. DfaNode.h : represents the node in the DFA state machine which is a combination of one or more NFA nodes and has some extra functions like transferring to only one other DfaNode on a specific applied input.
    6. DfaEdge.h : represents the connection between DfaNodes and has more functions than normal edge like the ability to be splitted into two edges
    7. DfaNodeWrapper.h : container for the dfa node that contains the nfa edges and nafa nodes that are represented by this dfa node and it is used during conversion from NFA to DFA.
    8. PartitionSet.h : represents a set of dfa nodes and is used during minimizing the dfa to minimum number of DFA states.
    9. TransitionTable.h : represents the transition table container which contains minimum number of dfa states and transitions of each state on each input.
    10. Token.h : is the token returned by the tokenizer which contains the type of the token and the lexeme value;

- **We defined the following helping enums:**
    1. MiniType: either OPERATION, CHAR_GROUP, WORD, EPSILON, PARENTHESES.
    2. TokenType: either PUNCTUATION, KEYWORD, REGULAR_EXPRESSION.

- **We used the following data structures:**
    1. Stack: We used it in the NFA to DFA converter to get the epslon closure of a set of Nodes, We used it to move in a DFS path till the end of epslon transitions.
    2. Queue: We used it in minimizer and NFA to DFA converter to move between states in BFS path to collect nodes together.
    3. Set: Used in many modules of the project as collection of related nodes also used as visited flags for nodes.
    4. Map: used for accessing nodes by their names efficiently in many modules of the project.
    5. Vector <Vector<> > used in the transition table to map each state with the other state that it transfers to it on each input.

ALGORITHMS AND TECHNIQUES USED

1. REGEX for grammar parsing

We used regex to read grammar file and match it with the grammar rules to match language rules and keywords and construct separate NFAs for this grammar. However, in case of Regular definition and expressions, our own parsing functions were implemented to separate it into explicit infix format then transform it to postfix to make creation of nfa easier.

2. Graph traversal algorithms

We used various graph traversal algorithms as DFS and BFS in constructing the NFA and converting it to DFA and getting nodes from DFA graph in a list to make things ready for minimization.

3. Using sorted edges to decrease minimization order

We sorted the edges in each DFA node according to the input range in its of them this reduced the order of edge comparison from O(N^2) to O(N).

4. Combining input characters

We build the transition table firstly each input character separated then combining all input characters that lead to same states for every state. This made building the table in O(Number of states).

5. Dual writing formats

We write the transition table into intermediate files in two different formats one of them is human readable and the other is made only to be easily read from the file by the analyzer.

6. Backtracking in Tokenizer

We used Backtracking technique in Tokenizer that is the tokenizer starts moving with the input in the graph until reaching an invalid state or the input ends then it backtracks until reaching the last acceptance state according to Maximal Munch.

INPUT GRAMMER FILE

This is the example grammar file in the project pdf

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: * | /

THE RESULTANT TRANSITION TABLE FOR THE MINIMAL DFA

Fully spreadsheet is found here for the resultant transition table:

https://docs.google.com/spreadsheets/d/1l55ldCE-sXushCMeCztCe_kkqaiwBOaBKNgSWLk_6a0/edit?usp=sharing

TEST PROGRAM

This is the test program in the project pdf

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

THE RESULTANT STREAM OF TOKENS FOR THE TEST PROGRAM

int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)

{
id
assign
id
addop
Num
;
}

ASSUMPTIONS AND JUSTIFICATIONS

1. Priorities of Accepted States

We assumed that accepted states have the following priorities from high to low (punctuation >> keywords >> regular expression) which is followed by most of the popular programming languages.

2. Intermediate Transition Table

We divided the phase into 2 separate programs (Generator and Analyzer) where the generator takes a grammar file as an input and builds from it a minimized transition table. This table is then written to files in two formats, one of them is human readable and the other is an easy format for reading it again. The Analyzer takes this loaded transition table and rebuilds the state machine to analyze a given program into tokens where each token contains the token class (id, num, if, else, ... etc) and the lexeme (x, y, name. ... etc) and outputs the list of tokens representing the input file. This assumption is justifiable as building transition table is a time consuming process and shouldn't be done on every new input program.

3. All Reserved character must be escaped

Any reserved character should always be escaped before usage, otherwise it would to considering it as an operation if suitable, or will declare an error.

Reserved characters are { '+', '-', '*', '|', '(', ')' }

4. Grouping Characters

Characters are grouped by their numerical values, so saying ‘a-z’, would lead to considering any character between the numeric values of a to z to be included. That would lead to us rejecting any character group where the ending character has a numeric value less than the starting character.

5. Regular Expressions Definition

Regular definitions can be used in multiple regular expressions or definitions, however regular expressions can’t be reused to define another regular expression or definition. Also, a regular definition will only be replaced if it was already defined before being referenced, or it will consider it a normal word.

6. Same Token Name

It is considered the responsibility of the user to prevent such case as it will lead to ambiguous and unaccepted generation.

BONUS

Lex, Yacc and Flex generate a lexical analyzer for a given regular expressions, we have three examples for using Flex.

There are three main steps:

1. Creating *.l file that contains the language grammar to be used by Lex.
2. Compile the *.l file to a c program using Lex using using the following command:

```
flex grammar_file.l
```

3. Compile to c program to runnable file using the following command:

```
gcc -o target_file lex.yy.c -lfl
```

Example 1:

It detects specific tokens {begin, hello, thanks, end}

The L File:

```
%{
#include <stdio.h>
%}

%%
begin   printf("Started\n");
hello   printf("Hello yourself!\n");
thanks  printf("Your welcome\n");
end     printf("Stopped\n");
%%
```

Running the example:

```
d_Year/2nd_Term/compilers/bonus$ ./exampleA
Hello
Hello
hello
Hello yourself!

begin
Started

thanks
Your welcome

end
Stopped
```

Example 2:

It detects Lowercase, Uppercase words, decimals, parentheses and semicolon.

```
%{
#include <stdio.h>
%}

%%
[a-z]*     printf("Lowercase word\n");
[A-Z]*     printf("Uppercase word\n");
[a-zA-Z]*  printf("Word\n");
[0-9]*[.][0-9]+    printf("Float\n");
[0-9]*     printf("Integer\n");
";"       printf("Semicolon\n");
"("       printf("Open parentheses\n");
")"       printf("Close parentheses\n");
%%
```

Running the example

```
ge/3rd_Year/2nd_Term/compilers/bonus$ ./exampleB
Hello
Word

hello
Lowercase word

HELLO
Uppercase word

123
Integer

123.3
Float

0.3
Float

(
Open parentheses

)
Close parentheses

()
Open parentheses
Close parentheses

Hello WORLD 1;
Word
 Uppercase word
 Integer
Semicolon
```

Example 3:

It detects integer, float numbers, identifiers, some keywords and arithmetic operations. It skips any white spaces.

```
DIGIT     [0-9]
ID        [a-z][a-z0-9]*
%%

{DIGIT}+                {
                printf("An integer: %s (%d)\n", yytext,
                atoi(yytext));
                }

{DIGIT}+"."{DIGIT}*     {
                printf("A float: %s (%g)\n", yytext,
                atof(yytext));
                }

if|then|begin|end|procedure|function          {
                printf("A keyword: %s\n", yytext);
                }

{ID}                    printf("An identifier: %s\n", yytext);

"+"|"-"|"*"|"/" printf("An operator: %s\n", yytext);

"{"[^}\n]*"}"           /* eat up one-line comments */

[ \t\n]+                /* eat up white space */

.                       printf("Unrecognized character: %s\n", yytext);

%%
```

Running the example:

```
ge/3rd_Year/2nd_Term/compilers/bonus$ ./exampleC
123
An integer: 123 (123)
Hello
Unrecognized character: H
An identifier: ello
hello
An identifier: hello
123.3
A float: 123.3 (123.3)
if x then y
A keyword: if
An identifier: x
A keyword: then
An identifier: y
5       + 7
An integer: 5 (5)
An operator: +
An integer: 7 (7)
```