Regular Expressions and Finite Automata

\[ a \mid (bc)^* \]
Compilation

Source code

Lexing

Tokens

Parsing

Syntax tree

Code Generation & Optimization

Machine code

"Back end"

Current focus

"Front end"
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

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

```
total = sum(vals) / n
```

<table>
<thead>
<tr>
<th>total</th>
<th>identifier</th>
<th>char</th>
<th>keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equals_op</td>
<td>*</td>
<td>star_op</td>
</tr>
<tr>
<td>sum</td>
<td>identifier</td>
<td>str</td>
<td>identifier</td>
</tr>
<tr>
<td>(</td>
<td>left_paren</td>
<td>=</td>
<td>equals_op</td>
</tr>
<tr>
<td>vals</td>
<td>identifier</td>
<td>&quot;hi&quot;</td>
<td>str_literal</td>
</tr>
<tr>
<td>)</td>
<td>right_paren</td>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>/</td>
<td>divide_op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>identifier</td>
<td></td>
<td></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
xyyzzzz
xyz
xyzz
xyyzz
xyyzz
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)
    • $A \rightarrow x \ B \ C$
    • $B \rightarrow y \ | \ y \ y$
    • $C \rightarrow z \ C \ | \ \varepsilon$
Languages

Chomsky Hierarchy of Languages

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

- **Language:**
  - $L = \{ \text{set of sequences of characters from } \Sigma \}$
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: $[0-9] \equiv [0...9] \equiv 0|1|2|3|4|5|6|7|8|9$
  - Positive closure: $a^2 \equiv aa$  $a^3 \equiv aaa$  $a^+ \equiv aa^*$
  - Grouping: $(a|b)c \equiv ac|bc$

Additionally: $\varepsilon$ 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

![Diagram of state machines for a, a|b, and ab]
Lexical Analysis

- **Deterministic vs. non-deterministic**
  - Non-deterministic: multiple possible states for given sentence
  - One edge from each state per character (deterministic)
  - 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**

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

Alternative $\delta$ representation:

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Non-deterministic finite automata

• Formal Definition
  - $S, \Sigma, s_0, \text{ and } S_A$ 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
• $\Sigma$: alphabet (set of characters)
• $\delta$: transition function: $(S, \Sigma) \rightarrow S$
• $s_0$: start state
• $S_A$: accepting/final states

accept()

Accept:

$s := s_0$

for each input $c$:

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

return $s \in S_A$

NFAs

• $\delta$ may return a set of states
• $\delta$ may contain $\varepsilon$-transitions
• $\delta$ may contain transitions to multiple states on a symbol

accept()

Accept:

$T := \varepsilon$-closure($s_0$)

for each input $c$:

$N := \emptyset$

for each $s$ in $T$:

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

$T := N$

return $|T \cap S_A| > 0$
Lexical Analysis

• Examples:

\[
\begin{align*}
\text{a|b} & : \quad \text{Diagram 1} \\
\text{ab} & : \quad \text{Diagram 2}
\end{align*}
\]
Lexical Analysis

• Examples:

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

• Examples:

- `a|b`

- `ab`

- `a*`

- `aa*|b`

- `ab*`

- `a(bc|c*)`
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
• PA2: Use Java regular expressions to tokenize Decaf files
  - Process the input one line at a time
  - Generally: 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 PA3 will be similar)
Activity

• Construct state machines for the following regular expressions:

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

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