

# Course Project 2

## Regular Expressions

CSE 30151 Spring 2018

Version of 2018/01/25

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 (among others):

```
bin.{linux,darwin}/
  parse_re
  union_nfa
  concat_nfa
  star_nfa
  string_nfa
  re_to_nfa
  agrep
  compare_nfa
perl/
  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 2.1 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`.

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

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 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  $\Sigma$  be the set of all (ASCII or Unicode) characters.

$$\begin{aligned}\text{Expr} &\rightarrow \text{Term } \{ | \text{ Term}\} \\ \text{Term} &\rightarrow \{\text{Factor}\} \\ \text{Factor} &\rightarrow \text{Primary } * \\ \text{Factor} &\rightarrow \text{Primary} \\ \text{Primary} &\rightarrow a & a \in \Sigma \setminus \{(, ), *, |, \backslash\} \\ \text{Primary} &\rightarrow ( \text{Expr} )\end{aligned}$$

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`.

---

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

---

```

1: function parse( $w$ )
2:    $x, i \leftarrow \text{parseExpr}(w, 0)$ 
3:   if  $i = |w|$  then
4:     return  $x$ 
5:   else
6:     error
7:   function parseExpr( $w, i$ )
8:      $x, i \leftarrow \text{parseTerm}(w, i)$ 
9:      $args \leftarrow [x]$ 
10:    while  $w_{i+1} = +$  do
11:       $x, i \leftarrow \text{parseTerm}(w, i + 1)$ 
12:      append  $x$  to  $args$ 
13:    function parseTerm( $w, i$ )
14:       $x, i \leftarrow \text{parseFactor}(w, i)$ 
15:       $args \leftarrow [x]$ 
16:      while  $w_{i+1} = *$  do
17:         $x, i \leftarrow \text{parseFactor}(w, i + 1)$ 
18:        append  $x$  to  $args$ 
19:      return  $\text{node}(\text{"mul"}, args), i$ 
20:    function parseFactor( $w, i$ )
21:      if  $w_{i+1} = 1$  then
22:        return  $\text{node}(\text{"const"}, 1), i + 1$ 
23:      else if  $w_{i+1} = ($  then
24:         $x, i = \text{parseExpr}(w, i + 1)$ 
25:        if  $w_{i+1} \neq )$  then
26:          error
27:        return  $x, i + 1$ 
28:      else
29:        error

```

---

Write a program to test your parser:

`parse_re` *regexp*

- *regexp*: a regular expression
- Output: string representation of the syntax tree for *regexp*

The output format should look like a Scheme expression. For example,

```
$ parse_re 'a'
(symbol "a")
$ parse_re ''
(epsilon)
$ parse_re '(a)'
(group (symbol "a"))
$ parse_re 'a*'
(star (symbol "a"))
$ parse_re 'abc'
(concat (symbol "a") (symbol "b") (symbol "c"))
$ parse_re 'a|b|c'
(union (symbol "a") (symbol "b") (symbol "c"))
```

Note that, to facilitate testing:

- A `union` or a `concat` always has two or more arguments.
- There should never be a `union` of unions; combine `(union (union  $\alpha$   $\beta$ )  $\gamma$ )` or `(union  $\alpha$  (union  $\beta$   $\gamma$ ))` into `(union  $\alpha$   $\beta$   $\gamma$ )`. Similarly for `concat`.

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

## 2 Converter

### 2.1 Regular operations

Write a function that creates an NFA that accepts exactly one string, and a program to test it:

`string_nfa` *w*

- *w*: a string (possibly empty)
- Output: an NFA recognizing the language  $\{w\}$

Write functions that perform the three regular operations, using the constructions given in the book, and programs to test them:

`union_nfa` *M*<sub>1</sub> *M*<sub>2</sub>

- *M*<sub>1</sub>, *M*<sub>2</sub>: NFAs
- Output: NFA recognizing language  $L(M_1) \cup L(M_2)$

`concat_nfa` *M*<sub>1</sub> *M*<sub>2</sub>

- $M_1, M_2$ : NFAs
- Output: NFA recognizing language  $L(M_1) \circ L(M_2)$

star  $M$

- $M$ : an NFA
- Output: NFA recognizing language  $L(M)^*$

Test all of these programs by running `test-cp2.sh`.

## 2.2 Building the NFA

Write a function that converts (the syntax tree of) a regular expression to an NFA, by walking the tree bottom-up and using the operations implemented in Section 2.1. Then combine your regular expression parser with this function into a test program:

`re_to_nfa regexp`

- `regexp`: Regular expression
- Output: NFA  $M$  equivalent to `regexp`

Test your program using `test-cp2.sh`.

## 3 Putting it together

Finally, combine your regular expression converter with your NFA simulator from CP1 to write a `grep` replacement, called `agrep` (for “automaton-based `grep`”):

`agrep regexp`

- `regexp`: regular expression
- Input: strings (one per line)
- Output: the input strings that match `regexp`

Note that 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  $(a|)^n a^n$  and tries to match it against the string  $a^n$ , 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 `studentnn.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

Part 1	
parsing	3
building syntax tree	3
parse_re	3
Part 2	
string_nfa	3
union_nfa	3
concat_nfa	3
star_nfa	3
re_to_nfa	3
Part 3 (agrep)	
correctness	3
time complexity	3
Total	30