Course Project 2
Regular Expressions

CSE 30151 Spring 2017

Version of January 19, 2017

In this project, we will write a regular expression matcher similar to `grep`. Our matcher will be called `mere` (for match and echo using regular expression). This has two major steps: first, parse a regular expression into regular operations; second, execute the regular operations to create a NFA. Because we'll 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.)

Getting started

The project repository includes the following files:

```
bin/
  compare_nfa
  parse_re
  union_nfa
  concat_nfa
  star_nfa
  mere
examples/
  sipser-n1.nfa
  sipser-n2.nfa
  sipser-n3.nfa
  sipser-n4.nfa
tests/
  test-cp2.sh
doc/
  cp2.pdf
cp2/
```

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

1 Parser

Note: Parts 1, 2, and 3 can 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 (with $S = \mathsf{Union}$):

$\mathsf{Union} \to \mathsf{Union}\ \boxed{\texttt{|}}\ \mathsf{Concat}$	
$\mathsf{Union} \to \mathsf{Concat}$	
$\mathsf{Concat} \to \mathsf{Concat}\ \mathsf{Unary}$	if not followed by $\boxed{\texttt{|}}$ or $\boxed{\texttt{)}}$ or end of string
$\mathsf{Concat} \to \varepsilon$	otherwise
$\mathsf{Unary} \to \mathsf{Primary}\ \boxed{\texttt{*}}$	
$\mathsf{Unary} \to \mathsf{Primary}$	
$\mathsf{Primary} \to a$	$\forall a \in \Sigma$
$\mathsf{Primary} \to \boxed{\texttt{@}}$	
$\mathsf{Primary} \to \boxed{\texttt{(}}\ \mathsf{Union}\ \boxed{\texttt{)}}$	

The alphabet Σ consists of all the symbols that are used anywhere in the regular expression. None of the following characters are allowed to be symbols: `{};:|*()@`

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 in Compilers. Here, we'll take the simplest route, which is a *recursive-descent parser*. The pseudocode is shown in Algorithm 1. For each nonterminal symbol X, there is a function, parse$X$, which tries to read in a string that matches X, and returns the semantic interpretation of that string.

"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 a function is called, something is pushed, and every time a function returns, something is popped.

The functions union, concat, star, emptyset, epsilon, and symbol are called *semantic actions*. Eventually, they will build up a NFA. But while you're testing the parser, write stubs for the semantic actions, such that union(x, y) returns the string `union(`x`,`y`)`, and so on. For example, the regular expression `(ab|a)*` should become the string

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

Algorithm 1 Pseudocode for recursive-descent parser.

```
function parseUnion()
    M ← parseConcat()
    while next token is |  do
        read |
        M ← union(M, parseConcat())
    return M

function parseConcat()
    if no next token, or next token is | or ) then
        return epsilon()
    M ← ParseUnary()
    while next token exists and is not | or ) do
        M ← concat(M, ParseUnary())
    return M

function parseUnary()
    M ← parsePrimary()
    if next token is * then
        read *
        return star(M)
    else
        return M

function parsePrimary()
    if next token is ( then
        read (
        M ← parseUnion()
        read )
        return M
    else if next token is @ then
        return emptyset()
    else
        read a
        return symbol(a)
```

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

`parse_re` *regexp*

should output (a string representing) the abstract syntax tree for *regexp*. Test your program by running `test-cp2.sh`.

2 Easy operations

Write functions that construct the following NFAs:

- emptyset() returns a NFA recognizing the empty language ($\emptyset$)
- symbol(a) returns a NFA recognizing the language $\{a\}$, for any $a \in \Sigma$
- epsilon() returns a NFA recognizing the language $\{\varepsilon\}$.

These operations are trivial to implement using the singleton operation from Project 1, 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) returns the NFA $M_1 \cup M_2$
- concat(M_1, M_2) that returns the NFA $M_1 \circ M_2$
- star(M) that returns the NFA 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.)

Write three programs to test your operations:

`union_nfa` *nfafile nfafile*	Writes union of NFAs to stdout
`concat_nfa` *nfafile nfafile*	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 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 lines from stdin and write to stdout just the lines that match the regular expression. Test your program by running `test-cp3.sh`. Unlike `grep`, the regular expression should match the entire line, not just part of the line. Test your program by running `test-cp2.sh`.

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, run `make` in subdirectory `cp2`, and then 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 `cp2`. If you need to re-submit, create another release with a new tag starting with `cp2`, like `cp2.1`.

Rubric

Parser	12
Easy operations	3
Union	4
Concatenation	4
Kleene star	4
Main program	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.

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

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.

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.

Similarly for the cross-product construction.

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`.