Regular Expressions and Finite Automata

a | (bc)*
Compiling is the process of transforming source code into machine code. The source code is the human-readable version, while the machine code is the binary code that the computer's processor can execute directly. The compilation process involves several stages:

1. **Lexing (Tokenization)**: This stage involves scanning the source code to identify tokens, which are the smallest units of code that have a meaning. Tokens are then converted into a stream of tokens that can be processed further.

2. **Parsing**: Tokens are then analyzed to build a syntax tree, which represents the structure of the code. This tree is used to check the correctness of the code and to generate instructions for the target machine.

3. **Code Generation**: The syntax tree is transformed into machine code, which is a binary representation of the instructions that the computer's processor can execute.

4. **Optimization**: The generated machine code is optimized to improve its performance, often by reducing the size of the code or the time it takes to execute.

The diagram illustrates these stages, starting with source code and going through lexing, parsing, code generation, and optimization to produce the final machine code.
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

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

char *str = "hi";

<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?
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
xyz
xyzz
xyyzz
xxyzzz
(etc.)

xy
xyy
xyz
xyyz
xyzz
xyyzz
xxyzz
xxyyyyy
(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)
    - $A \rightarrow x\,B\,C$
    - $B \rightarrow y \mid y\,y$
    - $C \rightarrow z\,C \mid \varepsilon$
Chomsky Hierarchy of Languages

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

• **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: $AB$ or $A_B$
  - ("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: $A^2 \equiv AA$ $A^3 \equiv AAA$ $A^+ \equiv AA^*$
  - Grouping: $(A|B)C \equiv AC|BC$

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)

Non-deterministic (NFA)
```
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** \( s \in S_A \)

\[
\begin{array}{c|c}
  S & \Sigma \\
  \hline
  s_1 & a \\
  \hline
  s_2 & \emptyset \\
\end{array}
\]

\( S = \{ s_1, s_2 \} \)  
\( \Sigma = \{ a \} \)  
\( \delta = \{ (s_1, a \rightarrow s_2), (s_2, a \rightarrow \emptyset) \} \)  
\( s_0: = s_1 \)  
\( S_A = \{ s_2 \} \)
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

\[
T := \varepsilon\text{-closure}(s_0)
\]

\textit{for each input} \( c \):

\[
N := \{
\]

\textit{for each} \( s \) \textit{in} \( T \):

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

\[
T := N
\]

\textit{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

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

**accept()**:

\[ T := \varepsilon\text{-}closure(s₀) \]

**for each input c:**

\[ N := \{ \} \]

**for each s in T:**

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

\[ T := N \]

**return** s ∈ S_A

\[ s := \delta(s,c) \]

\[ return \ |T \cap S_A| > 0 \]
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

\[ 0 | [1\ldots9] [0\ldots9]^* \]

Identifiers

\[ ([A\ldots Z] | [a\ldots z]) ([A\ldots Z] | [a\ldots z] | [0\ldots9])^* \]

Multi-line comments

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

- Construct state machines for the following regular expressions:
  
  \[
  \begin{align*}
  x^*yz^* & \quad 1(1|0)^* & \quad 1(10)^* & \quad (a|b|c)(ab|bc) \\
  (dd^*.d^*)&(d^*.dd^*) & \quad \text{← \(\varepsilon\)-transitions may make this one slightly easier}
  \end{align*}
  \]
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

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