Lecture 11: Regular Expressions

Regular expressions
Automata
Recursive backtracking
Upcoming important dates

Midterm in class this Wednesday

Sections on Thursday as usual

PA3 is out and will be due on 2/23

Work in teams of 3

Makeup lecture next Friday, 2/19

We will record it better than last time
Where are we?

We have seen two ways to build parse trees
  CYK and Earley parsers

Today, we’ll learn about REs and backtracking parsers
  Regular expressions and their compilation to automata
  Regexes and their implementation using backtracking
Puzzle 1

Write a regex that tests whether a number is prime.

Hint:
\n matches the string matched by the \n th regex group
Ex: regex c(aa|bb)\1 matches strings caaaa and cbbbbb

the prime tester must be a regex, not a regular expression!
the latter do not support \n
Sample Uses of Regular Expressions and of string processing in general
Sample string processing problems

Web scraping and rewriting
  collect data from web pages; “linkify” mailing addresses

Cucumber, a Ruby testing framework with “NLP”
  When I go to "Accounts" Then I should see link "My Savings"

Lexical analysis in interpreters and compilers
  float x=3.14 --> FLOATTYPE, ID("x"), EQ, FLOATLIT(3.14)

Config tools include “file name manipulation” languages
  ${dir}\foo --> "c:\MyDir\foo"

Editors, search and replace
  `quoted text’ D’Souza --> `quoted text’ D’Souza
1. Web scraping and rewriting

Rewrite a web page using GreaseMonkey scripting.

**The idea:** web pages are more readable when you view the print-friendly, ad-free pages. Automate this!

**Approach:** Your script will rewrite links on a page to point to the print-friendly version of target page.

**How:** When user clicks on a link, fetch (but do not display) the target page; use a regex to find in the target HTML text the (best-guess) link to the print-friendly page; rewrite the link to point to that page; follow the rewritten link to display the friendly page.
2. Cucumber: a Ruby testing framework

A sample Cucumber test file:

**Scenario:** Test the banking web service

*Given* I log in as "bonnie" with password "clyde"

*When* I go to "Accounts"

*Then* I should see a link "Our Robbery Savings"

*When* I follow this link

*Then* I the value of "Interest" should be "$1,024.00"

Meaning of this test:

"Given" makes the script go to login to the web site.

"When" clause clicks on the link Accounts.

"Then" clause tests that resulting page contains a link to given account.
3. Lexical analysis in a compiler/interpreter

**Input:** a program

```javascript
function timedCount() { // my function
document.getElementById('txt').value=c;
}
```

**Output:** a sequence of tokens

```
FUNCTION, ID("timedCount"), LPAR, RPAR, LCUR,
ID("document"), DOT, ID("getElementById"), LPAR,
STRING("txt"), RPAR, DOT, ID("value"), ASGN, ID("c"),
SEMI, RCUR
```

REs facilitate concise description of tokens
Notes on lexical analysis

The lexer partitions the input into lexemes

Lexemes are mapped to tokens

The stream of tokens is fed to the parser

Some tokens are associated with their lexemes

Whitespace and comments are typically skipped
Puzzle 2

Ideally, the lexical analyzer produces a list of tokens without consulting the parser

we want tokenizing to be syntax-insensitive, i.e., can be performed prior to parsing

Q: Give a JavaScript expression $E$ whose tokenization depends on the context, on where $E$ appears

i.e., lexer cannot tokenize input w/out feedback from parser
4. File name processing languages

In shell scripts and IDEs, command line args can refer to variables such as `workspace_loc`:

```
args = { "--inputDir", "c:\\wspace\\MyDirectory", "--outputDir", ".\\temp dir" }
```

This is translated into program arguments:

```
--inputDir $(workspace_loc)\\MyDirectory --outputDir ".\\temp dir"
```

Must escape `\` and quotes during translation

Tricky design. Eclipse designed this substitution language wrong: their escaping rules prevent you from expressing some values that you may want to pass into the program.
5. Search for strings in text editors

Imagine you want to search for names containing a ' and correct them. Examples:

D'Souza --&gt; D’Souza
D`Souza --&gt; D’Souza

The challenge: your replacement should not change quotes that delineate quotations, such as `quoted text`

Again, this can be solved conveniently with REs
Useful string processing operations

Accept: the whole string match

- Does the entire string $s$ match a pattern $r$?

Match: a prefix match

- Does some prefix of $s$ match a pattern $r$?

Search: find a substring

- Does a substring of $s$ match a pattern $r$?

Tokenizer: Lexical analysis

- Partition $s$ into lexemes, each accepted by a pattern $r$

Extract: as match and search but extract substrings

- Regex $r$ indicates, with ( ), which substrings to extract

Replace: replace substrings found with a new string
Discovering automata and REs
a revisionist history
Programming model of a small language

Design of a small (domain-specific) language starts with the choice of a suitable programming model

Why a new model may be needed? Why can’t use procedures? Because procedural abstraction (a procedure-based library) cannot always hide the plumbing (implementation details)

For string-based processing, automata and regular expressions are usually the right programming model

regexes are hard to hide under a procedure library, although we have seen how to do it with coroutines.

Let’s use string processing as a case study on how to discover the programming model for small languages
Let's write a lexer (a.k.a. tokenizer, scanner)

First, we’ll write the scanner by hand

- We'll examine the scanner’s repetitious plumbing
- Then we'll hide the plumbing in a programming model

A simple scanner will do. Only four tokens:

<table>
<thead>
<tr>
<th>TOKEN</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>a sequence of one or more letters or digits starting with a letter</td>
</tr>
<tr>
<td>EQUALS</td>
<td>“==“</td>
</tr>
<tr>
<td>PLUS</td>
<td>“+“</td>
</tr>
<tr>
<td>TIMES</td>
<td>“*“</td>
</tr>
</tbody>
</table>
Operational scanner

```c
if (c == '=') {
    c = nextChar();
    if (c == '=') { return EQUALS; }
}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
    c = nextChar();
    while (c is a letter or digit) {
        c = nextChar();
    }
    undoNextChar(c);
    return ID;
}
```

Note: this scanner does not handle errors. What happens if the input is "var1 = var2"? It should be var1 == var2. An error should be reported at around '='.
Imperative scanner

You could write your entire scanner in this style
   - and for small scanners this style is appropriate

Why does this code break as the task gets bigger? Try to add:
   - lexemes that start with the same string: “if” and “iffy”
   - C-style comments: /* anything here */ and nested comments */ */
   - string literals with escape sequences: “...\t... \”...”
   - error handling, e.g., “a string literal missing a closing quote

Real-world imperative scanners can get unwieldy
   - the lexical structure of the language may be hard to read out
   - the scanner code obscures it by spreading the string comparisons
     and other actions across the scanner code (rather than keeping it in
     a single specification table)
Real scanners in this style get unwieldy

```c
/* Scan XML comment. */
if (MatchChar(ts, '-') ) {
    if (!MatchChar(ts, '-') )
        goto bad_xml_markup;
    while ( ((c = GetChar(ts)) != '-' || !MatchChar(ts, '-') ) ) {
        if (c == EOF)
            goto bad_xml_markup;
        ADD_TO_TOKENBUF(c);
    }
    tt = TOK_XMLCOMMENT;
tp->t_op = JSOP_XMLCOMMENT;
    goto finish_xml_markup;
}

/* Scan CDATA section. */
if (MatchChar(ts, '[')) {
    jschar cp[6];
    if (PeekChars(ts, 6, cp) &&
        cp[0] == 'C' &&
        cp[1] == 'D' &&
        cp[2] == 'A' &&
        cp[3] == 'T' &&
        cp[4] == 'A' &&
        cp[5] == '[') {
        SkipChars(ts, 6);
        while ( ((c = GetChar(ts)) != ']' ||
            !PeekChars(ts, 2, cp) ||
            cp[0] != ']' ||
            cp[1] != '>') ) {
            if (c == EOF)
                goto bad_xml_markup;
            ADD_TO_TOKENBUF(c);
```

From http://mxr.mozilla.org/mozilla/source/js/src/jsscan.c
c=nextChar();
if (c == '=' ) { c=nextChar(); if (c == '=' ) {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) {  c=NextChar(); }
    undoNextChar(c);
    return ID;
}
Imperative Lexer: what vs. how

c=nextChar();
if (c == '='='') { c=nextChar(); if (c == '='='') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter)
    c=NextChar();
    while (c is a letter or digit) {  c=NextChar(); } undoNextChar(c); return ID;

little logic, much plumbing
Identifying the plumbing (the how, part 1)

c = nextChar();
if (c == '=' ) {
  c = nextChar();
  if (c == '=') {
    return EQUALS;
  }
}
if (c == '+') {
  return PLUS;
}
if (c == '*') {
  return TIMES;
}
if (c is a letter) {
  c = NextChar();
  while (c is a letter or digit) {
    c = NextChar();
  }
  undoNextChar(c);
  return ID;
}

Characters are read always the same way
c=nextChar();
if (c == '="') {
    c=nextChar();
    if (c == '="') {
        return EQUALS;
    }
}
if (c == '+') {
    return PLUS;
}
if (c == '*') {
    return TIMES;
}
if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) {
        c=NextChar();
    }
    undoNextChar(c);
    return ID;
}

.tokens are always return-ed
Identifying the plumbing (the **how**, part 3)

```c
    c=nextChar();
    if (c == '="') { c=nextChar(); if (c == '=' ) { return EQUALS;}}
    if (c == '+') { return PLUS; }
    if (c == '*') { return TIMES; }
    if (c is a letter) {
        c=NextChar();
        while (c is a letter or digit) { c=NextChar(); }  
        undoNextChar(c);
        return ID;
    }

    the lookahead is explicit (programmer-managed)
```
c = nextChar();
if (c == '=') {
    c = nextChar();
    if (c == '=') { return EQUALS; }
}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
    c = NextChar();
    while (c is a letter or digit) {
        c = NextChar();
    }
    undoNextChar(c);
    return ID;
}

must build decision tree out of nested if’s
Can we hide the plumbing?

To summarize, we want to avoid the following

- if’s and while’s to construct the decision tree
- calls to the read method
- explicit return statements
- explicit lookahead code

Ideally, we want code that looks like the specification:

<table>
<thead>
<tr>
<th>TOKEN</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>a sequence of one or more letters or digits starting with a</td>
</tr>
<tr>
<td></td>
<td>letter</td>
</tr>
<tr>
<td>EQUALS</td>
<td>“==“</td>
</tr>
<tr>
<td>PLUS</td>
<td>“+“</td>
</tr>
<tr>
<td>TIMES</td>
<td>“*“</td>
</tr>
</tbody>
</table>
Separate out the how (plumbing)

Luckily, the plumbing follows a regular pattern:

- read next char,
- compare it with some predetermined char
- if matched, jump to the next read of next char
- repeat this until a lexeme is built; then return a token.

What’s a programming model for encoding this?

- finite-state automata
- state corresponds to history of what we have read
- finite: number of states is fixed, i.e., input independent
Separate out the what

c = nextChar();
if (c == '=') { c = nextChar();
if (c == '=') { return EQUALS; }}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
c = nextChar();
while (c is a letter or digit) {
c = nextChar();
}
undoNextChar(c);
return ID;
}

Here is the automaton; we’ll refine it later
A declarative scanner

Part 1: declarative (the what)

describe each token as a finite automaton
this specification must be supplied for each scanner, of course
(it specifies the lexical properties of the input language)

Part 2: operational (the how)

connect these automata into a scanner automaton
common to all scanners (like a library),
responsible for the mechanics of scanning
Automata are hard to draw. Need a notation.

For convenience and clarity, we want text notation

Kleene invented regular expressions for the purpose:

- $a \cdot b$  
  a followed by b (sometimes written $a \cdot b$)
- $a^*$  
  zero or more repetitions of a
- $a \mid b$  
  a or b

Our example: $ab^*c$
Regular expressions

Regular expressions contain:

- characters: these must match the input string
- meta-characters: these serve as operators (*, |, [,], etc)

Operators operate on REs (it’s a recursive definition)

- `char` any character is a regular expression
- `r_1 r_2` so is `r_1` followed by `r_2`
- `r*` zero or more repetitions of `r`
- `r_1 | r_2` match `r_1` or `r_2`

- `r+` one or more instances of `r`, desugars to `rr*`
- `[1-5]` same as `(1|2|3|4|5)`; `[]` denotes a character class
- `[^a]` any character but a
- `\d` matches any digit
- `\w` matches any letter
One could invent a ton of other meta chars


...  
1957 - John Backus and IBM create FORTRAN. There's nothing funny about IBM or FORTRAN. It is a syntax error to write FORTRAN while not wearing a blue tie.  

...  
1964 - John Kemeny and Thomas Kurtz create BASIC, an unstructured programming language for non-computer scientists.  
1965 - Kemeny and Kurtz go to 1964.  

...  
1987 - Larry Wall falls asleep and hits Larry Wall's forehead on the keyboard. Upon waking Larry Wall decides that the string of characters on Larry Wall's monitor isn't random but an example program in a programming language that God wants His prophet, Larry Wall, to design. Perl is born.
Two flavors of regular expressions
(RE vs. regex)
Compare performance of RE in three languages

Consider regular expression \( X(\ .\ +)\+X \)

Input1: 
\"X=================================X\" match

Input2: 
\"X==================================\" no match

JavaScript: 
\"X====...==========X\".search(/X(\ .\ +)\+X/)  
– Match: fast  No match: slow

Python:  
re.search(r'X(\ .\ +)\+X','=XX========...=====X=')  
– Match: fast  No match: slow

awk:  
echo '=XX========...=====X=' | awk '/X(\ .\ +)\+X/'  
– Match: fast  No match: fast

Sidenote: compare how the three languages integrate regular expressions.
This problem occurs in practice: HW1 fall 2010

**Problem:** Find mailing addresses in HTML and wrap them in links to google maps.

**From the course newsgroup:** “I had been experiencing the same problem -- some of my regex would take several minutes to finish on long pages.

```
/((\w*\s*)*\d*)*Hi There/ times out on my Firefox.

/Hi There((\w*\s*)*\d*)/*/ takes a negligible amount of time.
```

It is not too hard to see why this is.

To fix this, if you have some part of the regex which you know must occur and does not depend on the context it is in (in this example, the string "Hi There"), then you can grep for that in the text of the entire page very quickly. Then gather the some number of characters (a hundred or so) before and after it, and search on that for the given regex. ...

I got my detection to go from several minutes to a second by doing just the first.”
Why do implementations differ?

Some are based on **backtracking** (can be slow)

to conclude that \(X=\cdots\) does not match \(X(\ast)^*X\),
backtracking needs to try all ways to match the string,
including: 
\((==)(===)(=)\ldots\) and 
\((=)(=)(==)\ldots\) and ...

Some are based on **automata**

automata can keep track of all possible matches at once

There are semantic differences between the two!

see the section later in this lecture
Implementing RE as an automaton
Finite-state automata (recall from 311)

An automaton reads an input string and accepts or rejects the string.

It does so by transitioning from state to state on each character.

It starts in the (unique) start state.

It accepts the string if on consuming the whole string, the automaton is in a dedicated final state.
Visual notation

- A state

- The start state

- A final state

- A transition
Finite Automata

Transition

\[ s_1 \xrightarrow{a} s_2 \]

Is read

In state \( s_1 \) on input “a” go to state \( s_2 \)

String accepted if

entire string consumed and automaton is in accepting state

Rejected otherwise. Two possibilities for rejection:
  - string consumed but automaton not in accepting state
  - next input character allows no transition (stuck automaton)
A finite automaton is a 5-tuple \((\Sigma, Q, \Delta, q, F)\) where:

- \(\Sigma\): an input \textbf{alphabet}
- \(Q\): a set of \textbf{states}
- \(q\): a \textbf{start state} \(q\)
- \(F\): a set of \textbf{final states} \(F \subseteq Q\)
- \(\Delta\): a state transition function: \(Q \times \Sigma \rightarrow Q\) (i.e., encodes transitions \textbf{state} \rightarrow \textbf{input state})
Our use of automata

We’ll use automata as recognizers:
- recognizer accepts a set of strings, and rejects all others

An automaton will tell us if a string is a valid lexeme
- Example: an automaton for identifiers accepts “xyx” but rejects “+3e4”
Finite automata, in more detail

Deterministic (DFA):
- state transition unambiguously determined by the input
- more efficient implementation

Non-deterministic (NFA):
- state transition determined by the input and an oracle
- less efficient implementation

NFAs have a composability property
which we’ll use to compile REs to automata
Deterministic Finite Automata

Example: JavaScript Identifiers

sequences of 1+ letters or underscores or dollar signs or digits, starting with a letter or underscore or a dollar sign:

```
letter | _ | $
```

```
letter | _ | $ | digit
```

Diagram:

```
S --> A
letter | _ | $ --> A
letter | _ | $ | digit
```


Example

Q: What does this DFA recognizes?
A: Integer literals
   with an optional + or - sign:
And another (more abstract) example

- Alphabet \( \{0, 1\} \)
- What strings does this recognize?
Language defined by DFA

The language defined by a DFA is the set of strings accepted by the DFA.

in the language of the identifier DFA shown above:
  x, tmp2, XyZzy, position27.

not in the language of the identifier DFA shown above:
  123, a?, 13apples.
NFAs
Deterministic vs. Nondeterministic Automata

Deterministic Finite Automata (DFA)
- in each state, at most one transition per input character
- no \( \varepsilon \)-moves: each transition consumes an input character

Nondeterministic Finite Automata (NFA)
- allows multiple outgoing transitions for one input
- can have \( \varepsilon \)-moves

Finite automata need finite memory
- we only need to encode the current state

NFA’s can be in multiple states at once
- It’s still a finite (and fixed) set of states
A simple NFA example

Alphabet: { 0, 1 }

Nondeterminism:
when multiple choices exist, automaton “magically”
guesses which transition to take so that the string can be accepted (if it is possible to accept the string)

Example:
on input “11” the automaton could be in either state
Epsilon Moves

Another kind of transition: $\varepsilon$-moves

The automaton is allowed to move from state A to state B without consuming an input character.
Execution of Finite Automata (DFA)

A DFA can take only one path through the state graph – completely determined by input

Implementation: table-based

```plaintext
nextState := transitions[currentState, nextChar]
```
Execution of Finite Automata (NFA)

NFAs can choose

- whether to make ε-moves
- which of multiple transitions for a single input to take

We can think of NFAs in two alternative ways:

1) the choice is determined by an oracle
   the oracle makes a clairvoyant choice (looks ahead into the input)
2) NFAs are in several states at once (see next slide)
   these states correspond to all possible past oracle choices

We can emulate NFA

Keep track of current states. $O(NS)$ time. $S=$#states

Or we can convert NFA to DFA.

$O(N)$ matching time. But the DFA can be $2^S$ in size.
Acceptance of NFAs

An NFA can get into multiple states

Rule: NFA accepts if it can get into a final state
i.e., there is a path from start to end labeled with the input

Input: 1 0 1
NFA vs. DFA (1)

NFA’s and DFA’s are equally powerful

   each NFA can be translated into a corresponding DFA
      one that recognizes same strings

   NFAs and DFAs recognize the same set of languages
      called regular languages

NFA’s are more convenient ...

   – allow composition of automata

... while DFAs are easier to implement, faster

   – there are no choices to consider
   – hence automaton always in at most one state
NFA vs. DFA (2)

For a given language the NFA can be simpler than a DFA

DFA can be exponentially larger than NFA

e.g., when the NFA is translated to DFA
Translating an NFA to DFA

**Key idea:** NFA can be in multiple states at once.

“The blue tokens can be in any subset of NFA states.”

Each such subset is called a configuration.

Let’s create a DFA with a state for each configuration.

There are $2^N$ such states.

How do we place transitions in this DFA?

If char ‘a’ moves tokens from NFA states 5, 7 into **exactly** states 9, 15, we add a-labeled edge between these configurations.
Compiling Regular Expressions to NFAs
Two theorems from 311

• Fact 1: For every NFA there is a DFA that recognizes exactly the same language

• Fact 2: A language is recognized by a DFA if and only if it has a regular expression

• Now we will discuss how to convert a regular expression into NFA
Regular Expressions to NFA (1)

For each kind of rexp, define an NFA

- Construct NFA for its constituents
- Notation: NFA for rexp M

Two base cases

For $\varepsilon$:

For literal character $a$:
Regular Expressions to NFA (2)

For $A \ B$

For $A \ | \ B$
Regular Expressions to NFA (3)

For $A^*$
Example of RegExp $\Rightarrow$ NFA

Consider the regular expression

$$(1|0)^*1$$

The NFA is
Example of RegExp → NFA

Consider the regular expression

\[(1|0)^*1\]

The NFA is
Example of RegExp $\rightarrow$ NFA

Consider the regular expression $(1|0)^*$

The NFA is
Example of RegExp $\Rightarrow$ NFA

Consider the regular expression

$$(1|0)^*1$$

The NFA is

Can we do this systematically?
RegExp $\rightarrow$ NFA: A two steps algorithm

Step 1: compile RE to AST
Step 2: walk over the AST and generate an NFA

RE: $(a(b)^*)^*$

AST:

```
  *
  /
  seq
  /   *
 /     /
a     b
```
Another example of AST for a RE

\((ac|de^*)+\)
SDT to obtain AST of an RE

%ignore /\n+/

%%

// A regular expression grammar

R -> 'a' %{ return ('prim', n1.val) }%
  | R R %{ return ('seq', n1.val, n2.val) }%
  | R '*' %{ return ('star', n1.val) }%
  | R '|' R %{ return ('alt', n1.val, n3.val) }%
  | '(' R ')' %{ return n2.val }%
;

Exercise:
check for ambiguity
Final step: compiling RE to NFA

Recursively rewrite AST into automaton

\[(1|0)^*1\]
Final step: compiling RE to NFA

Recursively rewrite AST into automaton

\[(1|0)^*1\]
Final step: compiling RE to NFA

Recursively rewrite AST into automaton

(1|0)*1
SDT that visualizes RE-to-NFA translation

SDT can translate \((1|0)^*1\) not only to RE but also to a dotty file that visualizes this RE.

Dotty file:

```dotty
digraph G {
  f7 -> f8 [label="a"]
  f7 [label=""]
  f8 [label=""]f9 -> f10 [label="b"]
  f9 [label=""]
  f10 [label=""]
  f11 -> f9 [label=""]
  f10 -> f12 [label=""]
  f11 -> f12 [label=""]
  f12 -> f11 [label=""]
  f11 [label=""]
  f12 [label=""]f13 -> f14 [label="c"]
  f13 [label=""]
  f14 [label=""]
  f15 -> f13 [label=""]
  f14 -> f16 [label=""]
  f15 -> f16 [label=""]
  f16 -> f15 [label=""]
  f15 [label=""]
  f16 [label=""]
  f17 -> f11 [label=""]
  f17 -> f15 [label=""]
  f12 -> f18 [label=""]
  f16 -> f18 [label=""]
  f17 [label=""]
  f18 [label=""]
  f19 -> f7 [label=""]
  f8 -> f17 [label=""]
  f18 -> f20 [label=""]
  f19 [label=""]
  f20 [label=""]}
```
Answer to 2\textsuperscript{nd} challenge question

Q: Give a JavaScript scenario where tokenizing depends on the context of the parser. That is, lexer cannot tokenize the input entirely prior to parsing.

A: In this code fragment, // could be div’s or a regex:

```
ID Regex literal ← incorrect
e / f / g

ID DIV ID DIV ID ← correct
```

parser needs to tell lexer that at $\Delta$, an op is expected, not an expression
Implementation via backtracking
Implementation with btracking via coroutines

Step 1: Use SDT to compile RE to AST
Step 2: Walk over the AST and generate code that invokes a regex library

RE:      (a(b)*)*
AST:
Generated code:

    match(s,star(seq(prim("a"),star(prim("b"))))))
Generate code w/out intermediate AST

%ignore /\n+/

%%

// A regular expression grammar

R -> 'a'       %{ return 'prim("%s")' % n1.val %}
  | R R   %dprec 2 %{ return 'seq(%s,%s)' % (n1.val, n2.val) %}
  | R '*'  %dprec 1 %{ return 'star(%s)' % n1.val %}
  | R '|' R  %dprec 3 %{ return 'alt(%s,%s)' % (n1.val,n3.val) %}
  | '(' R ')'  %{ return n2.val %}
;

Regex matching

The goal is to decide if the regex matches a string.

Pattern ("abc" | "de")."x" can be defined as follows:

\[
\text{patt} = \text{seq}(\text{alt}(\text{prim("abc")}, \text{prim("de")}), \text{prim("x")})
\]

which effectively encodes the pattern’s AST.

seq, alt, prim are implemented with coroutines.
And now the main match routine

def match(S, patt):
    def m = coroutine.wrap(lambda(): { patt(S, 0) })
    for (pos in m):
        if (pos == len(S)):
            return true
    return false

match("de", alt(prim("abc"), prim("de")))
    --> true
Regex matching with coroutines

-- matching a string literal (primitive goal)

```python
def prim(str) {
    lambda(S,pos) {
        def len = len(str)
        if (sub(S,pos,pos+len-1)==str) {
            yield(pos+len)
        }
    }
}
```

-- alternative patterns (disjunction)

def alt(patt1,patt2) {
    lambda(S,pos) {
        patt1(S,pos);
        patt2(S,pos)
    }
}

-- sequence of sub-patterns (conjunction)

def seq(patt1,patt2) {
    lambda(S,pos) {
        def btpoint=coroutine.wrap(function(){ patt1(S,pos) })
        for npos in btpoint { patt2(S,npos) }
    }
}
Semantic differences between regexes and REs
Semantic differences in regexes and REs

It seems that regexes and REs are equivalent just like DFAs and NFAs are equivalent.

This is not true in all situations even if you restrict yourself to |, *, . ie you omit back-references \1, \2, ...

We will see that a pattern such as a|ab* can be interpreted differently by REs and regexes.
First, performance differences

Regexes are implemented with backtracking

This regex requires exponential time to discover that it does not match the input string X=============.

regex: X(.+)X

REs are implemented by translation to NFA

NFA may be translated to DFA.
Resulting DFA requires linear time, ie reads each char once
Now, the String Match Problem

Consider the problem of detecting whether a pattern (regex or RE) matches an (entire) string

\[ \text{match(string, pattern)} \rightarrow \text{yes/no} \]

The regex and RE interpretations of any pattern do agree on this problem.

That is, both give same answer to this Boolean question

Example: \( X(.+)+X \)

It does not matter whether this regex matches the string \( X===X \) with \( X(\.)(\.)X \) or with \( X(\.)(\.)(\.)X \), assigning different values to the ‘+’ in the regex. While there are many possible matches, all we care about is whether any match exists.
Let’s now focus on when regex and RE differ

Can you think of a question that where they give a different answer?

Answer: find a substring
Imagine you want to parse a config file:

```
filesToCompile=a.cpp b.cpp
```

The regex for this command line format:

```plaintext
[a-zA-Z]+=.*
```

Now let’s allow an optional \n-separated 2nd line:

```plaintext
filesToCompile=a.cpp b.cpp \\n\nd.cpp e.h
```

We extend the original regex correspondingly:

```plaintext
[a-zA-Z]+=.*(\\n.*)?
```

This regex does not match our two-line input. Why?

```plaintext
.* matches the \n
```
What compiler textbooks don’t teach you

The textbook string matching problem is simple:

Does a regex \( r \) match the entire string \( s \)?

– a clean statement suitable for theoretical study
– here is where regexes and FSMs are equivalent

In real life, we face the sub-string matching problem:

Given a string \( s \) and a regex \( r \), find a substring in \( s \) matching \( r \).

- tokenization is a series of substring matching problems
Substring matching: careful about semantics

Do you see the language design issues?

– There may be many matching substrings.
– We need to decide **which** substring to return.

It is easy to decide where this substring should **start:**

– the matched substring should be the **leftmost** match

RE and regex differ in where the string should **end:**

- longest match, first match, shortest match, best match
- if best, how do we define “best”?
Where should the matched string end?

**Declarative approach:** longest of all matches
- conceptually, enumerate all matches and return longest

**Operational approach:** define behavior of *, | operators
- e* match e as many times as possible while allowing the remainder of the regex to match (greedy semantics)
- e|e select leftmost choice while allowing remainder to match

```
filesToCompile=a.cpp b.cpp \n> d.cpp e.h
```

```
[a-zA-Z]+ = .* ( \ \ \n \n . * ) ?
```
These are important differences

We saw a non-contrived regex can behave differently
  – personal story: I spent 3 hours debugging a similar regex
  – despite reading the manual carefully

The (greedy) operational semantics of *
  – does not guarantee longest match (in case you need it)
  – forces the programmer to reason about backtracking

It may seem that backtracking is nice to reason about
  – because it’s local: no need to consider the entire regex
  – cognitive load is actually higher, as it breaks composition
Where in history of RE did things go wrong?

It’s tempting to blame perl

– but the greedy regex semantics seems older
– there are other reasons why backtracking is used

Hypothesis 1: creators of re libs didn’t know that NFA

– can be the target language for compiling regexes
– finds all matches simultaneously (no backtracking)
– Can be implemented efficiently (convert NFA to DFA)

Hypothesis 2: their hands were tied

– Ken Thompson’s algorithm for re-to-NFA was patented

With backtracking came the greedy semantics

– longest match would be expensive (must try all matches)
– so semantics was defined greedily, and non-compositionally
Regular Expressions Concepts

• Syntax tree-directed translation (re to NFA)
• recognizers: tell strings apart
• NFA, DFA, regular expressions = equally powerful
• but \1 (backreference) makes regexes more powerful
• Syntax sugar: e+ to e.e*
• Compositionality: be weary of greedy semantics
• Metacharacters: characters with special meaning
Summary of DFA, NFA, REs

What you need to understand and remember

– what is DFA, NFA, regular expression
– the three have equal expressive power
– what is meant by the “expressive power”
– you can convert

  • RE $\rightarrow$ NFA $\rightarrow$ DFA
  • NFA $\rightarrow$ RE

  and hence also DFA $\rightarrow$ RE, because DFA is a special case of NFA
– NFAs are easier to use, more costly to execute

  • NFA emulation $O(S^2)$-times slower than DFA
  • conversion NFA$\rightarrow$DFA incurs exponential cost in space
Puzzle 1: (An inefficient) Primality Test
Intermission: Answer to Primality Test puzzle

First, represent a number n as a unary string:

\[ 7 \equiv '1111111' \]

Conveniently, we'll use Python's * operator:

\[ \text{str} = '1' \times n \quad \# \text{concatenates '1' n times} \]

n not prime if str can be written as ('1'*k)*m, k>1, m>1

\[ (11+)\backslash 1+ \quad \# \text{recall that } \backslash 1 \text{ matches whatever } (11+) \text{ matches} \]

Special handling for n=1. Also, $ matches end of string

\[ \text{re.match}(r'1$|(11+)\backslash 1+$', '1'*n) . \text{group}(1) \]

Note this is a regex, not a regular expression

Regexes can tell apart strings that reg expressions can't
What strings can we tell apart with RE?
Exercise

Write a RE or automaton that accepts a string over the alphabet \{ (, ) \} iff the string has balanced parentheses:

(()(())) balanced
(()((()))) not balanced

Can’t be done. We need to count open left parens. Since the input can be arbitrarily large, we need a counter or a set of states that is unbounded in size.

Sadly, finite automata have only finite number of states.
Expressiveness of RE recognizers

What does it mean to "tell strings apart"?

Or "test a string" or "recognize a language", where language = a (potentially infinite) set of strings

It is to accept only strings that have some property
- e.g., can be written as ('1'*k)*m, for some k>1, m>1
- or contain only balanced parentheses: (((()))((())))
Primality testing revisited

Why can't RE test if a string matches ('1'*k)*m, k>1, m>1?

It may seem that

('1'*k)*m, k>1, m>1

is equivalent to

(11+)(11+)+

Exercise: Find a string that matches the latter but does not match the former.