Regular Expressions and Finite Automata
Compilation

Source code

char data[20];
int main() {
  float x = 42.0;
  return 7;
}

Tokens

Lexing

Parsing

Syntax tree

Code Generation & Optimization

Machine code

"Front end"

"Back end"

Current focus
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

\[
\text{total} = \frac{\text{sum}(\text{vals})}{\text{n}}
\]

char *\text{str} = "hi";

```
total = identifier = equals_op sum identifier ( left_paren vals identifier ) right_paren / divide_op n identifier 
char = keyword * star_op str identifier = equals_op "hi" str_literal ; semicolon
```
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
xyyyyy
xyz
xyzz
xyyzz
xyyzzz
(etc.)

xy
xyy
xyz
xyyz
xyzz
xyyzz
xyyzzz
(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)
    • $A \to x \ B \ C$
    • $B \to y \ | \ y \ y$
    • $C \to z \ C \ | \ \varepsilon$
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: \([0-9] \equiv [0...9] \equiv 0|1|2|3|4|5|6|7|8|9\)
  - Positive closure: \( a^2 \equiv aa \quad a^3 \equiv aaa \quad a^+ \equiv aa^* \)
  - 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
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 $\varepsilon$-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
  s₀: start state
  Sₐ: accepting/final states

• Acceptance algorithm
  s := s₀
  for each input c:
    s := δ(s, c)
  return s ∈ Sₐ

S = \{ s₁, s₂ \}
Σ = \{ a \}
δ = \{ (s₁, a → s₂), (s₂, a → Ø) \}
s₀ = s₁
Sₐ = \{ s₂ \}

Alternative δ representation:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>s₂</td>
</tr>
<tr>
<td>s₂</td>
<td>Ø</td>
</tr>
</tbody>
</table>
Non-deterministic finite automata

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

• Acceptance algorithm
  
  $T := \epsilon$-closure($s_0$)

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

  for each $s$ in $T$:
  
  $N := N \cup \epsilon$-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():
    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 $\epsilon$-transitions
- $\delta$ may contain transitions to multiple states on a symbol

```
accept():
    T := \epsilon-closure(s_0)
    for each input c:
        N := {} 
        for each s in T:
            N := N \cup \epsilon-closure(\delta(s,c))
        T := N
    return |T \cap S_A| > 0
```
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, 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 PA3 will be similar

```java
char data[20];
int main() {  
    float x = 42.0;
    return 7;
}
```
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

\[ /* ( ^* \mid *+^/ )^* */ \]
Exercise

• Construct state machines for the following regular expressions:

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

\[ (dd^*.d^*)|(d^*.dd^*) \]

← \( \varepsilon \)-transitions may make this one slightly easier