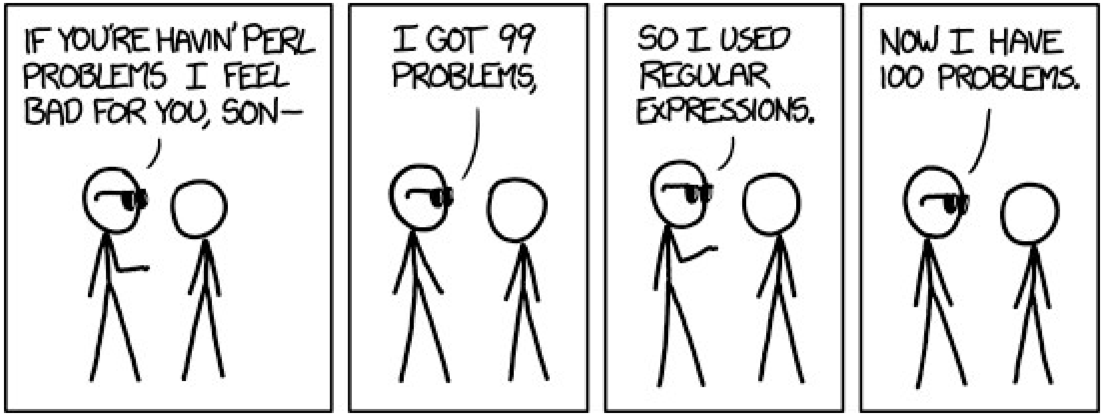
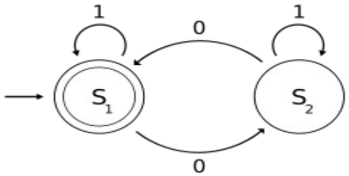
CS 432
Fall 2024

Mike Lam, Professor



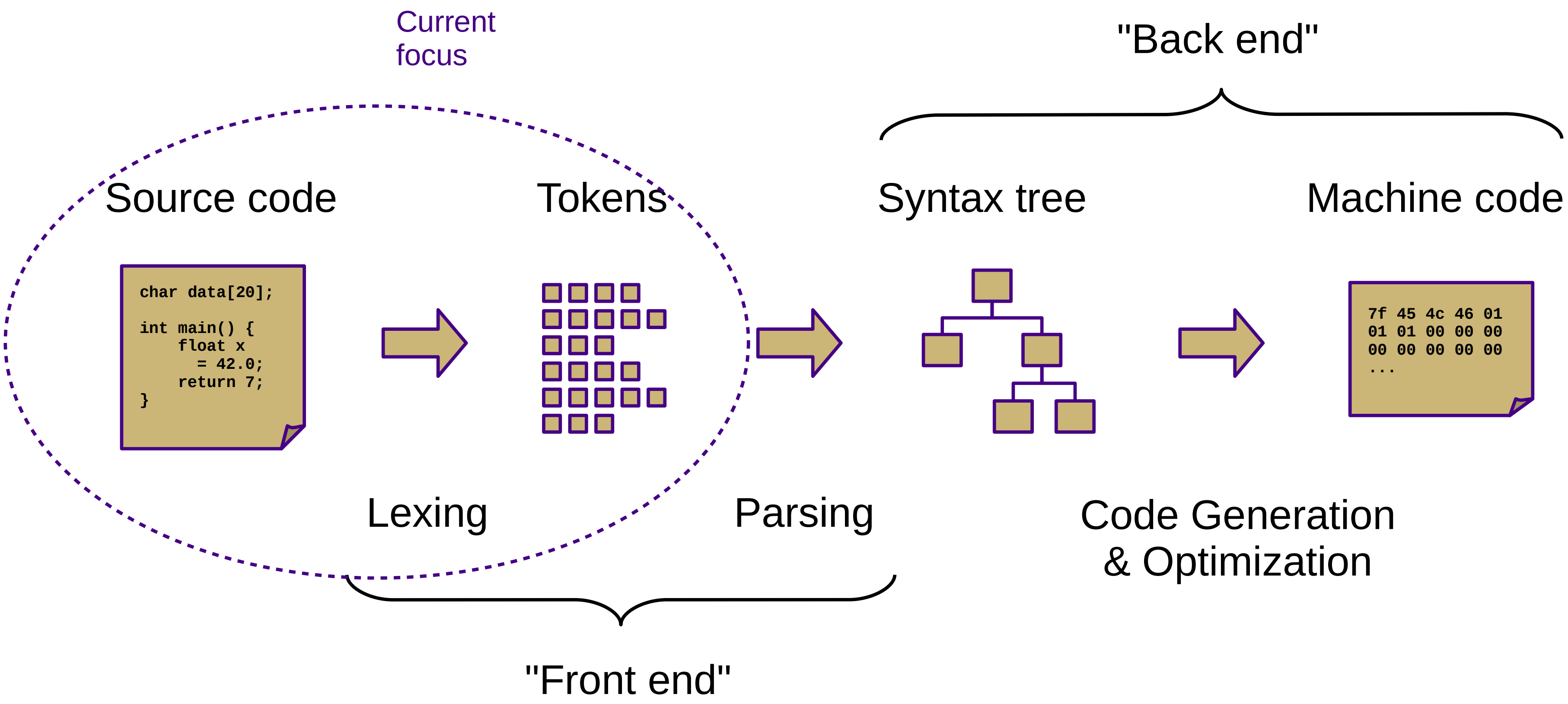
https://xkcd.com/1171/

`a|(bc)*`



Regular Expressions
and
Finite Automata

Compilation



Lexical Analysis

- Lexemes or tokens: the smallest building blocks of a language's syntax
- Lexing or scanning: the process of separating a character stream into tokens

```
total = sum(vals) / n

total       identifier
=           equals_op
sum         identifier
(           left_paren
vals        identifier
)           right_paren
/           divide_op
n           identifier
```

```
char *str = "hi";

char        keyword
*           star_op
str         identifier
=           equals_op
"hi"        str_literal
;           semicolon
```

Discussion question

- What is a *language*?

Language

- A language is "a (potentially infinite) set of strings over a finite alphabet"

Discussion question

- How do we describe languages?

xyy
xy
xyyzzz
xyz
xyzz
xyyzz
xyyz
xyzzz
(etc.)

xy
xyy
xyz
xyyz
xyzz
xyyzz
xyzzz
xyyzzz
(etc.)

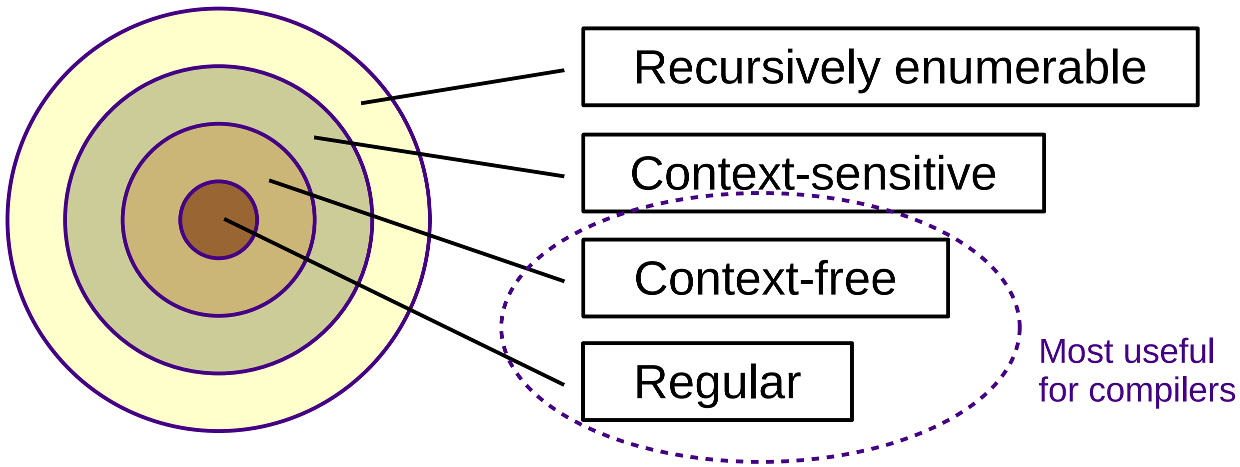
xy	**xyy**
xyz	**xyyz**
xyzz	**xyyzz**
xyzzz	**xyyzzz**
(etc.)	

Language description

- Ways to describe languages
  - Ad-hoc prose
    - “A single ‘x’ followed by one or two ‘y’s followed by any number of ‘z’s”
  - Formal regular expressions (current focus)
    - x(y|yy)z*
  - Formal grammars (in two weeks)
    - $A \rightarrow x\ B\ C$
    - $B \rightarrow y \mid y\ y$
    - $C \rightarrow z\ C \mid \varepsilon$

Languages

Chomsky Hierarchy of Languages



- Alphabet:
  - Σ = { finite set of all characters }
- Language:
  - L = { potentially infinite set of sequences of characters from Σ }

Regular expressions

- Regular expressions describe regular languages
  - Can also be thought of as generalized search patterns

- Three basic recursive operations:
  - Alternation: **A|B** — Lowest precedence
  - Concatenation: **AB** *or* **A␣B**
  - ("Kleene") Closure: **A*** — Highest precedence

Additionally: **ε** is a regex that matches the empty string

- Extended constructs:
  - Character sets/classes: **[0-9] ≡ [0...9] ≡ 0|1|2|3|4|5|6|7|8|9**
  - Repetition / positive closure: $\mathbf{A^2} \equiv$ **AA** $\mathbf{A^3} \equiv$ **AAA** $\mathbf{A^+} \equiv$ **AA***
  - Grouping: **(A|B)C ≡ AC|BC**

These are not covered extensively in your textbook!

Regular expressions

- Symbols with special meaning in regular expressions must be “escaped” to match the actual symbol
  - E.g., **a*** matches an “a” followed by an asterisk (“*”)
  - This is not usually necessary inside a character class
    - E.g., **a[*]** ≡ **a***
- Alternation of character classes can be condensed
  - E.g., **[a-z]|[A-Z]** ≡ **[a-zA-Z]**
- Starting a character class with a caret (“^”) forms the complement
  - E.g., **[^abc]** matches any character that is **NOT** “a”, “b”, or “c”
  - Outside a character class, **^** matches the beginning of a string and **$** matches the end of a string

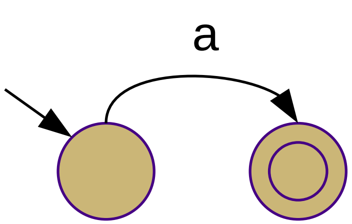
Discussion question

- How would you implement regular expressions?
  - Given a regular expression and a string, how would you tell whether the string belongs to the language described by the regular expression?

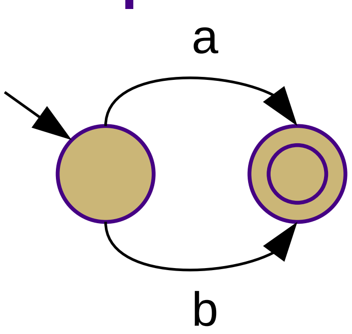
Lexical Analysis

- Implemented using state machines (finite automata)
  - Set of states with a single start state
  - Transitions between states on inputs (w/ implicit dead states)
  - Some states are final or accepting

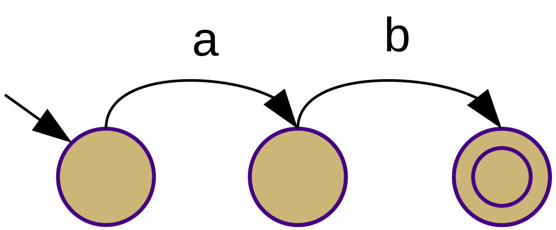
a



a|b

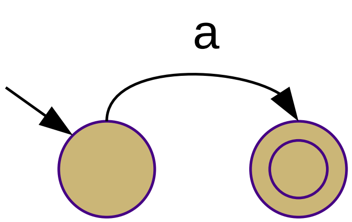


ab

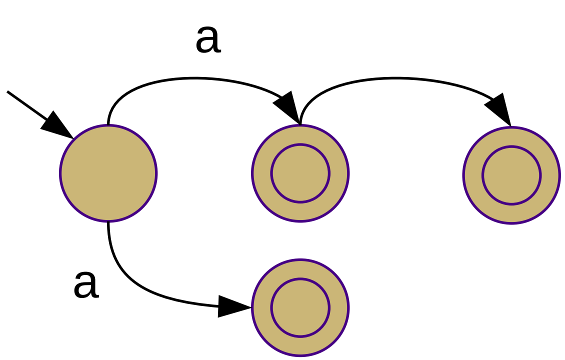


Lexical Analysis

- Deterministic vs. non-deterministic
  - Non-deterministic: multiple possible states for given sequence
  - One edge from each state per character (deterministic)
    - Might lead to implicit “dead state” w/ self-loop on all characters
  - Multiple edges from each state per character (non-deterministic)
  - “Empty” or ε-transitions (non-deterministic)



Deterministic (DFA)



Non-deterministic (NFA)

Deterministic finite automata

- Formal definition

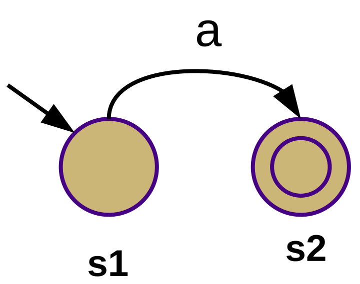
  S: set of states

  Σ: alphabet (set of characters)

  δ: transition function: (S, Σ) → S

  s_0: start state

  S_A: accepting/final states



S = { s1, s2 }
Σ = { a }
δ = { (s1, a → s2), (s2, a → Ø) }
s_0 = s1
S_A = { s2 }

- Acceptance algorithm

  $s := s_0$

  for each input c:

  $\quad s := \delta(s,c)$

  return $s \in S_A$

Alternative δ representation:

	a
s1	s2
s2	Ø

Non-deterministic finite automata

- Formal Definition
  - S, Σ, s_0, and S_A same as DFA
  - δ: $(S, \Sigma \cup \{\varepsilon\}) \rightarrow [S]$
  - ε-closure: all states reachable from s via ε-transitions
    - Formally: $\varepsilon\text{-closure}(s) = \{s\} \cup \{ t \in S \mid (s, \varepsilon) \rightarrow t \in \delta \}$
    - Extended to sets by union over all states in set

- Acceptance algorithm

$T := \varepsilon\text{-}closure(s_0)$

for each input c:

  $N := \{\}$

  for each s ***in*** T:

    $N := N \cup \varepsilon\text{-}closure(\delta(s,c))$

  $T := N$

return $|T \cap S_A| > 0$

Summary

DFAs

- S: set of states
- Σ: alphabet (set of characters)
- δ: transition function: (S, Σ) → S
- s_0: start state
- S_A: accepting/final states

accept():

$s := s_0$

for each input c:

$s := \delta(s,c)$

return $s \in S_A$

NFAs

- δ may return a set of states
- δ may contain ε-transitions

accept():

$T := \varepsilon\text{-}closure(s_0)$

for each input c:

$N := \{\}$

for each s ***in*** T:

$N := N \cup \varepsilon\text{-}closure(\delta(s,c))$

$T := N$

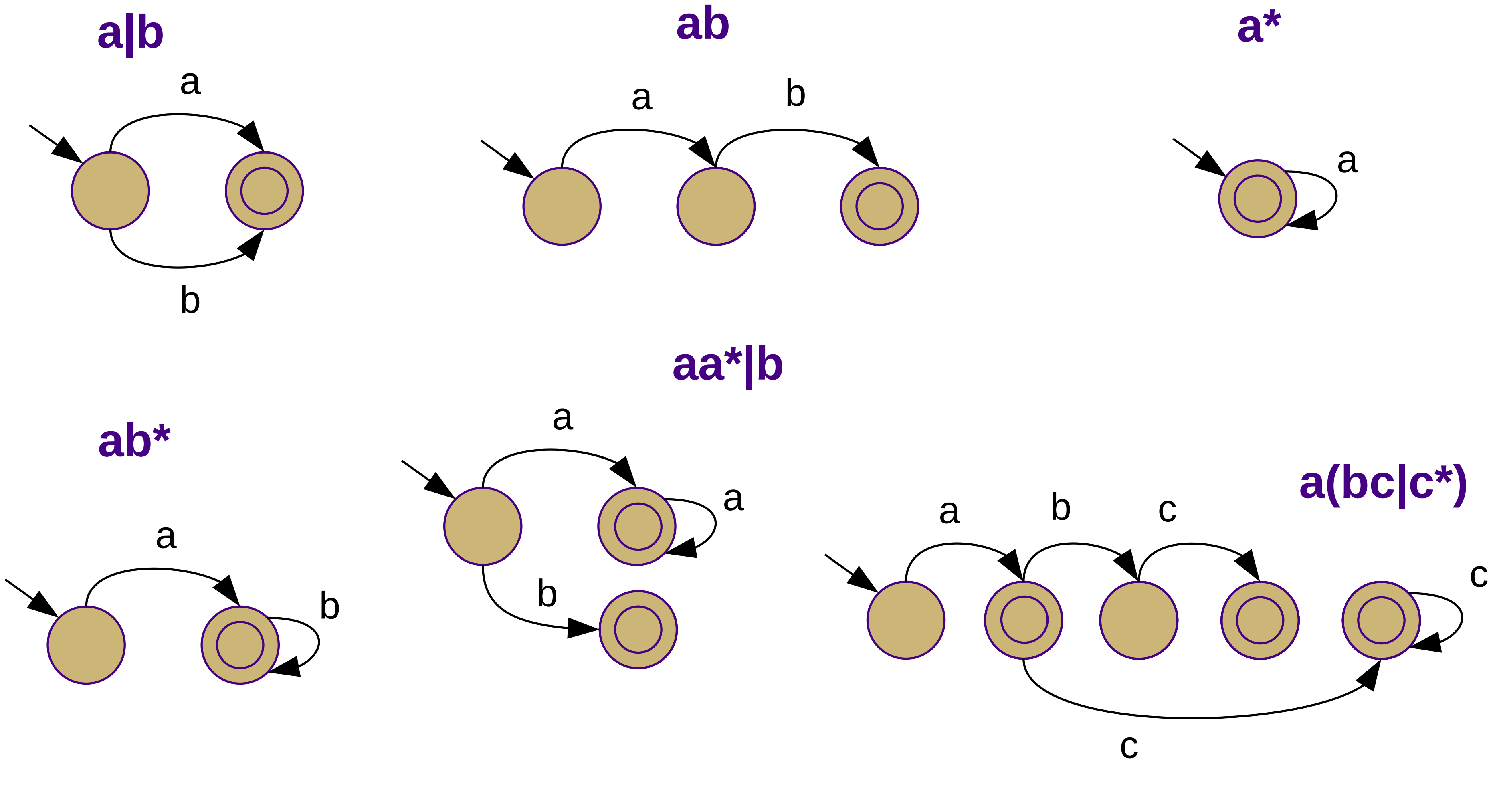
return $|T \cap S_A| > 0$

Equivalence

- A regular expression and a finite automaton are equivalent if they recognize the same language
  - Same applies between different REs and between different FAs
- Regular expressions, NFAs, and DFAs all describe the same set of languages
  - "Regular languages" from Chomsky hierarchy
- Next week, we will learn how to convert between them

Lexical Analysis

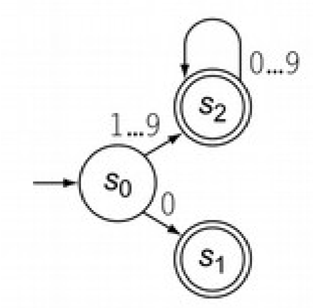
- Examples:



Examples

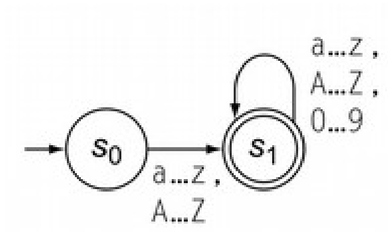
Unsigned integers

$0 \mid [1 \ldots 9]\,[0 \ldots 9]^{*}$



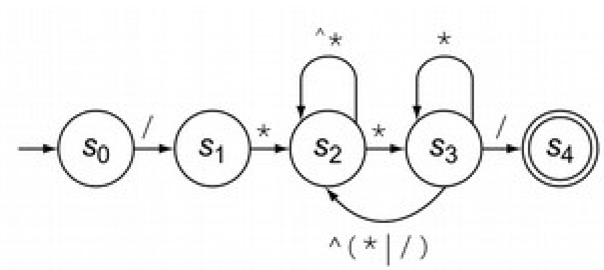
Identifiers

$([A \ldots Z] \mid [a \ldots z])\ \ ([A \ldots Z] \mid [a \ldots z] \mid [0 \ldots 9])^{*}$



Multi-line comments

$/ \ast \; (\, \hat{}\ast \mid \ast^{+} \, \hat{}/ \,)^{*} \; \ast /$



Exercise

- Construct state machines for the following regular expressions:

x*yz* **1(1|0)*** **1(10)*** **(a|b|c)(ab|bc)**

(dd*.d*)|(d*.dd*) ← ε-transitions may make this one slightly easier

Application

- P1: Use POSIX regular expressions to tokenize Decaf files
  - Process the input one line at a time
  - Generally, create one regex per token type
    - Each regex begins with “^” (only match from beginning)
    - Prioritize regexes and try each of them in turn
    - When you find a match, extract the matching text
    - Repeat until no match is found or the input is consumed
  - Less efficient than an auto-generated lexer
    - However, it is simpler to understand
    - Our approach to P2 will be similar

Source code

Tokens