CS 301

Lecture 05 – Applications of Regular Languages

Stephen Checkoway

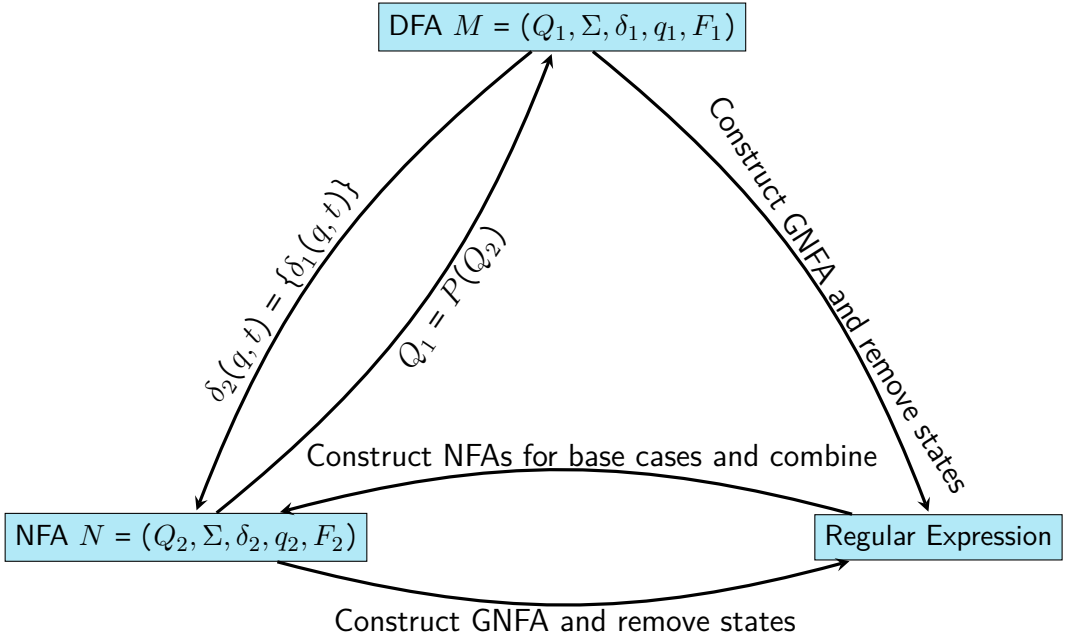
January 31, 2018

Characterizing regular languages

The following four statements about the language A are equivalent

- The language A is regular
- Some DFA M recognizes A (i.e., $L(M) = A$)
- Some NFA N recognizes A (i.e., $L(N) = A$)
- Some regular expression R generates (or describes) A (i.e., $L(R) = A$)

Converting between DFA, NFA, regex



Types of regular expressions

- Formal language-theoretic regular expressions (this class)
- Portable Operating System Interface (POSIX) basic and extended regular expressions
- Perl-compatible regular expressions (PCRE) (not always regular!)
  Many languages use similar regex, Java, JavaScript, Python, Ruby, ...
- Vim regular expressions
- Boost regular expressions
- ...

Regex in text processing

Alphabet is usually ASCII characters

Common tasks include

- Finding lines that match (or have a substring that matches) the regex
- Text substitution: match a regex, replace parts of it
  E.g., restructuring formatted data
- Validating input
  E.g., untainting user input in Perl
- Web (or other data) scraping
- Syntax highlighting in editors

POSIX regex

Most characters match literally

E.g., the formal regex $\underline{\texttt{red}}$ would be written `red` or `/red/`

Metacharacters

- Equivalent to $\underline{\Sigma}$: matches any character (not completely true as newlines are typically not matched)

POSIX regex

Most characters match literally

E.g., the formal regex $\underline{\texttt{red}}$ would be written `red` or `/red/`

Metacharacters

- `.` Equivalent to $\underline{\Sigma}$: matches any character (not completely true as newlines are typically not matched)
- `[ ]` Matches characters contained in the brackets
  E.g., `[abc]` is $\underline{\texttt{a} \mid \texttt{b} \mid \texttt{c}}$; `[a-zA-Z0-9]` is
  $\underline{\texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9}}$

POSIX regex

Most characters match literally

E.g., the formal regex $\underline{\texttt{red}}$ would be written `red` or `/red/`

Metacharacters

`.` Equivalent to $\underline{\Sigma}$: matches any character (not completely true as newlines are typically not matched)

`[ ]` Matches characters contained in the brackets
E.g., `[abc]` is $\underline{\texttt{a} \mid \texttt{b} \mid \texttt{c}}$; `[a-zA-Z0-9]` is
$\underline{\texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9}}$

`[^ ]` Matches characters not contained in the brackets
E.g., `[^abc]` matches any character except `a`, `b`, or `c`

POSIX regex

Most characters match literally

E.g., the formal regex red would be written `red` or `/red/`

Metacharacters

`.` Equivalent to Σ: matches any character (not completely true as newlines are typically not matched)

`[ ]` Matches characters contained in the brackets
E.g., `[abc]` is a | b | c; `[a-zA-Z0-9]` is
a | b | · · · | z | A | B | · · · | Z | 0 | 1 | · · · | 9

`[^ ]` Matches characters not contained in the brackets
E.g., `[^abc]` matches any character except a, b, or c

`^` Matches the start of the string or the start of the line

POSIX regex

Most characters match literally

E.g., the formal regex red would be written `red` or `/red/`

Metacharacters

- `.` Equivalent to Σ: matches any character (not completely true as newlines are typically not matched)
- `[ ]` Matches characters contained in the brackets
  E.g., `[abc]` is a | b | c; `[a-zA-Z0-9]` is a | b | ··· | z | A | B | ··· | Z | 0 | 1 | ··· | 9
- `[^ ]` Matches characters not contained in the brackets
  E.g., `[^abc]` matches any character except a, b, or c
- `^` Matches the start of the string or the start of the line
- `$` Matches the end of the string or the end of the line

POSIX regex

Most characters match literally

E.g., the formal regex red would be written `red` or `/red/`

Metacharacters

`.` Equivalent to Σ: matches any character (not completely true as newlines are typically not matched)

`[ ]` Matches characters contained in the brackets
E.g., `[abc]` is a | b | c; `[a-zA-Z0-9]` is
a | b | ··· | z | A | B | ··· | Z | 0 | 1 | ··· | 9

`[^ ]` Matches characters not contained in the brackets
E.g., `[^abc]` matches any character except a, b, or c

`^` Matches the start of the string or the start of the line

`$` Matches the end of the string or the end of the line

`( )` Defines a subexpression

POSIX regex

More metacharacters

- `*` Matches the preceding element zero or more times
  E.g., `ab*c` is $\underline{\texttt{ab}^*\texttt{c}}$
- `+` Matches the preceding element one or more times
  E.g., `ab+c` is $\underline{\texttt{abb}^*\texttt{c}}$
- `?` Matches the preceding element zero or one times
  E.g., `ab?c` is $\underline{\texttt{a(b} \mid \varepsilon\texttt{)c}}$
- `{m,n}` Matches the preceding element at least m and at most n times
  E.g., `.{2,4}` is $\underline{\Sigma\Sigma \mid \Sigma\Sigma\Sigma \mid \Sigma\Sigma\Sigma\Sigma}$
- `|` Normal "or"
  E.g., `abc|def` is $\underline{\texttt{abc} \mid \texttt{def}}$

Character classes

Character classes are shorthands for `[ ]` or `[^ ]` expressions

`[:alpha:]` Equivalent to `[A-Za-z]`

`[:digit:]` Equivalent to `[0-9]` (written `\d` in PCRE or Vim)

...

The POSIX ones (with the brackets and colons) must appear inside brackets

E.g., `[[:digit:]abc]` matches a digit or `a`, `b`, or `c`

Some common tools

- grep (or egrep): Selects lines that match a regex
  `egrep '((1-)?[0-9]{3}-)?[0-9]{3}-[0-9]{4}' file`
- awk (or gawk or mawk or nawk): Runs a program on lines that match
  `awk '/cat|hat/ { print $1, $3 }'`
- sed: Reads lines from files and applies commands
  `sed -E 's/([^,]*),(.*)/\2,\1/' file`

Programming language support

Built-in support

- Perl: `$foo =~ /foo|bar?/` or `$foo =~ s/red/blue/`
- Bash: `if [[ "$x" =~ foo|bar|baz ]]; then echo match; fi`
- Ruby: `'haystack' =~ /hay/`
- ...

Standard library support

- Python. `re` module has `re.compile('ab*')` and related functions
- C++11. `std::regex`
- ...

Languages without built-in support usually use strings for regex and this leads to lots of escaping: `/\d/` becomes `"\\\d"`

Match objects or variables

Usually, just matching a string isn't enough
We want to extract matching substrings and do something with them

Parentheses denote "capturing groups" and the text that matches the corresponding subexpression is available

- using special variables (like $1, $2, ...)

```
'foo␣bar␣baz' =~ /([^ ]+) ([^ ]+)/;
print "$1\n"; # prints foo
print "$2\n"; # prints bar
```

- via returned match object

```
>>> import re
>>> m = re.match(r'([^␣]+)␣([^␣]+)', "foo␣bar␣baz")
>>> m.group(1)
'foo'
>>> m.group(2)
'bar'
```

Much much more

There's a lot more than I've touched on

Read some of the documentation to see how best to use regex in your language of choice

Many popular regex implementations have extentions that allow the language to match strings from some nonregular languages

You cannot parse HTML with regular expressions!

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a bre*ach between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of re**gex parsers for HTML will ins**tantly transport a *programmer's consciousness i*nto a wo*rld* of ceaseless screaming, he comes, ~~the pestilent sli~~thy regex-infection will **devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fight* h**e comes, his** unholy radiancé d*estroying all enlightenment, HTML tags le~~a~~k~~ing from your eyes~~ *like liquid* pain, the song of regular expression parsing will ext*inguish the voices of mor***tal man from the s***phere I can see it can you see it it is beautiful* the f`inal snuf` fing of *the lie***s of Man ALL IS LOST ALL IS** LOST the *pony he comes* he comes he c~~omes~~ ***the***.ich~~or~~ permeat*es all* MY FACE *MY FACE oh god no* ***NO NOOOO*** NO stop *the anguish* .*are* **not real ZALGO IS** TONY THE **PONY, HE COMES**

Compiler construction

Compilers typically operate in phases

1. Lexical analysis (lexing or tokenizing) splits sequences of characters into tokens
2. Syntax analysis (parsing) generates a parse tree and checks that the program is syntatically correct (more on this later!)
3. Semantic analysis checks if the parse tree follows the rules of the language
4. Code generation and optimization (the bulk of the work of a compiler)

Lexing

Lexing splits a sequence of characters into tokens with types and values

Consider

```
int foo = 32;
```

This might be split into a sequence of tokens
⟨IDENTIFIER, “int”⟩, ⟨IDENTIFIER, “foo”⟩, ⟨EQUAL SIGN⟩, ⟨INTEGER, 32⟩, ⟨SEMICOLON⟩

The parsing stage might have a rule that says that a variable declaration consists of two identifiers, an equal sign, an expression, and a semicolon

The semantic analysis phase would check that the first identifier was a valid type and that the second identifier was a valid variable name, and that the expression was valid

Flex

Flex is a tool that is used to construct (usually C) source code to run as tokens are created

```
/* Definitions */
IDENTIFIER [A-Za-z_][A-Za-z0-9_]*
DIGIT      [0-9]

%%

/* Rules for what code to run when matching the
 * corresponding regular expression */
{DIGIT}+            { /* construct INTEGER token */ }
{DIGIT}+"."{DIGIT}* { /* construct FLOAT token */ }
{IDENTIFIER}        { /* construct IDENTIFIER token */ }
```

Implementing regular expression matching

Some options

- Table driven: convert to DFA and encode δ as a table
- Encode as loops and conditionals: convert to DFA but encode the transitions using control structures from the target language
- Backtracking: convert to NFA and employ a backtracking strategy if a choice was incorrect
- Brzozowski derivative (named for Janusz Brzozowski): for the first character t in the string, construct a new regular expression $t^{-1}R$ to match against the remaining characters, repeat