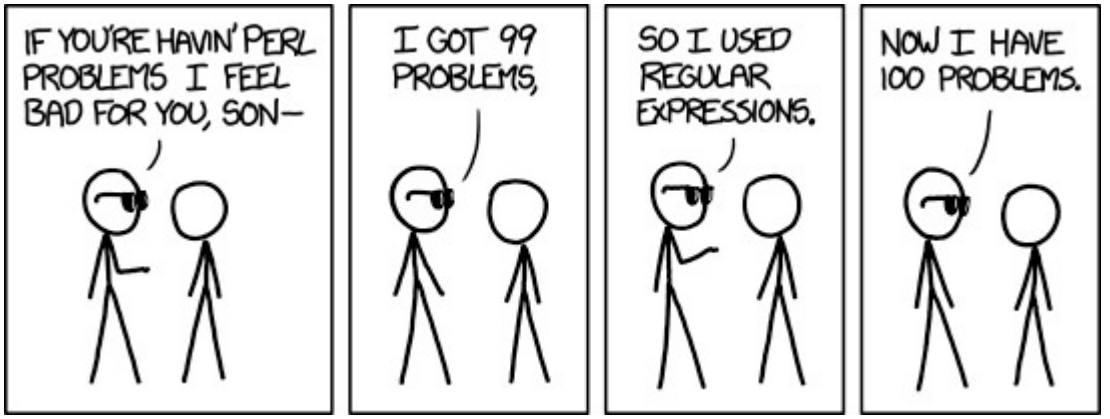
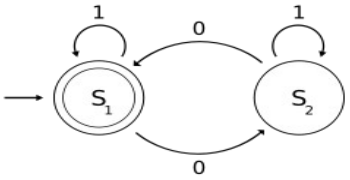
CS 432
Fall 2026

Mike Lam, Professor



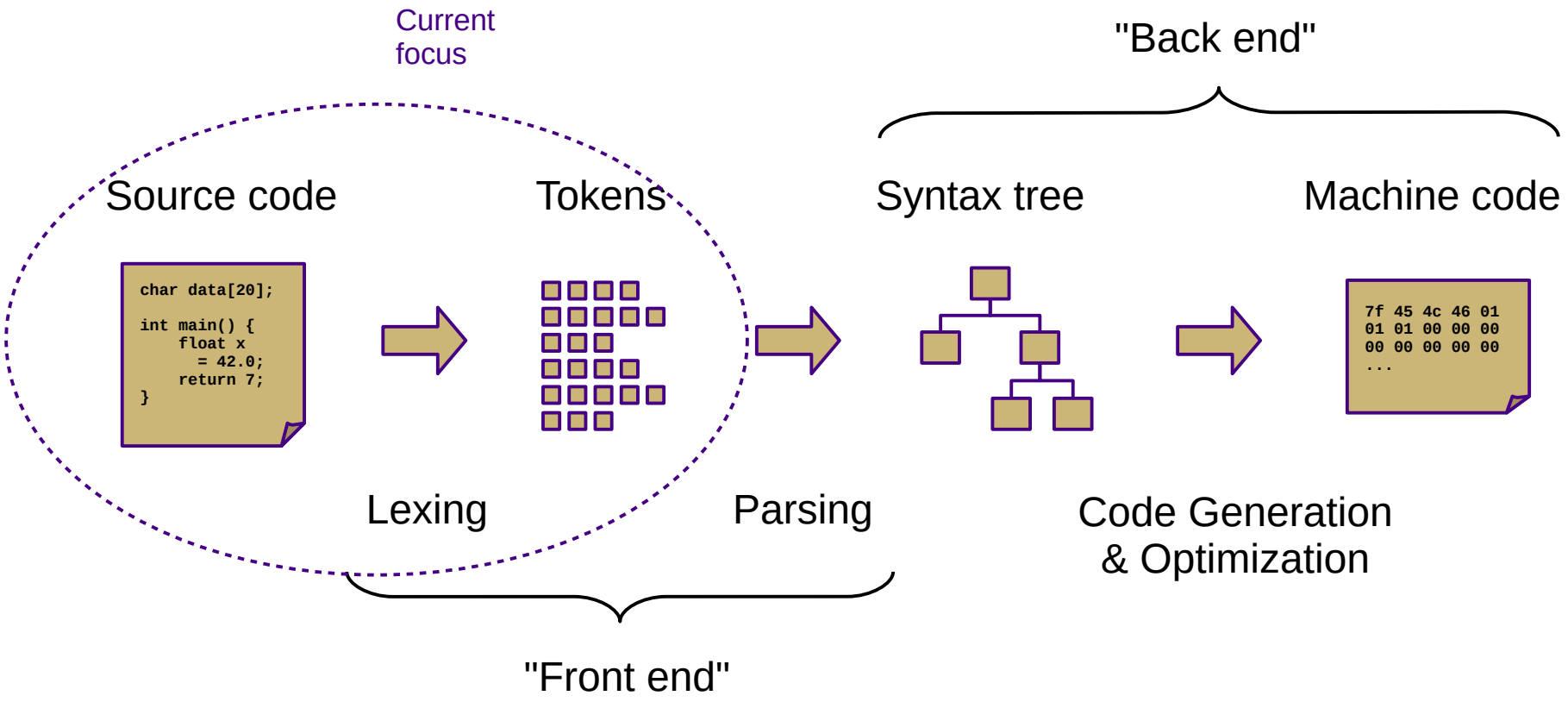
https://xkcd.com/1171/

a|(bc)*



Regular Expressions
and
Finite Automata

Compilation



Lexical Analysis

- Lexemes: a substring of the input (“building blocks”)
- Tokens: a lexeme w/ a category
- Lexing or scanning: the process of separating a character stream into tokens

`total = sum(vals) / n`

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

`char *str = "hi";`

`char`	`keyword`
`*`	`star_op`
`str`	`identifier`
`=`	`assign_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.)

Unhelpful

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

Better

xy
xyz
xyzz
xyzzz
(etc.)

xyy
xyyz
xyyzz
xyyzzz

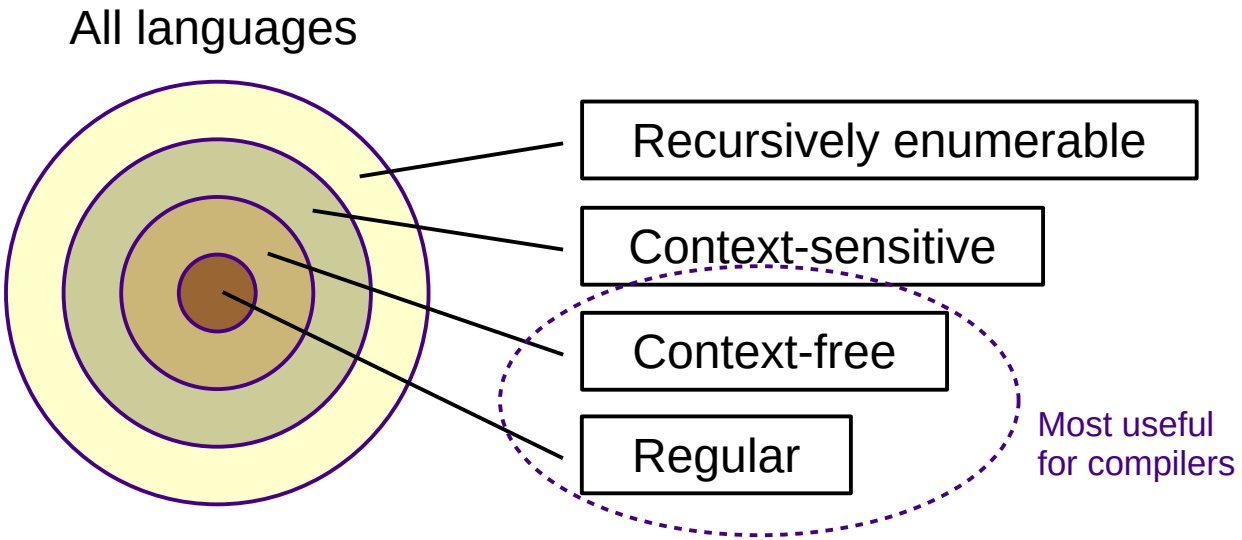
Even better

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 → x B C
    - B → y | y y
    - C → z C | ε

Languages

Chomsky Hierarchy of Languages



- Alphabet:

  Σ: a finite set of all characters

- Language:
  - L: a potentially infinite subset of all strings from Σ ($L \subseteq \Sigma^*$)

Aside: Alphabets

- In this class, we will mostly stick to ASCII characters
  - This is mostly for simplicity
  - However, it is a very American-focused character set
  - (It's in the name! American Standard Code for Information Interchange)
- Many modern languages support Unicode for variable and function names
  - This covers most major world scripts
  - But keywords and core library function names are usually in English, creating barriers for non-native English speakers
  - If you are interested in an alternative, check out Hedy, a multi-lingual programming language

Regular expressions

- Regular expressions describe regular languages
  - Can also be thought of as generalized search patterns
- Base cases: **ε** (empty string) and **a** ∈ Σ (single character)
- Three basic recursive operations:
  - Alternation: **A|B** — **Lowest precedence**
  - Concatenation: **AB** *or* **A␣B**
  - ("Kleene") Closure: **A*** — **Highest precedence**
- Extended constructs:
  - Character sets/classes: **[0-9] ≡ [0...9] ≡ 0|1|2|3|4|5|6|7|8|9**
  - Repetition / positive closure: **A^2 ≡ AA A^3 ≡ AAA A^+ ≡ AA***
  - Grouping: **(A|B)C ≡ AC|BC**

Extended constructs are not covered extensively in your textbook!

Regular expression caveats

- 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”
  - POSIX: outside a character class, **^** matches the beginning of a string and **$** matches the end of a string (this is not the case in your textbook)

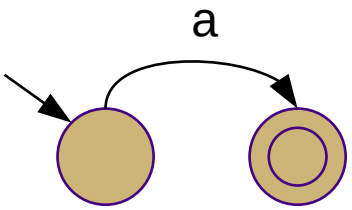
Discussion question

- 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?

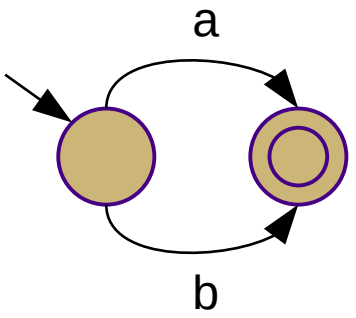
Finite automata

- Implemented using state machines (finite automata)
  - Set of states with a single start state (arrow w/ no source)
  - Transitions between states on inputs (w/ implicit dead states)
  - Some states are final or accepting (double outline)

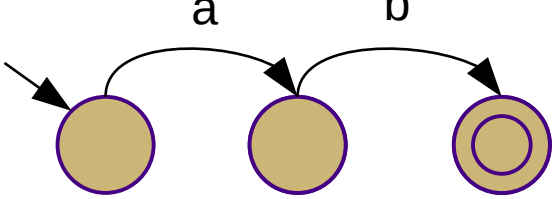
a



a|b

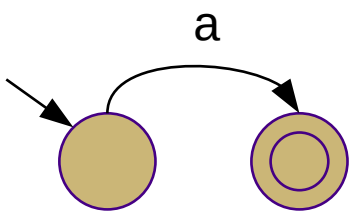


ab

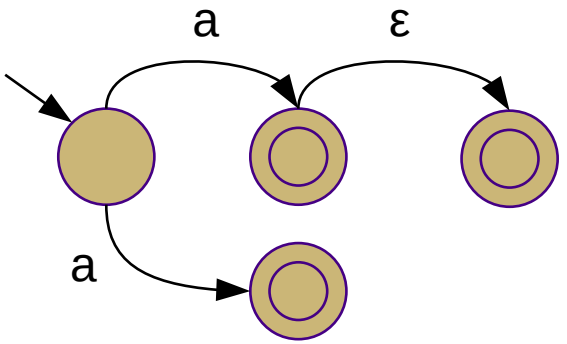


DFA vs. NFA

- Deterministic vs. non-deterministic
  - Deterministic
    - One edge (to one state) from each state per character
    - Might lead to implicit “dead state” w/ self-loop on all characters
  - Non-deterministic
    - Possibility of multiple edges from each state per character
    - Possibility of “empty” or ε-transitions



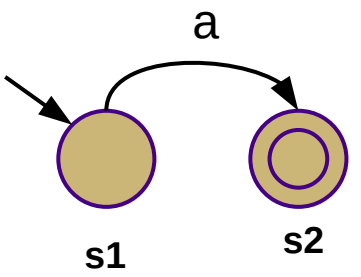
Deterministic (DFA)



Non-deterministic (NFA)

Deterministic finite automata

- Formal definition
  - S: set of states (finite)
  - Σ: alphabet (set of characters)
  - δ: transition function: $S \times \Sigma \rightarrow S$
    - (use ∅ to denote "dead state")
  - s_0: start state
  - S_A: accepting/final states ($S_A \subseteq S$)



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

- Acceptance algorithm

$s := s_0$
for each input c:
  $s := \delta(s,c)$
return $s \in S_A$

Alternative δ representation:

	a
s1	s2
s2	∅

Non-deterministic finite automata

- Formal Definition
  - S, Σ, s_0, and S_A same as DFA
  - δ: S × (Σ ∪ {ε}) → PowerSet(S)
  - ε-closure: all states reachable from s via ε-transitions
    - Formally: ε-closure(s) = {s} ∪ $U_{t \in \delta(s, \varepsilon)}$ ε-closure(t)
    - 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: $S \times \Sigma \rightarrow S$
- 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 returns 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:

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

$T := N$

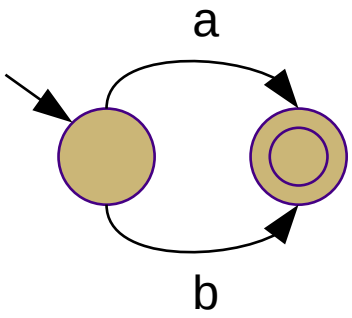
return $|T \cap S_A| > 0$

Equivalence

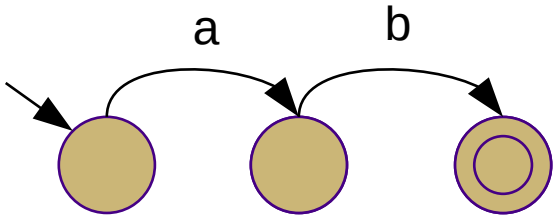
- A regular expression and a finite automaton are equivalent if they recognize the same language
  - Same applies between different REs and between different FAs
- Regular expressions, NFAs, and DFAs all describe the same set of languages
  - "Regular languages" from Chomsky hierarchy
- Next week, we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

Examples

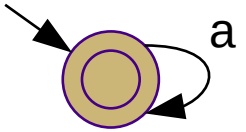
a|b



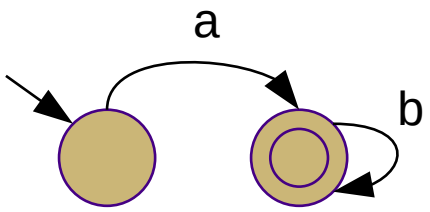
ab



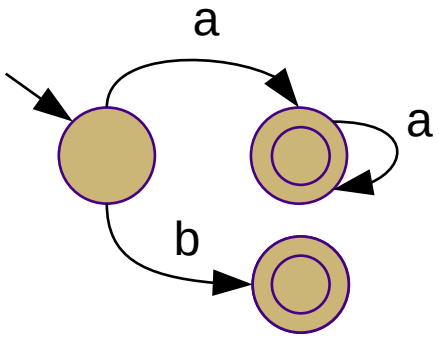
a*



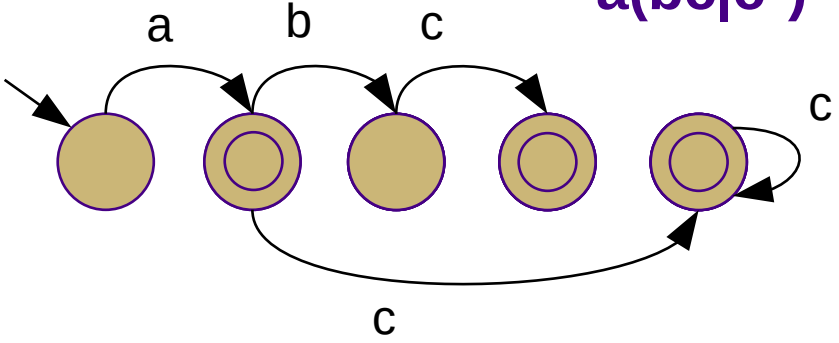
ab*



aa*|b



a(bc|c*)



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

Lexing (P1)

- Option 1: Hand-code the state machine
  - Provides the most control over performance and error messages
  - Most “real” compilers do this
    - (e.g., GCC, Clang, Go, Rust, CPython)
  - I suspect most AI tools will do this by default
  - Weakness: becomes spaghetti code at scale
    - Hard to read and explain
    - Easy to introduce subtle bugs that are hard to catch

Lexing (P1)

- Option 2: Use a regex library
  - Produces the shortest and most readable lexers
  - Regexes encode the language specification directly
  - This is what the original CS 432 P1 required
    - We used POSIX regular expressions (`regex.h`) with a thin object-oriented wrapper
  - Weakness: performance at scale
    - Usually need to run multiple regexes repeatedly
    - This doesn't really matter for Decaf programs

Lexing (P1)

- Option 3: Use a lexer generator
  - Auto-generates table-based lexer source code
  - Converts from regex → NFA → DFA → code
  - Best of both worlds: concise and fast
    - This is why we will learn the theory of it next week
  - Weakness: requires learning a tool (probably `flex`)
    - Not really the point of the project