Roll Your Own Regular Expression Engine

CS 317

Due October 23, 2006

1 Introduction

For this programming project you will construct a *regular expression engine* in the language of your choice; You will then use this engine to search for patterns in files. The "engine" involves creating *Nondeterministic Finite Automata* (NFA) from user supplied regular expressions and then using these NFA to perform pattern matching. Optionally, the NFA may be optimized (*e.g.,* converted to a minimal DFA) for speedier matching.

2 Regular Expressions

expression	*semantics*
c	single non-metacharacter (metacharacters are "`()|*+?&`")
`.`	wild card (matches *any* single character)
`\d`	digit ('0' . . . '9')
`\D`	non-digit
`\w`	alphanumeric character ('a' . . . 'z', 'A' . . . 'Z', '0' . . . '9')
`\W`	non-alphanumeric character
`\s`	whitespace character (space, tab, . . .)
`\S`	non-whitespace character
`\t`	tab
\metachar	literal meta-character
$r_1 \mid r_2$	union (match either r_1 or r_2)
$r_1\ r_2$	concatenation (match r_1 followed by r_2)
r^*	Kleene closure (match r zero or more times)
r^+	match r one or more times
$r?$	match r zero or one time
(r)	parenthesized expression
`^`	anchor for matching the beginning of a line
`$`	anchor for matching the end of a line

Figure 1: Regular expression syntax and semantics. Operators are listed in increasing order of precedence.

Figure 1 lists the syntax we will use for infix regular expressions. *Metacharacters*, such as parentheses, asterisks, questions marks, etc… are special characters used to create regular expressions. If you wish to match the literal form of a metacharacter, it must be "escaped" using a backslash (*e.g.,* `\?` would match a literal question mark). Certain "wildcards" are used to match any single character from a set of characters (*e.g.,* `\d` is shorthand for `(0|1|2|3|4|5|6|7|8|9)`).

2.1 Postfix Regular Expressions

Most people who use HP calculators are used to postfix notation for arithmetic expressions. For example, the infix expression

```
(3 + (4 * 8)) + ((6 + 7)/5)
```

is expressed as

```
3 4 8 * + 6 7 + 5 / +
```

in postfix notation. In prefix form, this expression would be

```
+ + 3 * 4 8 / + 6 7 5
```

Both prefix and postfix notation are nice because parentheses are unneeded since they do not have the operator-operand ambiguity inherent to infix expressions. Postfix expressions are also easy to parse (which is why we are discussing them here). Later in the course, when we discuss context-free languages, we will revisit the problem of parsing infix expressions.

Using the same idea for regular expressions, the infix regular expression

```
((a|b)*aba*)*(a|b)(a|b)
```

can be represented as

```
ab|*a&b&a*&*ab|&ab|&
```

in postfix notation. Notice we removed the need for parentheses, but added the `&` operator for concatenation. Your program (for now) will work with regular expressions in their postfix form. I will provide you with a tool for converting infix expressions to postfix for testing purposes.

2.2 Compiling a Postfix Regular Expression

We will convert a postfix regular expression into an NFA using a stack, where each element on the stack is an NFA. The input expression is scanned from left to right and the stack is manipulated as shown in Figure 2. When this process is completed, the stack should contain exactly one NFA. We construct our NFA's based on the inductive rules given in Figure 3 as discussed in class. We will typically searching for patterns embedded in strings;

Since the user is typically trying to look for patterns embedded in long strings, we infer that the input regular expression α really is shorthand for $.^{*}\alpha.^{*}$ (unless the beginning and end of line anchors are used). Instead of modifying the input regular expression, the resulting NFA is usually modified appropriately. The algorithm in Figure 2 assumes that any anchors have been stripped from the beginning and/or end of the input postfix regular expression.

```
while (not end of postfix expression) {
    c = next character in postfix expression;
    if (c == \) { /* escape */
        if (end of postfix expression) syntax error;
        c = next character in postfix expression;
        if (c == 'd')
            push(NFA that accepts a digit);
        else if (c == 'D')
            push(NFA that accepts a non-digit);
        else if other escape chars...
            push(NFA for other escape chars)...
    } else if (c == '.') {    /* wildcard */
        push(NFA that accepts *any* single character);
    } else if (c == '&') {
        nfa2 = pop();
        nfa1 = pop();
        push(NFA that accepts the concatenation of $L$(nfa1) followed by $L$(nfa2));
    } else if (c == '|') {
        nfa2 = pop();
        nfa1 = pop();
        push(NFA that accepts $L(\texttt{nfa1}) \cup L(\texttt{nfa2})$);
    } else if (c == '*') {
        nfa = pop();
        push(NFA that accepts $L(\texttt{nfa})^*$);
    } else if (c == '+') {
        nfa = pop();
        push(NFA that accepts $L(\texttt{nfa})^+$);
    } else if (c == '?') {
        nfa = pop();
        push(NFA that accepts $\{\epsilon\} \cup L(\texttt{nfa})$);
    } else
        push(NFA that accepts a single character c);
}
```

Figure 2: Converting a postfix regular expression into an NFA.

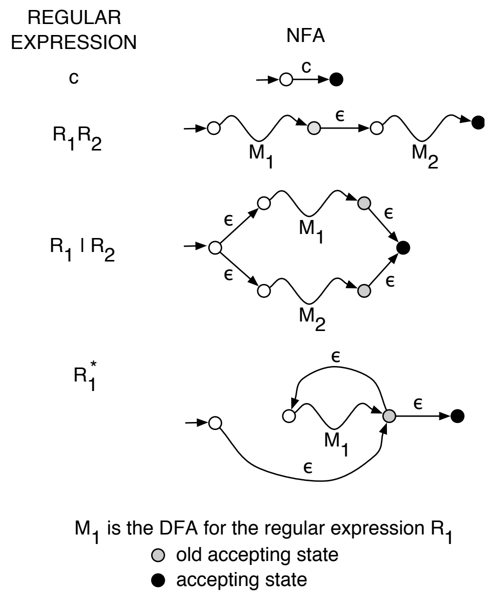

Figure 3: Four cases used to build NFA's from regular expressions consisting of a *single character c*, *concatenation, union*, and *Kleene closure*. In each case there is only one start state (with no edges leading to it) and only one accepting state (with no edges emanating from it).

3 NFA data structure

We discussed several alternatives for representing NFA's in class. While constructing the NFA using the inductive rules in Figure 3, I suggest using the following simple representation.

- Represent control states with positive integers.
- Store labeled directed edges with integer pairs to hold source and destination states and use a (16-bit or more) integer to store the label. Use normal ASCII values for normal character labels, but use internally defined constants for ϵ-transitions and the various wild cards.
- Each NFA will have a unique start and final state during construction (optimized NFA's may have more).
- The NFA's digraph can be stored as a set of edges. Your choice of data structures used to represent a set may affect performance.

It may be worthwhile to switch to an alternate data structure after the initial NFA is constructed. I would suggest beginning with the simplest representation. Use appropriate data encapsulation tricks so that the internal data structure may be modified in a future version.

4 String acceptance

You will need to create a method for string acceptance as discussed in class. This method must be able handle nondeterminism including ϵ-transitions. If you follow the simple construction above, the NFA will contain exactly one start and one final state. Potential optimizations allow for more efficient acceptance algorithms – these include the following:

- Removing ϵ-transitions and subsequent unreachable states *(note: multiple accepting states possible)*;
- Converting the NFA into a DFA using the subset construction *(note: watch out for exponential explosion of states)*;
- Minimizing DFA using the quotient construction.

You are not required to perform any optimizations.

5 Searching for Patterns in Files

Your program will first read a postfix regular expression specified on the command line and build the corresponding NFA. Note that you may need to strip anchors from the beginning and end of the expression before processing. The NFA will then be modified according to the original existence of the anchors as described in Section 2.2.

Next, your program will read from *standard input* one line at time (strip any trailing newlines and/or carriage returns). The resulting string is then tested for acceptance by the NFA and is echoed to *standard output* if accepted.

The program will use the following exit codes:

0 At least one match found

1 No matches found

2 Bogus postfix expression

For example, if the name of your program is `mygrep`, an example run of your program might look like the following:

```
$ ./mygrep 'ud&p&' < /etc/services
tcpmux            1/udp     # TCP Port Service Multiplexer
compressnet       2/udp     # Management Utility
compressnet       3/udp     # Compression Process
#                 4/udp    Unassigned
rje               5/udp     # Remote Job Entry
<snip>
```

Here is an example using the beginning of line anchor:

```
$ ./mygrep '^ud&p&' < /etc/services
udp-sr-port     1624/udp   # udp-sr-port
udp-sr-port     1624/tcp   # udp-sr-port
udpradio        1833/udp     # udpradio
udpradio        1833/tcp     # udpradio
udpplus         5566/udp    # UDPPlus
udpplus         5566/tcp    # UDPPlus
```

You are free to implement your solution in your language of choice. If I do not have a compiler or interpreter for your the language you choose, I may ask you to demo your program for me.

6 Submitting your solution

All of your source code, documentation, etc... will be archived in a single compressed archive file and submitted electronically. With every project submission you are to include a text file named `README` that includes

1. The name of all authors (which hopefully includes you), your student ID number, and an email address you can be contacted at.
2. A brief description of what you are submitting.
3. A description of how to build and use your program.
4. A list of all files that should be in the archive, and a one line description of each file.

Make sure you thoroughly test your code (I will). Start early and ask questions. Follow the directions found at the following site for submitting your solution:

`http://ezekiel.vancouver.wsu.edu/~cs317/submit/submit.html`

This project is due at midnight on the due date.