CSCI 493.66 Unix Tools

with emphasis on string algorithms and Bioinformatics

Homework 3
Due 03/12/09

Saad Mneimneh
Computer Science
Hunter College of CUNY

Problem 1: Basic regular expression matcher

In this problem, you are asked to implement a basic, **but efficient**, regular expression matcher. This will follow the approach of building a non-deterministic finite automaton NFA as described in class. The following regular expression features will be supported:

Operands

c	a symbol
!	a wild card symbol
()	a sub regular expression

Operators

|	union of regular expressions
.	concatenation of regular expressions
$*$	zero or more repetition of a regular expression
$+$	one or more repetition of a regular expression
?	zero or one occurrence of a regular expression

The basic building blocks for the NFA are provided to you through the following C++ code (also available in Perl):

```
struct State {
  int c; //256=epsilon, 257=accept
  State * next;

  //used only when c is epsilon, but could be null
  State * alt;

  bool visited;

  State(int c, State * next=0, State * alt=0) {
    this->c=c;
    this->next=next;
    this->alt=alt;
    visited=false;
  }

  bool isAccept()const {
    return c==257;
  }

  bool isSplit()const {
    return c==256;
  }
};
```

```
struct NFA {
  State * in;
  State * out;

  NFA(State * in, State * out) {
    this->in=in;
    this->out=out;
  }
};
```

```
NFA sym(int c) {
  State * s=new State(c);
  return NFA(s, s);
}

NFA bar(NFA nfa1, NFA nfa2) {
  State * s=new State(256, nfa1.in, nfa2.in);
  State * t=new State(256);
  nfa1.out->next=nfa2.out->next=t;
  return NFA(s, t);
}

NFA dot(NFA nfa1, NFA nfa2) {
  nfa1.out->next=nfa2.in;
  return NFA(nfa1.in, nfa2.out);
}

NFA star(NFA nfa) {
  State * s=new State(256, 0, nfa.in);
  nfa.out->next=s;
  return NFA(s, s);
}

NFA plus(NFA nfa) {
  State * s=new State(256, 0, nfa.in);
  nfa.out->next=s;
  return NFA(nfa.in, s);
}
```

```
NFA que(NFA nfa) {
  State * t=new State(256);
  State * s=new State(256, nfa.in, t);
  nfa.out->next=t;
  return NFA(s, t);
}

NFA accept(NFA nfa) {
  State * s=new State(257);
  nfa.out->next=s;
  return NFA(nfa.in, s);
}

void freeNFA(State * s) {
  if (!s || s->visited)
    return;
  s->visited=true;
  freeNFA(s->next);
  freeNFA(s->alt);
  delete s;
}

void freeNFA(NFA nfa) {
  freeNFA(nfa.in);
}
```

One could build an NFA for '$a.(b.b) + .a|c$' as follows (note that . is the concatenation operator here):

```
int main() {
  NFA nfa=accept(
                bar(
                    dot(
                        sym('a'),
                        dot(
                            plus(
                                 dot(
                                     sym('b'),
                                     sym('b')
                                    )
                                ),
                              sym('a')
                             )
                        ),
                    sym('c')
                   )
               );
}
```

Each NFA is determined by its input state and output state. Each state contains a symbol to match, c, and a pointer to the next state, $next$, in case of a match. A state with an ϵ arrow ($c = 256$) has an additional pointer alt to an alternative next state (which makes the state machine non-deterministic). The accept state is characterized by $c = 257$.
The regular expression matcher must perform seven tasks:

- Accept a regular expression like '$a(bb)+a|c$' at the command prompt, and possibly a file name. If no file name is specified, the standard input is considered (this could be either a pipe, a redirection, or the keyboard).
- The regular expression is transformed by adding zero or more repetition of the wild card at the beginning and the end. Therefore, '$a(bb) + a|c$' is transformed into '$!*(a(bb) + a|c)!*$'

- The regular expression is transformed to explicitly contain the concatenation operation. For instance, '$!*(a(bb)+a|c)!*$' is transformed into '$!*.(a.(b.b)+.a|c).!*$'.

- The regular expression is converted from infix notation to postfix notation. Therefore, '$!*.(a.(b.b)+.a|c).!*$' is converted to '$!*abb.+.a.c|.!*.$'. Since most of you have not taken CSCI 235, a code to convert from infix to postfix is provided to you in C++. It is easy to port this code to Perl if needed.

```
#include<vector>
using std::vector;

void in2post(const char * s, char * t) {
  vector<char> v; //empty stack
  int count=0;
  while (*s!='\0') {
    if (*s=='|') {
      while (!v.empty() &&
             v[v.size()-1]!='(') {
        t[count]=v[v.size()-1];
        v.pop_back();
        count++;
      }
      v.push_back(*s);
    }
    else
      if (*s=='.') {
        while(!v.empty() &&
              v[v.size()-1]!='|' &&
              v[v.size()-1]!='(') {
          t[count]=v[v.size()-1];
          v.pop_back();
          count++;
        }
        v.push_back(*s);
      }
      else
        if (*s=='*' ||
            *s=='+' ||
            *s=='?')
          v.push_back(*s);
        else
          if (*s=='(')
            v.push_back(*s);
          else
            if (*s==')') {
                while (v[v.size()-1]!='(') {
                t[count]=v[v.size()-1];
                v.pop_back();
                count++;
              }
              v.pop_back();
            }
            else {
              t[count]=*s;
              count++;
            }
    s++;
  }
  while (!v.empty()) {
    t[count]=v[v.size()-1];
    v.pop_back();
    count++;
  }
  t[count]='\0';
}
```

- The postfix regular expression is used to build an NFA using the following strategy: the regular expression is scanned from left to right. When a symbol is encountered, an NFA corresponding to that symbol is pushed onto the stack. When an operator is encountered, a number of NFAs, consistent with the operator, is popped and used to construct a new NFA using the corresponding operator. That newly constructed NFA is then pushed onto the stack. This is repeated until the expression is exhausted and there is only one NFA in the stack. This NFA is popped and a final NFA is constructed using accept.
- A program to simulate (run) the built NFA on an input string must be implemented (this is probably the hardest part). As discussed in class, this boils down to keeping track of the list of states that are reachable after reading each symbol of the input string.
- Input is read one line at a time, and each line goes through the NFA to determine whether the line contains the given regular expression. If the regular expression is found, the line is sent to the standard output.

Problem 2: Global alignment

Implement the global alignment algorithm based on dynamic programming as explained in class. Your program should accept two strings as input, and output the best alignment based on the following scoring schemes for a match, a mismatch, and a gap.

- $(1, -1, -2)$
- $(1, 0, 0)$, what does that give you?
- $(0, -1, -1)$, what does that give you?
- $(0, -\infty, -1)$, what does that give you?

Which of the last three schemes are related? and how?