Regular Expressions
and
Finite Automata

\[ a | (bc)^* \]
int main() {
    float 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";
```

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

- $xyy$
- $xy$
- $xyyyyyzzzz$
- $xyz$
- $xyzz$
- $xyyyyyzzzz$
- $xyyzz$
- $xyzzz$
- $xyyyyyzzzz$
- $(etc.)$

- $xy$
- $xyy$
- $xyz$
- $xyyz$
- $xyzz$
- $xyyyyyzzzz$
- $xyzzz$
- $xyyyyyzzzz$
- $(etc.)$
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)
  - $A \rightarrow x \ B \ C$
  - $B \rightarrow y \ | \ y\ y$
  - $C \rightarrow z \ C \ | \ \epsilon$
Languages

Chomsky Hierarchy of Languages

- **Regular**
- **Context-free**
- **Context-sensitive**
- **Recursively enumerable**

- **Most useful for compilers**

- **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: \( A \mid B \)
  - 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: \((A\mid B)C \equiv AC\mid BC\)

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

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
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 $\varepsilon$-transitions (non-deterministic)

![Deterministic (DFA)](image1)

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

• Formal definition
  
  S: set of states
  Σ: alphabet (set of characters)
  δ: transition function: (S, Σ) → S
  s₀: start state
  S_A: accepting/final states

• Acceptance algorithm

  \[ s := s₀ \]

  \text{for each input } c:\]

  \[ s := δ(s,c) \]

  \text{return } s ∈ S_A

Artificial δ representation:

\[
\begin{array}{c|c}
\text{a} & \text{S} \\
\hline
\text{s1} & \text{s2} \\
\text{s2} & \emptyset
\end{array}
\]
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\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
- \( \Sigma \): 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 := \( \epsilon\text{-closure}(s_0) \)
    for each input c:
        N := \{ \}
        for each s in T:
            N := N \( \cup \epsilon\text{-closure}(\delta(s, c)) \)
        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:

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

Unsigned integers

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

Identifiers

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

Multi-line comments

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

• Construct state machines for the following regular expressions:

- $x^*yz^*$
- $1(1|0)^*$
- $1(10)^*$
- $(a|b|c)(ab|bc)$
- $(dd^*.d^*)|(d^*.dd^*)$  ← $\varepsilon$-transitions may make this one slightly easier
• 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

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