Course Project 2
Regular Expressions

CSE 30151 Spring 2018

Version of 2018/01/19

In this project, you'll write a regular expression matcher similar to `grep`. 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

To make sure your repository is up to date, please have one team member run the commands

```
git pull https://github.com/ND-CSE-30151-SP18/theory-project-skeleton
git push
```

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

```
bin.{linux,darwin}/
  parse_re
  re_to_nfa
  agrep
  agrep.pl
tests/
  test-cp2.sh
cp2/
```

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

1 Parser

Note: This part and part 2 can be done in parallel.

In this first part, we'll write a parser for regular expressions. We'll write the simplest kind of parser, a *recursive-descent* parser.

Recursive-descent parsing

To illustrate how to do this, let's look at the simple grammar from Sipser, Example 2.4. The start symbol is Expr.

$$\begin{aligned}
\mathsf{Expr} &\rightarrow \mathsf{Term}\ \ \{\texttt{+}\ \mathsf{Term}\} \\
\mathsf{Term} &\rightarrow \mathsf{Factor}\ \ \{\texttt{*}\ \mathsf{Factor}\} \\
\mathsf{Factor} &\rightarrow \texttt{(}\ \mathsf{Expr}\ \texttt{)} \\
\mathsf{Factor} &\rightarrow \texttt{1}
\end{aligned}$$

We write terminal symbols using `typewriter font` and nonterminal symbols using sans-serif font. The curly braces mean "zero or more copies of," like Kleene star, but we use a different notation to avoid confusion with the Kleene star in regular expressions. The reason for this is that it's more amenable to recursive-descent parsing.

Algorithm 1 Recursive-descent parser for grammar in Example 2.4.

```
function parse(w)
    x, i ← parseExpr(w, 0)
    if i = |w| then
        return x
    else
        error
function parseExpr(w, i)
    x, i ← parseTerm(w, i)
    args ← [x]
    while w_{i+1} = + do
        x, i ← parseTerm(w, i + 1)
        append x to args
function parseTerm(w, i)
    x, i ← parseFactor(w, i)
    args ← [x]
    while w_{i+1} = * do
        x, i ← parseFactor(w, i + 1)
        append x to args
    return node("mul", args), i
function parseFactor(w, i)
    if w_{i+1} = 1 then
        return node("const", 1), i + 1
    else if w_{i+1} = ( then
        x, i = parseExpr(w, i + 1)
        if w_{i+1} ≠ ) then
            error
        return x, i + 1
    else
        error
```

Algorithm 1 shows pseudocode for a recursive-descent parser for this grammar. The function parse takes a string containing an arithmetic expression and returns a tree for the expression. It has a helper function for each nonterminal symbol. The helper function for nonterminal X takes two arguments, w and i ($0 \leq i \leq |w|$), and tries to find a j such that $X \Rightarrow^* w_{i+1} \cdots w_j$. If there is one, it builds a tree node and returns the node and j. If there isn't such a j, it generates an error.

Back to regular expressions

Below is a grammar for regular expressions. The start nonterminal is Expr. Let Σ be the set of all (ASCII or Unicode) characters.

$$
\begin{aligned}
\mathsf{Expr} &\to \mathsf{Term}\,\{\,\texttt{|}\ \mathsf{Term}\} \\
\mathsf{Term} &\to \{\mathsf{Factor}\} \\
\mathsf{Factor} &\to \mathsf{Primary}\ \texttt{*} \\
\mathsf{Factor} &\to \mathsf{Primary} \\
\mathsf{Primary} &\to a & a \in \Sigma \setminus \{\texttt{(},\texttt{)},\texttt{*},\texttt{|},\texttt{\\}\} \\
\mathsf{Primary} &\to \texttt{(}\ \mathsf{Expr}\ \texttt{)}
\end{aligned}
$$

Implementing the parser for this grammar should be analogous to the one shown for the grammar of arithmetic expressions, except that in the rule for Term, there's no operator between the Factors. How do we know when to try to parse another Factor and when to stop? Note that every Term must be followed by the end of the string, `|`, or `)`. And a Factor can't start with any of these. So we can look ahead one character, and if it's the end of the string, `|`, or `)`, then stop; otherwise, try to parse another Factor.

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

`parse_re` *regexp*

should output a string representation of the syntax tree for *regexp*. For example,

```
$ parse_re '(ab|)*'
star(union(concat(symbol("a"),symbol("b"))),epsilon())
```

Note that:

- A `union` or a `concat` always has two or more arguments.
- There should never be a `union` of `union`s; instead, combine them into a single `union`. Similarly for `concat`.

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

2 Compiler

You've already implemented the regular operations (union, concatenation, and Kleene star) in the previous project. We also need functions that construct the following NFAs:

epsilon()

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

symbol(a)

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

Then, write a function that converts regular expressions to NFAs:

- Argument: regular expression α
- Return: NFA M equivalent to α

Put your function into a program called `re_to_nfa`:

`re_to_nfa` *regexp*

should write an equivalent NFA to stdout. Test your program using `test-cp2.sh`.

3 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 `agrep`:

`agrep` *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`.

The test script also tests the time complexity of `agrep`. This test is the same as in CP1, but now we can say a bit more about it. For various values of n, it creates the regular expression $\texttt{(a|)}^n\texttt{a}^n$ and tries to match it against the string $\texttt{a}^n$, using our `agrep` and yours. For fun, we've provided a Perl implementation, called `agrep.pl`, which you can try for comparison. (I ran out of patience and killed it.)

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 `cp2-123`, indicating which parts you're submitting.

Rubric

`parse_re`	12
epsilon	3
symbol	3
`re_to_nfa`	6
`agrep`	
correctness	3
time complexity	3
Total	30