Course Project 2
Regular Expressions

CSE 30151 Spring 2017

Version of February 16, 2017

In this project, you'll write a regular expression matcher similar to `grep`, called `mere` (for match and echo using regular expression). This has three major steps: first, parse a regular expression into regular operations; second, execute the regular operations to create a NFA; third, run the NFA on input strings. Because we use a linear-time NFA recognition algorithm, our regular expression matcher will actually be much faster than one written using Perl or Python's regular expression engine. (Most implementations of `grep`, as well as Google RE2, are linear like ours.)

You will need a correct solution for CP1 to complete this project. If your CP1 doesn't work correctly (or you just weren't happy with it), you may use the official solution or another team's solution, as long as you properly cite your source.

Getting started

We've made some minor updates, so please have one team member run the commands

```
git pull https://github.com/ND-CSE-30151-SP17/theory-project-skeleton
git push
```

and then other team members should run `git pull`. The project repository should then include the following files:

```
bin/
  compare_nfa
  parse_re
  union_nfa
  concat_nfa
  star_nfa
  mere
examples/
  sipser-n{1,2,3,4}.nfa
tests/
  test-cp2.sh
  time-cp2.sh
cp2/
```

Please place the programs that you write into the `cp2/` subdirectory.

1 Parser

Note: Parts 1 and the subparts of 2 and 3 can all be written and tested independently.

In this first part, we'll write a parser for regular expressions. Our regular expressions allow union (`|`), concatenation, Kleene star (`*`), and empty set (`@`). Parentheses are used for grouping. The grammar is as follows. The start nonterminal S is Union, and we write terminal symbols inside boxes.

$$
\begin{array}{rll}
\mathsf{Union} \to & \mathsf{Union}\boxed{|}\mathsf{Concat} & \\
\mathsf{Union} \to & \mathsf{Concat} & \\
\mathsf{Concat} \to & \mathsf{Concat}\ \mathsf{Unary} & \text{if not followed by } \boxed{|} \text{ or } \boxed{)} \text{ or end of string} \\
\mathsf{Concat} \to & \varepsilon & \text{otherwise} \\
\mathsf{Unary} \to & \mathsf{Primary}\boxed{*} & \\
\mathsf{Unary} \to & \mathsf{Primary} & \\
\mathsf{Primary} \to & \boxed{a} & \text{for } a \text{ not in } \boxed{@}\ \boxed{(}\ \boxed{)}\ \boxed{*}\ \boxed{|} \\
\mathsf{Primary} \to & \boxed{@} & \\
\mathsf{Primary} \to & \boxed{(}\mathsf{Union}\boxed{)} &
\end{array}
$$

A parser for CFG for a programming language essentially converts the CFG into a deterministic pushdown automaton (DPDA) and runs the DPDA on programs. There are several ways of doing this conversion. This is a complex topic that you can learn more about by taking Compilers. Here, we'll take the simplest route, a *recursive-descent parser* (which many of you implemented in Data Structures for a fragment of Scheme). The pseudocode is shown in Algorithm 1. The top-level function is parseRegexp. For each nonterminal symbol X, there is a function, parse$X$, which tries to read in a string that matches X. For example, if the regular expression is `ab`, the trace of the parser is shown in Figure 1.

"If this is a pushdown automaton," you must be asking, "where is the stack?" The call stack itself is being used as the stack: every time we call a function, we push a return address, and every time a function returns, the return address is popped. So the stack alphabet is the set of return addresses, of which there is clearly a finite number.

Each of these functions returns the *semantics* of the (sub)expression that it reads in. The functions emptyset, epsilon, and symbol create semantic objects, and the functions union, concat, star build semantic objects from smaller semantic objects. Eventually, the semantics of a regular expression will be an NFA equivalent to the regular expression.

But initially, to facilitate unit testing, the semantics will just be a string containing the regular expression written in prefix notation (kind of like Scheme). For example, the regular expression `(ab|a)*` should become the string:

```
star(union(concat(symbol(a),symbol(b)),symbol(a)))
```

So, for testing purposes, these functions should do the following:

emptyset()

- Arguments: none

line		input
	parseRegexp()	ab
2:	$M \leftarrow$ parseUnion()	ab
8:	$M \leftarrow$ parseConcat()	ab
17:	$M \leftarrow$ parseUnary()	ab
22:	$M \leftarrow$ parsePrimary()	ab
38:	read a	b
39:	**return** symbol(a)	b
22:	$M \leftarrow$ symbol(a)	b
29:	**return** symbol(a)	b
17:	$M \leftarrow$ symbol(a)	b
19:	$M \leftarrow$ concat(symbol(a), ParseUnary())	b
22:	$M \leftarrow$ parsePrimary()	b
38:	read b	ε
39:	**return** symbol(b)	ε
22:	$M \leftarrow$ symbol(b)	ε
29:	**return** symbol(b)	ε
19:	$M \leftarrow$ concat(symbol(a), symbol(b))	ε
20:	**return** concat(symbol(a), symbol(b))	ε
8:	$M \leftarrow$ concat(symbol(a), symbol(b))	ε
12:	**return** concat(symbol(a), symbol(b))	ε
2:	$M \leftarrow$ concat(symbol(a), symbol(b))	ε
4:	**return** concat(symbol(a), symbol(b))	ε
	concat(symbol(a), symbol(b))	ε

Figure 1: Trace of parser (Algorithm 1) for example regular expression `ab`. Computed values that have been substituted into the code are shown in blue.

Algorithm 1 Pseudocode for recursive-descent parser.

```
 1: function parseRegexp()
 2:     $M \leftarrow$ parseUnion()
 3:     if no next token then
 4:         return $M$
 5:     else
 6:         error

 7: function parseUnion()
 8:     $M \leftarrow$ parseConcat()
 9:     while next token is $\boxed{|}$ do
10:         read $\boxed{|}$
11:         $M \leftarrow$ union($M$, parseConcat())
12:     return $M$

13: function parseConcat()
14:     if no next token, or next token is $\boxed{|}$ or $\boxed{)}$ then
15:         return epsilon()
16:     else
17:         $M \leftarrow$ ParseUnary()
18:         while next token exists and is not $\boxed{|}$ or $\boxed{)}$ do
19:             $M \leftarrow$ concat($M$, ParseUnary())
20:         return $M$

21: function parseUnary()
22:     $M \leftarrow$ parsePrimary()
23:     if next token is $\boxed{*}$ then
24:         read $\boxed{*}$
25:         return star($M$)
26:     else
27:         return $M$

28: function parsePrimary()
29:     if next token is $\boxed{(}$ then
30:         read $\boxed{(}$
31:         $M \leftarrow$ parseUnion()
32:         read $\boxed{)}$
33:         return $M$
34:     else if next token is $\boxed{@}$ then
35:         read $\boxed{@}$
36:         return emptyset()
37:     else if next token is not in $\boxed{(}$ $\boxed{)}$ $\boxed{*}$ $\boxed{|}$ then
38:         read $a$
39:         return symbol($a$)
40:     else
41:         error
```

- Returns: string `"emptyset()"`

emptyset()

epsilon()

- Arguments: none
- Returns: string `"epsilon()"`

symbol(a)

- Argument: alphabet symbol $a \in \Sigma$
- Returns: string "`symbol(`a`)`"

union(M_1, M_2)

- Arguments: strings M_1, M_2
- Returns: string "`union(`M_1`,`M_2`)`"

concat(M_1, M_2)

- Arguments: strings M_1, M_2
- Returns: string "`concat(`M_1`,`M_2`)`"

star(M)

- Arguments: string M
- Returns: string "`star(`M`)`"

Write a program called `parse_re` to test your parser:

`parse_re` *regexp*

should output the prefix-notation version of *regexp*. Test your program by running `test-cp2.sh`.

2 Easy operations

Write functions that construct the following NFAs.

emptyset()

- Arguments: none
- Returns: NFA recognizing the empty language $\emptyset$

epsilon()

- Arguments: none
- Returns: NFA recognizing the language $\{\varepsilon\}$

symbol(a)

- Argument: Alphabet symbol $a \in \Sigma$
- Returns: NFA recognizing the language $\{a\}$

These operations are trivial to implement, and we won't bother writing tests for them.

3 Regular operations

Write functions that perform the regular operations, using the constructions given in the book:

union(M_1, M_2)

- Arguments: NFAs M_1, M_2
- Returns: NFA recognizing language $L(M_1) \cup L(M_2)$

concat(M_1, M_2)

- Arguments: NFAs M_1, M_2
- Returns: NFA recognizing language $L(M_1)L(M_2)$

star(M)

- Argument: NFA M
- Returns: NFA recognizing language $L(M)^*$

Optional: The book's construction creates a lot of ε-transitions, and in later projects, these ε-transitions will proliferate. For greater efficiency, you could try using the construction in Appendix A, which creates no ε-transitions. (However, note that if you do this, then the tests for `union_nfa`, `concat_nfa`, and `star_nfa` will fail; please contact the instructor if you choose this option.)

Write three programs, called `union_nfa`, `concat_nfa`, and `star_nfa`, to test your operations. Each takes one or two command-line arguments, each of which is the name of a file containing a NFA, in the same format you used in CP1, and each writes a NFA to stdout in the same format.

`union_nfa` *nfafile1 nfafile2*	Writes union of NFAs to stdout
`concat_nfa` *nfafile1 nfafile2*	Writes concatenation of NFAs to stdout
`star_nfa` *nfafile*	Writes Kleene star of NFA to stdout

Test your programs by running `test-cp2.sh`.

4 Putting it together

Write a function that puts all the above functions and your NFA simulator from CP1 together:

- Arguments: regular expression α, string w
- Return: true if α matches w, false otherwise.

Put your function into a command-line tool called `mere`:

`mere` *regexp*

where *regexp* is a regular expression. The program should read zero or more lines from stdin and write to stdout just the lines that match the regular expression. Unlike `grep`, the regular expression should match the entire line, not just part of the line. Test your program by running `tests/test-cp2.sh`.

5 Testing performance

Now run `tests/time-cp2.sh`. This script, for $n = 1, \ldots, 20$, creates the regular expression

$$\underbrace{\texttt{(a|)} \cdots \texttt{(a|)}}_{n \text{ copies}} \underbrace{\texttt{a} \cdots \texttt{a}}_{n \text{ copies}}$$

and tries to match it against the string $\texttt{a}^n$, using a short Perl script, our solution (`bin/mere`), and your solution (`cp2/mere`). Observe that Perl has an exponential running time, whereas our solution has roughly quadratic running time. Hopefully, yours does too.

In order to get full credit, your solution must run faster than Perl for *some* value of n. If needed, you may increase the maximum value of n by modifying the variable `NMAX` in `time-cp2.sh`. Please be sure to commit your change.

However, a better strategy would be to optimize your code, looking in particular for inadvertent linear searches and copies. (Python code is especially prone to the former, and C++ code is especially prone to the latter.) The hot spots are the innermost loops of the intersection and emptiness algorithms, so look there first.

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, `cd` into its root directory, run `make -C cp2`, and run `tests/test-cp2.sh`. You're advised to try all of the above steps and ensure that all tests pass.

To submit your work, please push your repository to Github and then create a new release with tag version `cp2` (note that the tag version is not the same thing as the release title). If you are making a partial submission, then use a tag version of the form `cp2-12345`, indicating which parts you're submitting.

Rubric

Parser	9
Easy operations	3
Union	4
Concatenation	4
Kleene star	4
Main program	3
Speed	3
Total	30

A The Berry-Sethi construction

The Berry-Sethi construction [Berry and Sethi, 1986] is a more efficient way of converting a regular expression to a NFA. Unlike the construction in the book, it does not create any ε-transitions.

Base cases If $\alpha = \emptyset$, $\alpha = \varepsilon$, or α is a single symbol $a \in \Sigma$, the construction is the same as in the book.

Union If $\alpha = \alpha_1 \cup \alpha_2$:

- Convert α_1 and α_2 to

  $$M_1 = (Q_1, \Sigma, s_1, F_1)$$
  $$M_2 = (Q_2, \Sigma, s_2, F_2)$$

  where $Q_1 \cap Q_2 = \emptyset$.
- Create a new start state s.
- For each transition $s_i \xrightarrow{a} r$, create a transition $s \xrightarrow{a} r$.
- State s is an accept state if either s_1 or s_2 is.
- Delete s_1, s_2, and their outgoing transitions.

Concatenation If $\alpha = \alpha_1 \circ \alpha_2$:

- Convert α_1 and α_2 to

  $$M_1 = (Q_1, \Sigma, s_1, F_1)$$
  $$M_2 = (Q_2, \Sigma, s_2, F_2)$$

  where $Q_1 \cap Q_2 = \emptyset$.
- For each accept state $q \in F_1$ and transition $s_2 \xrightarrow{a} r$, create a transition $q \xrightarrow{a} r$.
- Each accept state $q \in F_1$ continues to be an accept state iff s_2 is an accept state.
- Delete s_2 and its outgoing transitions.

Kleene star If $\alpha = \alpha_1^*$:

- Convert α_1 to

  $$M_1 = (Q_1, \Sigma, s_1, F_1).$$
- For each accept state $q \in F_1$ and transition $s_1 \xrightarrow{a} r$, create a transition $q \xrightarrow{a} r$.
- Make s_1 an accept state.

References

Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986. URL `http://dx.doi.org/10.1016/0304-3975(86)90088-5`.