

Chapter 18

Capstone Project

grep

```
grep bo*t f1.txt
```

Three versions of grep

G1: parser only

G2: parser, code generator that builds NFA

G3: parser, code generator that builds NFA,
pattern matcher

Token manager

`b* \ *`

kind	CHAR	STAR	CHAR	EORE

beginColumn	1	2	3	5

image	"b"	"*"	"*"	"<EORE>"

Grammar for regular expressions

- 1) `expr` \rightarrow `term termList`
- 2) `termList` \rightarrow `"|" term termList`
- 3) `termList` $\rightarrow \lambda$
- 4) `term` \rightarrow `factor factorList`
- 5) `factorList` \rightarrow `factor factorList`
- 6) `factorList` $\rightarrow \lambda$
- 7) `factor` \rightarrow `<CHAR> factorTail`
- 8) `factor` \rightarrow `<PERIOD> factorTail`
- 9) `factor` \rightarrow `"(" expr ")" factorTail`
- 10) `factorTail` \rightarrow `"*" factorTail`
- 11) `factorTail` $\rightarrow \lambda$

Structure of G1

```
1 class G1
2 {
3     public static void main(String[] args)
4     {
5         // Check here if number of args is correct
6
7         G1TokenMgr tm = new G1TokenMgr(args[0]);
8         G1Parser parser = new G1Parser(tm);
9
10        try
11        {
12            // parse regular expression
13            parser.parse();
14        }
15        catch (RuntimeException e)
16        {
17            System.err.println(e.getMessage());
18            System.exit(1);
19        }
20    }
21    //=====
22    interface G1Constants
23    {
24        // see Fig. 18.1
25    }
26    //=====
27    class G1TokenMgr implements G1Constants
28    {
29        // Contains constructor and getNextToken method
30    }
31    //=====
32    class G1Parser implements G1Constants
33    {
34        // Contains constructor, parse, advance, and consume
35        // methods. Also contains the methods for the recursive
36        // descent parser based on the grammar in Fig. 18.3
37    }
```

Represent each state with NFAState

```
1 class NFAState
2 {
3     public NFAState arrow1;
4     public char label1;
5     public NFAState arrow2;          // arrow2 always lambda
6     public NFAState acceptState;
7     //-----
8     public NFAState()
9     {
10         arrow1 = arrow2 = acceptState = null;
11         label1 = 0;                  // zero represents lambda
12     }
13     //-----
14     public static void displayNFA(NFAState startState)
15     {
16         // display NFA
17     }
18 }
```

Parser/translator

```
1  private NFASState expr()
2  {
3      NFASState p;
4
5      p = term();
6      p = termList(p);
7      return p;
8  }
9  //-----
10 private NFASState termList(NFASState p)
11 {
12     NFASState q;
13
14     switch (currentToken.kind)
15     {
16         case OR:
17             consume(OR);
18             q = term();
19             p = cg.make(OR, p, q);
20             p = termList(p);    // pass new NFA to termList
21             break;
22         case RIGHTPAREN:
23         case EORE:
24             ;
25             break;
26         default:
27             throw genEx("\\"|"\", \"\")\", or <EORE>");
28     }
29     return p;
30 }
```


factorTail method

```
1  private NFASState factorTail(NFASState p)
2  {
3      switch(currentToken.kind)
4      {
5          case STAR:
6              consume(STAR);
7              p = cg.make(STAR, p);
8              p = factorTail(p);
9              break;
10         default:
11             ;
12             break;
13     }
14     return p;
15 }
```

make method

```
1  public NFASState make(int op, NFASState p, NFASState q)
2  {
3      // s is new start state; a is new accept state
4      NFASState s, a;
5
6      switch(op)
7      {
8          case OR:
9              s = new NFASState();
10             a = new NFASState();
11             s.arrow1 = p;          // make s point to p and q
12             s.arrow2 = q;
13             // make accept states of p and q NFAs point to a
14             p.acceptState.arrow1 = a;
15             q.acceptState.arrow1 = a;
16             s.acceptState = a;    // make a the accept state
17             return s;
18             case CONCAT:
19                 ...
20             default:
21                 throw new RuntimeException("Bad call of make");
22         }
23     }
```

make method

```
25 public NFASState make(int op, Token t)
26 {
27     // s is new start state; a is new accept state
28     NFASState s, a;
29
30     switch(op)
31     {
32         case CHAR:
33             s = new NFASState();
34             a = new NFASState();
35             s.arrow1 = a;           // make s point to a
36             s.label1 = t.image.charAt(0);
37             s.acceptState = a;     // make a the accept state
38             return s;
39         case PERIOD:
40             ...
41         default:
42             throw new RuntimeException("Bad call of maker");
43     }
44 }
```

lambdaClosure method

Pseudocode for `lambdaClosure()`

```
set gotAccept to false

for each state s in currentStates
{
    if startState.acceptState is s
    then set gotAccept to true

    if s.arrow1 is non-null (i.e., there is an outgoing arrow) and
       s.label1 is 0 (i.e.,  $\lambda$ ) and
       s.arrow1 is not already in currentStates
    then add s.arrow1 to currentStates.

    if s.arrow2 is non-null (i.e., there is an outgoing arrow) and
       s.arrow2 is not already in currentStates
    then add s.arrow2 to currentStates.
}

return gotAccept
```

applyChar method

Pseudo code for `applyChar(char c)`

```
clear nextStates

for each state s in currentStates
{
    if(s.arrow1 is non-null (i.e., there is an outgoing arrow) and
       (s.label1 is PERIOD or s.label1 matches c) )
        then add s.arrow1 to nextStates
}

swap currentStates and nextStates
```

match method

```
1      // process input line in buf
2      for (startIndex = 0; startIndex < buf.length();
3           startIndex++)
4      {
5          currentStates.clear();
6          currentStates.add(startState);
7          bufIndex = startIndex;
8
9          // apply substring starting at bufIndex to
10         // NFA. Exit on an accept, end of substring,
11         // or trap state
12         while (true)
13         {
14             gotAccept = lambdaClosure();
15             if (gotAccept // accept state entered
16                 || bufIndex >= buf.length() // end substring
17                 || currentStates.size() == 0) // trap state
18                 break;
19             applyChar(buf.charAt(bufIndex++));
20         }
21
22         // display line if match occurred somewhere
23         if (gotAccept)
24         {
25             System.out.println(buf);
26             break; // go to next line
27         }
28     } // end of for loop
```