3240 Project Phase 1

Team Alpha

Baris Arin
David Esposito
Farzon Lotfi

Project Initialization
The project gets executed through the creation of a driver object. driver.java takes in either an array of strings from FileHandler.java (where each string respresents a line in the read-in file. driver object creates a map of all the definition names as a key and the corresponding parsetree as the value. This map gets passed along to a function that builds a large tree out of the results starting from the root function that the user will have to define. In C this root function is the main function and all execution Starts from here. To test our project you can use TestMain.java with the first file as the grammar and the second as the input string to test the grammar.

Scanner
Tokens are formed by a finite state machine. Single characters are returned unless the current character is a '\', 'I' or '$'. For the case where the the current character is equal to '\' the next character was returned as well and treated as an escaped character. When the next character was 'I', if the next two characters were "N " then it was returned as a token for the exclusion set. For the final case, defined classes are considered to begin with the character '$'. Defined classes append characters until the until a special character is reached.

Parser
The creation of the Parse Tree involves ParseNode.java, ParseTree.java, Parse3240.java, and Scanner.java The parse tree is constructed using the recursive decent parser define on pages 144-151 of the Louden book. As we parse the grammar we construct a tree out of the tokens generate by the scanner per line. The tokens become nodes in each lines parse tree. We assume that the first token we get from scanner is a definition, so when we are constructing our map of the definitions this value gets stored as a key if it is a defined class (ie has a $ in front of it). The rest of the tokens on that line get stored in a ParseTree Data structure where the levels are determined by the depth of parsing expected per regex operation possibility. When making our Larger NFA we search each parse tree in the map and fill in the corresponding ParseTree definition for each defined class found in the tree.

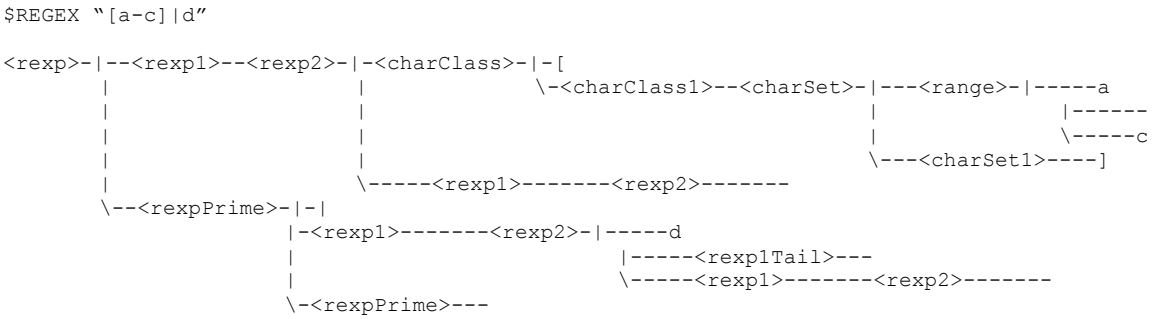

Figure 1: A sample ParseTree

State Table
After constructing a parse tree out of the regex provided, our program passes the parse tree to our State_Table class. The parse tree is essentially our representation of the NFA. With this parse tree, we create a DFA by converting it into a state table, represented by our State_Table

class. We first traverse the parse tree by starting at the root node and recursively working our way down the tree. We add a new row to the State table for each <rexp1> we encounter. The following figure puts the structure of our implementation into perspective:

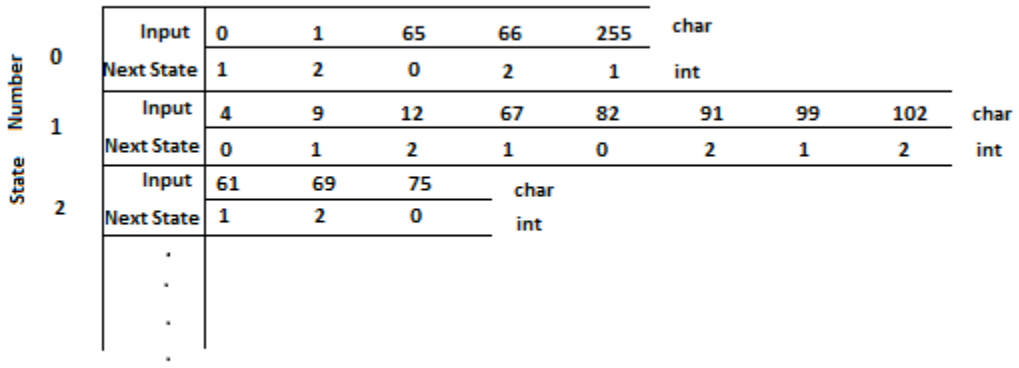

Figure 2: A sample instance of our state table

Our state table is at heart a Map of Integers to State_Rows. A State_Row is a map of characters to integers, where the characters are transition conditions and the corresponding integers are the next unique state to transition to. If our input ends on a particular State_Row, then a boolean variable is checked to see if the State_Row is accepting. If it is, a true is returned. If it isn't, a false is returned.

Figure 2 above shows that not every state will map every possible input character, nor will every state necessarily have the same mapping as every other state. This design helps cut down on storage space as we only map transitions we need rather than filling each row with every possible char value and inserting a -1 to denote an invalid next state.

As far as any assumptions go, we made the assumption that the user who is providing the input will also provide the head of the specification file which is essentially where in the input specification our driver will start from. So for instance, for the sample_input_specification.txt given for this assignment, the user also has to enter $CONSTANT as the second parameter when constructing the instance of the Driver class as well as $IDENTIFIER. This was done so as to eliminate any possible ambiguity in the input specification.

All in all, we were able to implement each module for the project successfully. We encountered a few difficulties along the way. For instance, when converting our parse tree into our state table, it was difficult to figure out which state to go to next whenever a * or + was encountered. The issue was that whenever we encountered repetition, we had no way of knowing where the

expression to be repeated started. To remedy this, we used instance variables to help us count these characters and determine which state to go back to.

Here are some of the test cases we ran to demonstrate functionality:

Test Case 1:

```
%%  Definitions for character classes will be contained in this section – examples below
$DIGIT                  [0-9]
$NON-ZERO    [^0]  IN  $DIGIT
$SMALLCASE          [a-z]
$LETTER       [A-Za-z]
$UPPERCASE  [^a-z] IN $LETTER
$PERIOD [\.]
%%  Token definitions will be contained in this section using regexes – examples below
$INTEGER     $NON-ZERO ($DIGIT)*
$PLAIN_DECIMAL     $INTEGER $PERIOD ($DIGIT)+
```

On the following:
3099999904.00000000228
with head:
"$PLAIN_DECIMAL"

Yields this state table:

```
 0   | { 3:1 2:1 1:1 7:1 6:1 5:1 4:1 9:1 8:1 }
 1   | { 3:2 2:2 1:2 0:2 7:2 6:2 5:2 4:2 9:2 8:2 }
 2   | { 3:2 2:2 1:2 0:2 7:2 6:2 5:2 4:2 9:2 8:2 .:3 }
 3   | { 3:4 2:4 1:4 0:4 7:4 6:4 5:4 4:4 9:4 8:4 }
*4   | { 3:4 2:4 1:4 0:4 7:4 6:4 5:4 4:4 9:4 8:4 }
```

* means it is an accept state

Test Case 2:

```
%%  Definitions for character classes will be contained in this section – examples below
$DIGIT                  [0-9]
$NON-ZERO    [^0]  IN  $DIGIT
$SMALLCASE          [a-z]
$LETTER       [A-Za-z]
$UPPERCASE  [^a-z] IN $LETTER
%%  Token definitions will be contained in this section using regexes – examples below
$INTEGER     $NON-ZERO ($DIGIT)*
```

On the following:
900002738684003290

With head:
“$INTEGER”

Yields
```
 0   | { 3:1 2:1 1:1 7:1 6:1 5:1 4:1 9:1 8:1 }
 1   | { 3:2 2:2 1:2 0:2 7:2 6:2 5:2 4:2 9:2 8:2 }
*2   | { 3:2 2:2 1:2 0:2 7:2 6:2 5:2 4:2 9:2 8:2 }
```

Test Case 3:

```
%%  Definitions for character classes will be contained in this section – examples below
$DIGIT                  [0-9]
$NON-ZERO    [^0]  IN  $DIGIT
$LOWERCASE          [a-z]
$LETTER       [A-Za-z]
$UPPERCASE  [^a-z] IN $LETTER
$PERIOD [.]
%%  Token definitions will be contained in this section using regexes – examples below
$SENTENCE_START          $UPPERCASE ($LOWERCASE)*
$SENTENCE_WORD\ ($LOWERCASE)*
$SENTENCE_END    \ ($LOWERCASE)+ $PERIOD
$SENTENCE  $SENTENCE_START ($SENTENCE_WORD)* $SENTENCE_END
```

On
This is a valid sentence.

With head:
“$SENTENCE”

Yields
```
 0   | { D:1 E:1 F:1 G:1 A:1 B:1 C:1 L:1 M:1 N:1 O:1 H:1 I:1 J:1 K:1 U:1 T:1 W:1 V:1 Q:1 P:1 S:1 R:1 Y:1 X:1 Z:1 }
 1   | { f:2 g:2 d:2 e:2 b:2 c:2 a:2 n:2 o:2 l:2 m:2 j:2 k:2 h:2 i:2 w:2 v:2 u:2 t:2 s:2 r:2 q:2 p:2 z:2 y:2 x:2 }
 2   | { f:2 g:2  :3 d:2 e:2 b:2 c:2 a:2 n:2 o:2 l:2 m:2 j:2 k:2 h:2 i:2 w:2 v:2 u:2 t:2 s:2 r:2 q:2 p:2 z:2 y:2 x:2 }
 3   | { f:4 g:4 d:4 e:4 b:4 c:4 a:4 n:4 o:4 l:4 m:4 j:4 k:4 h:4 i:4 w:4 v:4 u:4 t:4 s:4 r:4 q:4 p:4 z:4 y:4 x:4 }
 4   | { f:4 g:4  :3 d:4 e:4 b:4 c:4 a:4 n:4 o:4 l:4 m:4 .:5 j:4 k:4 h:4 i:4 w:4 v:4 u:4 t:4 s:4 r:4 q:4 p:4 z:4 y:4 x:4 }
*5   | { }
```