Simple Grep

1 Introduction

For this project you will implement a simple version of the Unix `grep` utility. First your program will parse a regular expression (using recursive descent) specified on the command line and construct an NFA that will be used for subsequent pattern matching. The grammar for regular expressions is given in Section 2. I will provide you with a C++ module for building NFA's and pattern matching as described in Section 3. What to submit is detailed in Section 7.

2 Grammar for Regular Expressions

Interestingly enough, the language of regular expressions is not regular, but it is context-free. Below is an unambiguous grammar for regular expressions we will use in this project:

$$
\begin{array}{rcll}
R & \rightarrow & U & (1)\\
 & \rightarrow & \texttt{\^{}}\ U & (2)\\
 & \rightarrow & U\ \texttt{\$} & (3)\\
 & \rightarrow & \texttt{\^{}}\ U\ \texttt{\$} & (4)\\
U & \rightarrow & U\ \texttt{|}\ C & (5)\\
 & \rightarrow & C & (6)\\
C & \rightarrow & C\ K & (7)\\
 & \rightarrow & K & (8)\\
K & \rightarrow & L\ \texttt{*} & (9)\\
 & \rightarrow & L\ \texttt{+} & (10)\\
 & \rightarrow & L\ \texttt{?} & (11)\\
 & \rightarrow & L & (12)\\
L & \rightarrow & M & (13)\\
 & \rightarrow & \texttt{(}\ U\ \texttt{)} & (14)\\
 & \rightarrow & \texttt{[}\ S\ \texttt{]} & (15)\\
 & \rightarrow & \texttt{[\^{}}\ S\ \texttt{]} & (16)\\
M & \rightarrow & \texttt{LIT} & (17)\\
 & \rightarrow & \texttt{.} & (18)\\
S & \rightarrow & S\ T & (19)\\
 & \rightarrow & T & (20)\\
T & \rightarrow & \texttt{LIT} & (21)\\
 & \rightarrow & \texttt{LIT - LIT} & (22)
\end{array}
$$

The ^ and $ meta-characters are beginning-of-line and end-of-line *anchors* (respectively) when they occur as the first and/or last characters in the regular expression. Productions 5, 7, and 9 represent *union*, *concatenation*, and *Kleene-Closure* operations respectively. *Character classes* are encoded in square brackets as in Productions 15 and 16; The ^ character is used to complement a character class. The user can match any single character (except a \n) with the "wild-card" . in production 18.

3 NFA's

Your parser will construct a NFA that will be used for pattern matching. I will provide you with a C++ NFA class that provides the following methods:

```
class NFA {
  NFA(int c);
  NFA(const CharSet& s);
  ~NFA();
  NFA *clone() const;
  NFA *union_(const NFA *other);
  NFA *concat(const NFA *other);
  NFA *kleene();
  bool accept(const std::string& s) const;
};
```

The implementation I provide uses very simplistic data structures and could be optimized greatly for speed. Once the NFA is constructed, the accept method will be used for pattern matching as described in Section 5.

3.1 NFA primitives

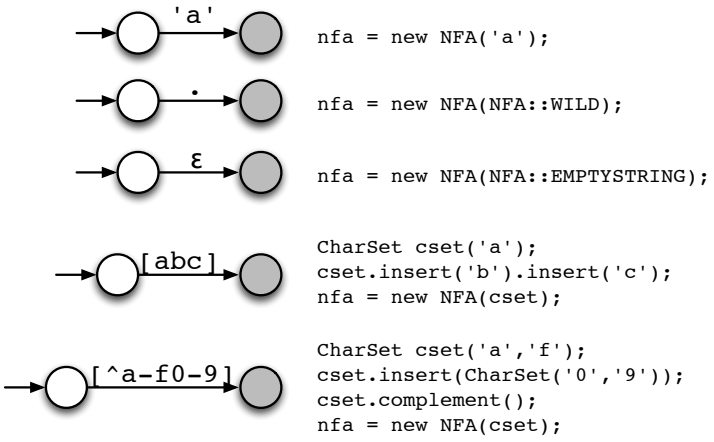

Figure 1: Construction of NFA primitives.

The NFA class provides two constructors for building primitive NFA's as shown in Figure 1. To build an NFA that accepts a single character 'a', use the first constructor with c = 'a'. This constructor can also be used to build an NFA that accepts *any* single character with c = NFA::WILD. One can also build an NFA that accepts the empty-string using c = NFA::EMPTYSTRING. The second constructor is use to create NFA's that accept a set of characters – this is used for *Character Classes* common to most regular expression libraries whose syntax is specified in Productions 15 and 16; The following class encapsulates a character set:

```
class CharSet {
  CharSet(int c);
  CharSet(int a, int b);
  bool contains(int c) const;
  CharSet& insert(int c);
  CharSet& insert(const CharSet& other);
  CharSet& complement();
};
```

The two constructors build `CharSet`'s that contain a single character or a range of characters. More characters can be added to the set via the `insert` methods. The last method computes the complement of the entire set (used for processing Production 16).

3.2 NFA composition

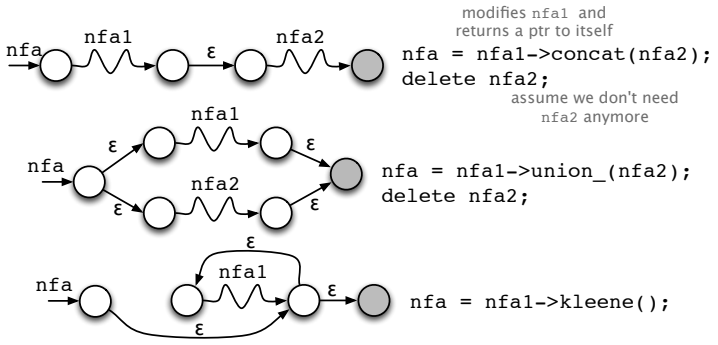

Figure 2: Composition of NFA's for concatenation, union, and Kleene-Closure.

Figure 2 demonstrates how to compose NFA's for the standard regular operations: *concatenation, union,* and *Kleene-Closure.* The `clone` method creates a "deep copy" of the NFA. These methods can also be used to build NFA's for regular expressions $\alpha+$ and $\alpha?$ by noting the following identities:

$$\alpha+ \quad \equiv \quad \alpha\alpha* \tag{23}$$

$$\alpha? \quad \equiv \quad \alpha \cup \epsilon \tag{24}$$

For example, given an `nfa` for the expression α, we can modify it for the pattern $\alpha+$ as follows:

```
NFA *nfaCopy = nfa->clone();
nfa->concat(nfaCopy->kleene());
delete nfaCopy;
```

We can alter the `nfa` for the pattern $\alpha?$ as follows:

```
NFA *e = new NFA(NFA::EMPTYSTRING);
nfa->union_(e);
delete e;
```

3.2.1 Handling anchors

When the user provides a regular α, they expect to match lines that contain a substring that matches the expression. In other words, the final NFA needs to be modified based on which anchors are used (if any):

$$\alpha \quad \Rightarrow \quad .*\alpha.* \tag{25}$$

$$\hat{}\alpha \Rightarrow \alpha.* \quad (26)$$
$$\alpha\$ \Rightarrow .*\alpha \quad (27)$$
$$\hat{}\alpha\$ \Rightarrow \alpha \quad (28)$$

For example, given an `nfa` for the expression α we would modify it for the final expression $\hat{}\alpha$ as follows:

```
NFA *any = (new NFA(NFA::WILD))->kleene();
nfa->concat(any);
delete any;
```

4 Recursive Descent Construction of NFA

It is fairly straight forward to construct a recursive descent parser directly from the grammar in Section 2. The few cases where left-recursion would be removed and factoring would be performed can be trivially handled. For example, the routine corresponding to the start symbol R would invoke `U` and check for anchors at the appropriate points:

```
NFA *R() {
  bool beginAnchor = false;
  if (lookahead == '^') {beginAnchor = true; match('^');}
  NFA *nfa = U();
  bool endAnchor = false;
  if (lookahead == '$') {endAnchor = true; match('$');}
  // modify nfa based on (lack of) anchors
  return nfa;
}
```

4.1 Lexical Analysis

The lexical analyzer should scan the string stored in the first command line argument. If you are using `lex` (or `flex`) [1] then you tell `yylex()` to fetch its input from the `argv[1]` string as follows:

```
yy_scan_string(argv[1]);
```

The number of lexeme patterns needed are small:

```
%%
[-^$|+*?()\.\[\]]  {return yytext[0];}
\\.                {yylval.c = yytext[1]; return LIT;}
.                  {yylval.c = yytext[0]; return LIT;}
%%
```

The first pattern is for matching meta-characters. The second pattern allows for characters to be "escaped" so meta-characters can instead be interpreted by their literal values. Since we are matching a text file line by line, we don't worry about newlines `'\n'` in the input expression.

5 Matching Input Lines

C++ provides standard `string` and `iostream` libraries which we can use for all our I/O needs:

[1] It seems circular to use regular expressions to match lexemes for parsing regular expressions.

```
#include <string>
#include <iostream>
using namespace std;
```

We can read from standard input (input stream `cin`) a line at a time and echo lines to standard output (output stream `cout`) that match the user's regular expression as follows:

```
  string line;
  while (getline(cin, line)) {
    const int n = line.length();
    if (n > 0 && line[n-1] == '\n') // chomp
      line.resize(n-1);
    if (nfa->accept(line))
      cout << line << endl;
  }
```

6 Running your program

Your program should take exactly one command line argument which is the regular expression used to match lines. The program will read/write to/from `stdin/stdout` as described in Section 5.

Starting with the simple input file

```
$ cat test.in
cat
aarvark
dog
baaa
```

we can match using some simple expressions:

```
$ ./sgrep 'aa' < test.in
aarvark
baaa
$ ./sgrep '^aa' < test.in
aarvark
$ ./sgrep 'aa$' < test.in
baaa
```

Test using a large dictionary:

```
$ ./sgrep '[aeiou][aeiou][aeiou][aeiou]' < /usr/share/dict/words
Achromobacterieae
Acrasieae
Aeaean
Aissaoua
amidoguaiacol
Amoebobacterieae
Andreaea
Andreaeaceae
Andreaeales
aqueoigneous
...
$ ./sgrep '^[aeiou][aeiou][aeiou][aeiou]' < /usr/share/dict/words | more
euouae
```

I will provide a larger test suite you can try your program on.