CSC236 winter 2020, week 9: Formal languages and regular expressions

Recommended reading: Chapter 7 Vassos course notes

Colin Morris
colin@cs.toronto.edu
http://www.cs.toronto.edu/~colin/236/W20/

March 9, 2020

Outline

Upcoming dates

Iterative correctness wrap-up

Formal languages

Regular expressions

Upcoming stuff

- A2 due Thurs 3pm
  - Extra office hours today 2-4pm. BA 3201
  - Extended office hours Wednesday
- Term test 2, next Monday (March 16)
  - 12:10-13:00 @ EX320
  - 13:10-14:00 @ EX200

Term test 2

Covers weeks 4-8 (same material as A2). Potential topics for questions:[1]

- Devise a recurrence for the runtime of an algorithm
- Use unwinding to find a closed form for a recurrence
- Use Master Theorem to reason about big-Θ of divide-and-conquer recurrences
- Come up with formal specifications for an algorithm
- Use induction to prove correctness of a recursive algorithm
- Identify and prove loop invariants
- Use loop invariants to prove partial correctness
- Prove that an iterative algorithm terminates

[1] Not exhaustive.

Term test 2

Prefab cheet sheet will be provided. Will have Master Theorem, and possibly more. e.g.

- Geometric series identities ($\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$)
- Big-Θ definition
- Brief reminder of ‘recipes’ for proofs of recursive correctness, partial correctness, termination, etc.

Will be posted to course website at least a couple days before test. (But not necessarily indicative of what questions will be on the test.)

‘Clamping’ invariants
(Example from week 7 quiz v2)

```
1 def mult(x, y):
2     """Pre: x and y are ints. y is non-negative.
3     Post: return x * y
4     """
5     p = 0
6     while y > 0:
7         p += x
8         y -= 1
9     return p
```

Working backwards. I want to say that if the loop exits, y will be 0.
What loop invariant will allow this?

inv: $y_i \in \mathbb{N}$ or $y_i \geq 0$

$y \leq 0$, by l6

Clamping invariants: another example

$a_j = a_0 - j$

$b_j = b_0 - j$

```
def geq(a, b):
  """Pre: a and b are positive integers
  Post: return True iff a >= b
  """
  while a > 0 and b > 0:
    a -= 1
    b -= 1
  return b == 0
```

inv: 1) $(a_j - b_j) = a_0 - b_0$

2) $a_j \in \mathbb{N}$ (or $a_j \geq 0$)

$b_j \in \mathbb{N}$

What should be true when the loop exits?

if $b_j = 0$, $(a_j - 0) = a_0 - b_0$

$a_j = a_0 - b_0 \Rightarrow a_0 - b_0 \geq 0$

$a_0 \geq b_0$

WTS: $a_0 \geq b_0$

Anti-pattern: invariants involving loop counter

```python
def geq(a, b):
  """Pre: a and b are positive integers
  Post: return True iff a >= b
  """
  while a > 0 and b > 0:
    a -= 1
    b -= 1
  return b == 0
```

At the end of each iteration j...

- $a_j = a_0 - j$
- $b_j = b_0 - j$

Not *wrong*, but can be simplified.

Related anti-pattern: counting exact number of iterations

Can work, but generally more work than necessary

```python
def f(n):
  """Pre: n is a positive integer.
  """
  a = b = 0
  while n > 0:
    if n % 2 == 1:
      n -= 1
      a += 1
    else:
      n = n // 2
      b += 1
  return (a, b)
```

$m_i = n_i$

How many times will this loop iterate for a given n?

- $\log n$?
- $\lceil \log n \rceil$?
- $\lfloor \log n \rfloor$ + number of 1's in binary representation of n?

A: who cares

Invariants that follow directly from code

```python
def geq(a, b):
  """Pre: a and b are positive integers
  Post: return True iff a >= b
  """
  while a > 0 and b > 0:
    a -= 1
    b -= 1
  return b == 0
```

At the end of each iteration j...

- $a_j = a_{j-1} - 1$, for $j > 0$

This is (mostly) true, but how can we prove it?

(If your loop invariant just says what the code does, you can probably omit it.)

Example correctness proofs

$m_j = 36_j + w_j$

$= 100 6_j + w_j$

m_j

- Writeup of merge correctness posted with lecture slides
- Sample solutions for tutorial exercises and quizzes
- Vassos notes
- A2Q3 appendix

Δp	Δn
+5	-2
4	-1
-2	0

Formal languages

The rest of the course will be spent studying sets of strings (**languages**).
Relevance to computer science?

- Strings can represent any data type.
    - Ultimately, your computer uses strings of 1's and 0's to represent numbers, lists, trees, cat pictures, etc.
- Sets of strings are a convenient way to formalize the concept of a 'problem' in theoretical computer science
    - P and NP are sets of *languages*
    - SAT: the set of strings which represent satisfiable propositional formulas

Formal languages

Heading towards big question of theoretical CS: which problems can be solved algorithmically?

- Specifically, we'll be focusing on the question: which problems can be solved with extremely limited RAM?
- Along the way, we'll need to develop abstract mathematical models for problems, and algorithmic processes

(Connection to computation will become much clearer next week.)

Definitions

- **Alphabet**: a set of symbols (usually finite), denoted by Σ
  - e.g. $\Sigma = \{0, 1\}$, $\Sigma = \{a, b, c, \ldots, z\}$
  - we'll generally avoid alphabet symbols that could cause confusion, such as ε, $*$, parentheses, spaces, etc.
- Σ^* is the set of all finite strings over alphabet Σ
  - e.g. $\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$
  - recall ε is the **empty string** (like `""` in Python)
- $L \subseteq \Sigma^*$ is a **language**
  - languages may be finite, e.g. $L = \{baa, baabaa\}$
  - …or infinite, e.g. $L = \{s \in \{0,1\}^* \mid s \text{ has same number of 0's as 1's}\}$
  - but the strings they contain are always finite
  - NB: $\{\} \neq \{\varepsilon\}$

String operations

Let s, t be strings over some alphabet Σ.

- st is the **concatenation** of s and t
  - occasionally also written $s \circ t$
- s^n is repeated concatenation. Defined recursively:
  - $s^0 = \varepsilon$
  - $s^{j+1} = ss^j$
- $|s|$ is the number of symbols in s. Note that $|\varepsilon| = 0$.
  - ($\text{len}(s)$ is fine too)

$(ab)^2 = abab$

Operations on languages

$L \cup L'$: union

$L \cap L'$: intersection

$L - L'$: difference

$\overline{L}$: Complement of L, i.e. $\Sigma^* - L$. If L is language of strings over $\{0, 1\}$ that start with 0, then $\overline{L}$ is the language of strings that begin with 1 plus the empty string.

LL': concatenation, i.e. $\{st \mid s \in L, t \in L'\}$.
(What happens when one of these is $\{\}$ or $\{\varepsilon\}$?)

L^k: concatenation of L with itself k times. $L^0 = \{\varepsilon\}$.

$\{a, b\}\{0\}$
$= \{a0, b0\}$

$\varepsilon \notin \{a, b\}$

$\{a, b, \varepsilon\}$

$\{0\}\{a, b\}$
$= \{0a, 0b\}$

Kleene star

A few equivalent definitions

$\{aa, bb\}^*$

L^* contains the strings formed by concatenating zero or more (not necessarily distinct) strings from L

$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$.

Define L^* recursively as the smallest set such that:

1. $\varepsilon \in L^*$
2. if $s \in L$, then $s \in L^*$
3. if $s \in L$ and $t \in L^*$, then $st \in L^*$

Example: Going from formal description to intuition

Let $L = \{aa, b\}$

Define EVENA $= \{s \in \{a, b\}^* \mid s$ has an even number of a's$\}$

EVENA $\stackrel{?}{=} L^*$

aaaabb

$aba \notin L^*$

$\in$ EVENA

aabaaaa

X "strings w/ even # of a's, and all a's are beside each other"

From intuition to formal description

Let BIN $\subseteq \{0, 1\}^*$ be the language of binary numbers (with no redundant leading zeros).

Give a formal definition of BIN using only:

- One or more finite languages
- Operations such as union, complement, concatenation, Kleene star, etc.

$\{1\}\{0,1\}^* \cup \{0\}$

From intuition to formal description

UNIFORM = { a, aa, b, bb, aaaa, bbbb, ...

Let UNIFORM $= \{s \in \{a, b\}^* \mid s$ consists of non-zero repetitions of a single symbol$\}$.
Give a formal definition of UNIFORM using only:

a^n or b^n, $n > 0$

- One or more finite languages $= a^n$
- Operations such as union, complement, concatenation, Kleene star, etc.

$$\left(\{a\}^* \cup \{b\}^*\right) - \{\varepsilon\}$$

Regular expressions

Same idea as before, more concise notation

BIN: $0 + (1(0+1)^*)$

UNIFORM: $(aa^*) + (bb^*)$

Not you...

How to validate an email address using a regular expression?

Asked 11 years, 4 months ago Active 1 month ago Viewed 1.2m times

3256

1286

Over the years I have slowly developed a regular expression that validates MOST email addresses correctly, assuming they don't use an IP address as the server part.

I use it in several PHP programs, and it works most of the time. However, from time to time I get contacted by someone that is having trouble with a site that uses it, and I end up having to make some adjustment (most recently I realized that I wasn't allowing 4-character TLDs).

What is the best regular expression you have or have seen for validating emails?

I've seen several solutions that use functions that use several shorter expressions, but I'd rather have one long complex expression in a simple function instead of several short expression in a more complex function.

regex validation email email-validation string-parsing

2397

+50

The fully RFC 822 compliant regex is inefficient and obscure because of its length. Fortunately, RFC 822 was superseded twice and the current specification for email addresses is RFC 5322. RFC 5322 leads to a regex that can be understood if studied for a few minutes and is efficient enough for actual use.

One RFC 5322 compliant regex can be found at the top of the page at http://emailregex.com/ but uses the IP address pattern that is floating around the internet with a bug that allows `00` for any of the unsigned byte decimal values in a dot-delimited address, which is illegal. The rest of it appears to be consistent with the RFC 5322 grammar and passes several tests using `grep -Po`, including cases domain names, IP addresses, bad ones, and account names with and without quotes.

Correcting the `00` bug in the IP pattern, we obtain a working and fairly fast regex. (Scrape the rendered version, not the markdown, for actual code.)

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

* + . (?)

? | [a-z]

Theory vs. practice

The regular expressions we'll be studying have a close connection to 'regular expressions' in software (e.g. `grep`, or Python's `re` library), but there are significant differences.

- Software implementations come packed with *lots* of extra features and syntax (many different dialects)
  - Docs for Python's `re` library $\approxeq$ 9,000 words
  - Useful for programmers, bad for theoreticians
  - Some extra features increase expressive power, allowing matching languages which aren't truly 'regular' (more on what this means later)

Our RE syntax will be *very* simple

RE syntax

$\{0,1\}^* = L((0+1)^*)$ $L((0+1)) = \{ \}$

Recursive definition of $\mathcal{RE}$, the set of regular expressions over some alphabet Σ:

1. $\emptyset, \varepsilon \in \mathcal{RE}$ ε $\emptyset$
2. every symbol in Σ is in $\mathcal{RE}$
3. if T and S are REs, then so are: 0 1
   - 3.1 $(T + S)$ (union) — lowest precedence operator
   - 3.2 (TS) (concatenation) — middle precedence operator
   - 3.3 T^* (star) — highest precedence

The precedence rules allow us to make our REs more readable. e.g. we can write

$(a+b)b^* \neq$

$$a + bb^*$$

instead of

$(a + (b \times c))$

$a + b \times c$

$(a+b) \times c$

$$(a + (b(b^*)))$$

RE semantics $R = (0+1)$

$L(R) = \{0, 1\}$

$L(010) = \{010\}$

$\mathcal{L}(R)$ denotes the language represented by regex R.

Base cases:

- $\mathcal{L}(\emptyset) = \emptyset = \{\}$
  - (rarely used, but needed for completeness)
- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\mathcal{L}(a) = \{a\}$
  - where a is an arbitrary length-one string from our alphabet Σ

$L(a(b+c)) = \{ab, ac\}$

$abc \notin$

Constructor cases. For regular expressions S, T:

- $\mathcal{L}(S + T) = \mathcal{L}(S) \cup \mathcal{L}(T)$
  - 'take *either* S or T'
- $\mathcal{L}(ST) = \mathcal{L}(S)\,\mathcal{L}(T)$
- $\mathcal{L}(T^*) = \mathcal{L}(T)^*$
  - '0 or more repetitions of T'

$L(R) = L(0) \cup L(1)$

$= \{0\} \cup \{1\}$

$= \{0, 1\}$

$(00+11)$

$L((00+11)^*) = \{00, 11\}^* = \{\varepsilon, 00, 11, 1100, 1111$

RE identities

Most of these follow from definition. Some require proof. See 7.2.4 in Vassos notes.

- communitativity of union: $R + S \equiv S + R$
- associativity of union: $(R + S) + T \equiv R + (S + T)$
- associativity of concatenation: $(RS)T \equiv R(ST)$
- left distributivity: $R(S + T) \equiv RS + RT$
- right distributivity: $(S + T)R \equiv SR + TR$
- identity for union: $R + \emptyset \equiv R$
- identity for concatenation: $R\varepsilon \equiv R \equiv \varepsilon R$
- annihilator for concatenation: $\emptyset R \equiv \emptyset \equiv R\emptyset$
- idempotence of Kleene star: $(R^*)^* \equiv R^*$

Examples revisited

BIN: $0 + (1(0+1)^*)$

UNIFORM: $(aa^*) + (bb^*)$

$aa^* + bb^*$

More examples

$\{0,1\}^3$

$\underline{(0+1)^{3}}$ X not allowed in RE syntax

$000 + 001 + 010$

Devise REs (over $\Sigma = \{0, 1\}$) that represent

- all strings of length 3 $(0+1)(0+1)(0+1)$
- all strings of length 3 or 4 $(0+1)(0+1)(0+1)(\epsilon+0+1)$
- strings that start and end with 0 $0(0+1)^*0 + 0$
- 'sorted' strings (0's appear before 1's) $0^* 1^*$
- strings of the form 0^n1^n (sorted and balanced strings)

More examples

Devise REs (over $\Sigma = \{0, 1\}$) that represent

- all strings of length 3
- all strings of length 3 or 4
- strings that start and end with 0
- 'sorted' strings (0's appear before 1's)
- strings of the form $0^n 1^n$ (sorted and balanced strings)

Big question

$0^n 1^n$

For every language L, does there exist an RE R such that $\mathcal{L}(R) = L$?

How would we prove that a given language can't be represented by any RE?

We'll need another tool.

Proving RE equivalence

$A = B$

$A \subseteq B$

$B \subseteq A$

$$S = (a+b)^* a (a+b)^* b (a+b)^*$$
$$T = (a+b)^* ab (a+b)^*$$

Show[2] that S and T are **equivalent** – i.e. $\mathcal{L}(S) = \mathcal{L}(T)$.

Proof sketch:
In general, to prove two sets are equal, I need to show mutual inclusion.
Is L(T) a subset of L(S)? Given a string generated by T, I can also generate it from S by setting the middle (0+1)* to the empty string.
Is L(S) a subset of L(T)?
By inspection, I can see that L(T) is the set of all strings containing 'ab'.
Therefore, it suffices to show that every string generated by S contains 'ab'.
Let s be an arbitrary string generated by S. Then s is of the form s_1 a s_2 b s_3, where s_1, s_2, and s_3 are each arbitrary strings of a's and b's.
I can show that s always contains substring 'ab' based on the following cases for s_2:
if s_2 is empty
if s_2 starts with b
if s_2 ends with a
if s_2 starts with a and ends with b

[2] We could **prove** this from first principles, but it would be tedious. We'll generally ask for only an informal proof for problems like this.