Compilers – Phase 1

Mohamed ibrahim shaapan	55
Amr mohamed fathi	44
Abdel-rahman yasser	38
Mohamed El-sayed helmi	58

Data Structures and Algorithms

Data Structures

Automata	State: Represents single state in NFA machine Machine: Represents single machine in NFA graph Ex char ‘A’ machine or letter(letter|digit) rule
Transition Table	Composite State: Represents single entry state in transition table Transition Table: Table representation of the machine transitions Row: Single row in transition table

Algorithm

Rule Extraction	- Input = language rules file - Output = language rules extracted into classes
NFA Machine Generation	- Input = language rules classes - Output = NFA machine graph Convert rules into postfix expressions and calls appropriate Operator class (and, or, kleen closure, ..) which in turn returns the machine for each rule Then we combine all rules into single NFA graph
DFA Machine Generation	- Input = NFA machine graph - Output = minimized DFA table
Tokenizer	- Input = DFA machine and user program - Interface function = next_token() When next_token() is called

Class Roles

<table>
<tr><td rowspan="3">Lexical Analyzer Generator</td><td>Rule Extraction</td><td>
<table>
<tr><td>Alpha</td><td>Class to contain language alphabet</td></tr>
<tr><td>Key Word Rule</td><td>parsing keyword rule from file</td></tr>
<tr><td>Pattern Processor</td><td>Calls rule extractor on every rule in the input file</td></tr>
<tr><td>Postfix Expression Handler</td><td>Convert rule class into postfix expression</td></tr>
<tr><td>Punctuation Rule</td><td>parsing punctuation rule from file</td></tr>
<tr><td>Regular Rule</td><td>Determine defined expressions then substitutes with them in the regular expression and get the resultant rule</td></tr>
<tr><td>Rule</td><td>Represents data structure containing the rule</td></tr>
<tr><td>Rule Element</td><td>Represent each character in the rule and determines whether alphabet or an operator</td></tr>
<tr><td>Rule Extractor</td><td>Parsing rules by calling corresponding rule handler class (keyword rule, punctuation rule, regular rule)</td></tr>
</table>
</td></tr>
<tr><td>NFA Generator</td><td>
<table>
<tr><td>NFA Machine</td><td>Generate overall NFA graph</td></tr>
<tr><td>NFA Builder</td><td>Generate NFA rule by rule</td></tr>
<tr><td>And Operator</td><td>NFA Graph anding operator</td></tr>
<tr><td>Or Operator</td><td>NFA Graph oring operator</td></tr>
<tr><td>Kleen Closure Operator</td><td>NFA Graph kleen closure operator</td></tr>
<tr><td>Positive Closure Operator</td><td>NFA Graph positive closure operator</td></tr>
<tr><td>Machine</td><td>Represents single NFA machine</td></tr>
<tr><td>State</td><td>Represents single state in NFA graph</td></tr>
</table>
</td></tr>
<tr><td>DFA Generator</td><td>
<table>
<tr><td>DFA Machine</td><td>Generate minimized DFA table</td></tr>
<tr><td>DFA Table Builder</td><td>Convert NFA graph into DFA table</td></tr>
<tr><td>Transition Table</td><td>Data structure to hold DFA table</td></tr>
<tr><td>Composite State</td><td>Represents single entry in transition table</td></tr>
<tr><td>Partition</td><td>Minimizing DFA table</td></tr>
</table>
</td></tr>
<tr><td colspan="2">Lexical Tokenizer</td><td>Function : next_token()
Represents the single interface function for phase 1
Tokenizes user programn</td></tr>
</table>

How To Use Lex/Flex

Steps
• write the lex file using the format % first section % second section % third section The first section contains the header files we want to include to use in lex file. The second section contains the patterns we want to match and their corresponding action, we represent the patterns using regular expressions. The third section contains Main function of lex file which will be called when we call the executable file of lex
• Write in terminal lex lex_file_name (with extension .l) that will result in a file called lex.yy.c • Then compile that c file using the command gcc lex.yy.c -ll that will result in object file called a.out Then we run that object file by calling ./a.out that will require the user to enter inputs to match them then perform the mentioned actions at section 2 in the lex file.

How To Use Lex/Flex

Flex Examples

```
%{
#include<stdio.h>
%}
%%
"if" {printf("Lexeme = if , Type = T_if");}
"else" {printf("Lexeme = else , Type = T_else");}
"while" {printf("Lexeme = while , Type = T_while");}
"boolean" {printf("Lexeme = boolean , Type = T_boolean");}
"int" {printf("Lexeme = int , Type = T_int");}
"float" {printf("Lexeme = flloat , Type = T_float");}

[a-z|A-Z][a-z|A-Z|0-9]* {printf("Identifier");}
[0-9]+|[0-9]+"."[0-9]+(E[0-9]+)?  {printf("number");}

==|!=|>|<=|>=|<  {printf("relop");}
"+"|- {printf("addop");}
"*"|"/" {printf("mulop");}
"=" {printf("assign");}

";" {printf(";");}
"," {printf(",");}
"(" {printf("(");}
")" {printf(")");}
"}" {printf("}");}
"{" {printf("{");}

%%
int yywrap()
{
   return 1;
}
main()
{
   printf("Enter the code you want to analyze\n");
   yylex();
}
```

```
usera@usera-inspiron-3542: ~/EclipceC++/compiler/Flex
usera@usera-inspiron-3542:~/EclipceC++/compiler/Flex$ lex rules.l
usera@usera-inspiron-3542:~/EclipceC++/compiler/Flex$ gcc lex.yy.c -ll
rules.l:33:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 {
 ^
usera@usera-inspiron-3542:~/EclipceC++/compiler/Flex$ sudo ./a.out
[sudo] password for usera:
Enter the code you want to analyze
ident
Identifier
while
Lexeme = while , Type = T_while
if
Lexeme = if , Type = T_if
<
relop
>
relop
;()
;()
```

Token Example

Token example

rules1.txt (~/EclipceC++/compiler) - gedit

```
1 letter = a-d | A-D
2 digit = 0-2
3 id: letter (letter|digit)*
4
```

prog1.txt (~/EclipceC++/compiler) - gedit

```
1 aBCD012bYaa01CD
2 010abbbbbb012cdDDDDDD01
```

tokens.txt (~/EclipceC++/compiler) - gedit

```
1 aBCD012b          id
2 'Y'      Error don not match any rules
3 aa01CD010abbbbbb012cdDDDDDD01    id
```

rules.txt (~/EclipceC++/compiler) - gedit

```
1 letter = a-b | A-b
2 digit = 0 - 1
3 id: letter (letter|digit)*
4 digits = digit+
5 {boolean int float}
6 num: digit+ | digit+ . digits ( \L | E digits)
7 relop: \=\= | !\= | > | >\= | < | <\=
8 assign: =
9 { if else while }
10 [; , \( \) { }]
11 addop: \+ | -
12 mulop: \* | /
13
```

prog.txt (~/EclipceC++/compiler) - ge

```
1 int sum , count , pass ,
2 mnt; while (pass != 10)
3 {
4 pass = pass + 1 ;
5 }
```

```
1 int      T_int
2 ' '      Error don not match any rules
3 's'      Error don not match any rules
4 'u'      Error don not match any rules
5 'm'      Error don not match any rules
6 ' '      Error don not match any rules
7 ,        T_,
8 ' '      Error don not match any rules
9 'c'      Error don not match any rules
10 'o'     Error don not match any rules
11 'u'     Error don not match any rules
12 'n'     Error don not match any rules
13 't'     Error don not match any rules
14 ' '     Error don not match any rules
15 ,       T_,
16 ' '     Error don not match any rules
17 'p'     Error don not match any rules
18 a       id
19 's'     Error don not match any rules
20 's'     Error don not match any rules
21 ' '     Error don not match any rules
22 ,       T_,
23 'm'     Error don not match any rules
24 'n'     Error don not match any rules
25 't'     Error don not match any rules
26 ;       T_;
27 ' '     Error don not match any rules
28 while   T_while
29 ' '     Error don not match any rules
30 (       T_(
31 'p'     Error don not match any rules
32 a       id
33 's'     Error don not match any rules
34 's'     Error don not match any rules
```

Plain Text Tab Width: 8 Ln 9, Col 13 INS