Regular Expressions and Finite Automata

a | (bc) *

https://xkcd.com/1171/
char data[20];
int main() {
  float x;
  x = 42.0;
  return 7;
}
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
```

```
char *str = "hi";
```

```plaintext
<table>
<thead>
<tr>
<th>total</th>
<th>identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equals_op</td>
</tr>
<tr>
<td>sum</td>
<td>identifier</td>
</tr>
<tr>
<td>(</td>
<td>left_paren</td>
</tr>
<tr>
<td>vals</td>
<td>identifier</td>
</tr>
<tr>
<td>)</td>
<td>right_paren</td>
</tr>
<tr>
<td>/</td>
<td>divide_op</td>
</tr>
<tr>
<td>n</td>
<td>identifier</td>
</tr>
<tr>
<td>char</td>
<td>keyword</td>
</tr>
<tr>
<td>*</td>
<td>star_op</td>
</tr>
<tr>
<td>str</td>
<td>identifier</td>
</tr>
<tr>
<td>=</td>
<td>equals_op</td>
</tr>
<tr>
<td>&quot;hi&quot;</td>
<td>str_literal</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
</tbody>
</table>
```
Discussion question

• What is a 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
xzzz
xyyyyyzzzz
(etc.)
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)
  - $A \rightarrow x \ B \ C$
  - $B \rightarrow y \mid y \ y$
  - $C \rightarrow z \ C \mid \epsilon$
Languages

Chomsky Hierarchy of Languages

- **Alphabet:**
  - $\Sigma = \{ \text{finite set of all characters} \}$

- **Language:**
  - $L = \{ \text{potentially infinite set of sequences of characters from } \Sigma \}$
Regular expressions

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

- Three basic recursive operations:
  - Alternation: \( a|b \)  
  - Concatenation: \( ab \)
  - ("Kleene") Closure: \( a^* \)

- Extended constructs:
  - Character sets/classes: \([0-9] \equiv [0...9] \equiv 0|1|2|3|4|5|6|7|8|9\)
  - Repetition / positive closure: \( a^2 \equiv aa \quad a^3 \equiv aaa \quad a+ \equiv aa^*\)
  - Grouping: \((a|b)c \equiv ac|bc\)

Additionally: \( \epsilon \) is a regex that matches the empty string

These are not covered extensively 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?
Lexical Analysis

- Implemented using state machines (finite state automata)
  - Set of states with a single start state
  - Transitions between states on inputs (w/ implicit dead states)
  - Some states are final or accepting
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)](image1)

![Non-deterministic (NFA)](image2)
Deterministic finite automata

- **Formal definition**
  - $S$: set of states
  - $\Sigma$: alphabet (set of characters)
  - $\delta$: transition function: $(S, \Sigma) \rightarrow S$
  - $s_0$: start state
  - $S_A$: accepting/final states

- **Acceptance algorithm**

  $$s := s_0$$

  **for each input** $c$:

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

  **return** $s \in S_A$

- Alternative $\delta$ representation:

  $S = \{ s_1, s_2 \}$
  $\Sigma = \{ a \}$
  $\delta = \{ (s_1, a \rightarrow s_2), (s_2, a \rightarrow \emptyset) \}$
  $s_0: = s_1$
  $SA = \{ s_2 \}$
Non-deterministic finite automata

• Formal Definition
  - $S$, $\Sigma$, $s_0$, and $S_A$ same as DFA
  - $\delta$: $(S, \Sigma \cup \{\varepsilon\}) \rightarrow [S]$  
  - $\varepsilon$-closure: all states reachable from $s$ via $\varepsilon$-transitions
    - Formally: $\varepsilon$-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
  
  $T := \varepsilon$-closure($s_0$)

  for each input $c$:
    
    $N := \{\}$

    for each $s$ in $T$:
      
      $N := N \cup \varepsilon$-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
- s₀: start state
- S_A: accepting/final states

accept():

\[
\begin{align*}
    &s := s_0 \\
    &\text{for each input } c:\ \\
    &\hspace{1em} s := \delta(s, c) \\
    &\text{return } s \in S_A
\end{align*}
\]

NFAs

- δ may return a set of states
- δ may contain ε-transitions
- δ may contain transitions to multiple states on a symbol

accept():

\[
\begin{align*}
    &T := \varepsilon\text{-closure}(s_0) \\
    &\text{for each input } c:\ \\
    &\hspace{1em} N := \{\} \\
    &\hspace{1em} \text{for each } s \text{ in } T:\ \\
    &\hspace{2em} N := N \cup \varepsilon\text{-closure}(\delta(s, c)) \\
    &T := N \\
    &\text{return } |T \cap S_A| > 0
\end{align*}
\]
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
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 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
```c
char data[20];
int main() {
    float x = 42.0;
    return 7;
}
```
Lexical Analysis

• Examples:

- $a|b$
- $a b$
- $a^*$
- $aa^*|b$
- $ab^*$
- $a(bc|c^*)$
Examples

Unsigned integers

\[0 | [1...9][0...9]^*\]

Identifiers

\n
([A...Z] | [a...z]) ([A...Z] | [a...z] | [0...9])^*

Multi-line comments

\n
\/* ( ^* | *+^/ )* */
Exercise

• Construct state machines for the following regular expressions:

\[ x^* y z^* \quad 1(1|0)^* \quad 1(10)^* \quad (a|b|c)(ab|bc) \]

\[ (dd^*.d^*)|(d^*.dd^*) \quad \rightarrow \varepsilon\text{-transitions may make this one slightly easier} \]