Formal Languages
Strings

- **Alphabet**: a finite set of symbols
  - Normally characters of some character set
  - E.g., ASCII, Unicode
  - $\Sigma$ is used to represent an alphabet

- **String**: a finite sequence of symbols from some alphabet
  - If $s$ is a string, then $|s|$ is its length
  - The empty string is symbolized by $\epsilon$
String Operations

Concatenation

- $x = \text{hi}, y = \text{bye} \rightarrow xy = \text{hibye}$

- $s\epsilon = s = \epsilon s$

\[
s^i = \begin{cases} 
\epsilon, & \text{if } i = 0 \\
 s^{i-1}s, & \text{if } i > 0 
\end{cases}
\]
Parts of a String

- Prefix
- Suffix
- Substring
- Proper prefix, suffix, or substring
- Subsequence
Language

• A language is a set of strings over some alphabet

\[ L \subseteq \Sigma^* \]

• Examples:
  – \( \emptyset \) is a language
  – \( \{ \epsilon \} \) is a language
  – The set of all legal Java programs
  – The set of all correct English sentences
Operations on Languages

Of most concern for lexical analysis

- Union
- Concatenation
- Closure
The union of languages $L$ and $M$

$$L \cup M = \{ s \mid s \in L \text{ or } s \in M \}$$
Concatenation

The concatenation of languages \( L \) and \( M \)

\[
LM = \{ st \mid s \in L \text{ and } t \in M \}
\]
The Kleene closure of language $L$

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Zero or more concatenations
The positive closure of language $L$

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

One or more concatenations
Example

- Let $L = \{A, B, C, \ldots, Z, a, b, c, \ldots, z\}$
- Let $D = \{0, 1, 2, \ldots, 9\}$

$$
L \cup D \\
L^4 \\
L(L \cup D)^* \\
LD \\
L^* \\
D^+
$$
Regular Expressions

- A convenient way to represent languages that can be processed by lexical analyzers

- Notation is slightly different than the set notation presented for languages

- A regular expression is built from simpler regular expressions using a set of defining rules

- A regular expression represents strings that are members of some regular set
Rules for Defining Regular Expressions

- The regular expression $r$ denotes the language $L(r)$
- $\epsilon$ is a regular expression that denotes $\{\epsilon\}$, the set containing the empty string
- If $a$ is a symbol in the alphabet, then $a$ is a regular expression that denotes $\{a\}$, the containing the string $a$
- How to distinguish among these notations
Combining Regular Expressions

- Let $r$ and $s$ be regular expressions that denote the languages $L(r)$ and $L(s)$ respectively

  - $(r) | (s)$ is a regular expression denoting $L(r) \cup L(s)$
  - $(r)(s)$ is a regular expression denoting $L(r)L(s)$
  - $(r)^*$ is a regular expression denoting $(L(r))^*$
  - $(r)$ is a regular expression denoting $L(r)$

- The language denoted by a regular expression is called a regular set
More Formally

$a \in \Sigma$

$E$ and $F$ are regular expressions

\[
\begin{align*}
L(\emptyset) &= \emptyset \\
L(\varepsilon) &= \{\varepsilon\} \\
L(a) &= \{a\} \\
L(EF) &= \{ab \mid a \in L(E) \text{ and } b \in L(F)\} \\
L(E \mid F) &= L(E) \cup L(F) \\
L((E)) &= L(E) \\
L(E^*) &= L(E)^*
\end{align*}
\]
Precedence Rules

- Precedence rules help simplify regular expressions
  - Kleene closure has highest precedence
  - Concatenation has next highest
  - | has lowest precedence

- All operators associate left-to-right
Example

- Let $\Sigma = \{a, b\}$

- Find the strings in the language represented by the following regular expressions:

\[
\begin{align*}
    a & \quad | \quad b \\
    a^* & \\
    a & \quad | \quad a^*b \\
    (a \quad | \quad b)(a \quad | \quad b) & \\
    (a \quad | \quad b)^* & \\
    a(a \quad | \quad b)^*a & 
\end{align*}
\]
### Algebra of Regular Expressions

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>is commutative</td>
<td>$r</td>
</tr>
<tr>
<td>is associative</td>
<td>$(r</td>
</tr>
<tr>
<td>Concatenation is associative</td>
<td>$(rs)t = r(st)$</td>
</tr>
<tr>
<td>Concatenation distributes over</td>
<td>$r(s</td>
</tr>
<tr>
<td></td>
<td>$(s</td>
</tr>
<tr>
<td>$\epsilon$ is the identity</td>
<td>$\epsilon r = r = r \epsilon$</td>
</tr>
<tr>
<td>element for concatenation</td>
<td></td>
</tr>
<tr>
<td>Relation between $\ast$ and $\epsilon$</td>
<td>$(r</td>
</tr>
<tr>
<td>$\ast$ is idempotent</td>
<td>$r^{\ast\ast} = r^\ast$</td>
</tr>
</tbody>
</table>
Mathematically Describing Relational Operators

\[ \Sigma = \{ <, >, =, ! \} \]

\[ relop = < | > | <= | >= | == | != \]
Identifiers and Numbers

\[ \Sigma = \{ \text{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, } \} \]

\[ \text{letter} = \text{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z} \]

\[ \text{digit} = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \]

\[ \text{identifier} = \text{letter ( letter | digit)*} \]

\[ \text{number} = \text{digit digit*} \]
Finite Automata

A non-deterministic finite automaton (NFA) is a 5-tuple:

\[ \langle S, \Sigma, \phi, s_0, F \rangle \]

- \( S \) a set of states
- \( \Sigma \) a set of input symbols
- \( \phi \) a transition function \((S, \Sigma) \rightarrow S\)
- \( s_0 \) a distinguished state called the start state
- \( F \) a set of accepting or final states
NFA Representation

An NFA can be conveniently represented by both a directed graph and a table.

![NFA Diagram]

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>0</td>
<td>${0, 2}$</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Final states

- are double circled (graph)
- output a 1 (table)
An NFA can be built to recognize strings represented by a regular expression

(i.e., strings that are members of some regular set)
Given an NFA $M$, $L(M)$ is the language recognized by that machine.

If the NFA scans the complete string and ends in a final state, then the string is a member of $L(M)$.

We say $M$ accepts the string.

If the NFA scans the complete string and ends in a non-final state, then the string is not a member of $L(M)$.

We say $M$ rejects the string.

Because of non-determinism a string is accepted if there is a path to a final state; a string is rejected if there is no path to a final state.

Think about the NFA following all non-deterministic paths in parallel.
Deteminstic Finite Automata (DFA)

- A special case of an NFA
- Also called a finite state machine
- No state has an $\varepsilon$-transition
- $\forall s \in S$ and $\forall a \in \Sigma$, there is at most one edge labeled $a$ leaving $s$

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$</td>
<td>$d$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Discrete Mathematical Structures

Formal Languages
DFA Simulation

DFA() {
    s ← s₀;
    c ← nextchar();
    while c ≠ eof {
        s ← move(s, c);  —move is the φ : (S, Σ) → S function
        c ← nextchar();
    }
    if s ∈ F {
        return true;
    }
    return false;
}
\(\varepsilon\)-closure

- If \(s \in S\), then \(\varepsilon\)-closure\((s)\) is the set of states reachable from state \(s\) using only \(\varepsilon\)-transitions

- If \(V \subseteq S\), then \(\varepsilon\)-closure\((V)\) is the set of states reachable from some state \(s \in V\) using only \(\varepsilon\)-transitions
\[ \epsilon\text{-closure Computation} \]

\[
\text{StateSet } \epsilon\text{-closure}(\text{StateSet } T) \{
\text{result } \leftarrow T; \quad \text{stack } \leftarrow \emptyset; \quad \text{---stack is a stack of states}
\text{for all } s \in T \text{ do } \{
\text{stack.push}(s);
\}
\text{while stack } \neq \emptyset \{ 
\text{t } \leftarrow \text{stack.pop()};
\text{for each state } u \text{ with an edge from } t \text{ to } u \text{ labeled } \epsilon \text{ do } \{
\text{if } u \notin \text{result } \{
\text{result } \leftarrow \text{result } \cup u;
\text{stack.push}(u);
\}
\}
\text{return result;}
\}
\]
NFA() {

V ← ε-closure({s0});

nextchar();

c ← nextchar();

while c ≠ eof {

−move here returns the set of states to which there is a
−transition on input symbol c from some state s ∈ V

V ← ε-closure(move(V , c));

c ← nextchar();

}

if V ∩ F ≠ ∅ {

return true;

}

return false;
}
There are several strategies to build an NFA from a regular expression:

1. Parse the regular expression into its basic subexpressions
   - $\epsilon$ is a basic expression
   - an alphabet symbol is a basic expression
2. Create primitive NFAs for these subexpressions
3. Guided by the regular expression operators and parentheses, inductively combine the sub-NFAs into the composite NFA representing the complete regular expression

This is a syntax-directed approach.
For $\epsilon$, the NFA is

For $a \in \Sigma$, the NFA is

Observe that both of these NFAs have exactly one start state and one final state.
If $N(s)$ is the NFA for regular expression $s$, and $N(t)$ is the NFA for regular expression $t$, then $N(s \mid t)$ is
If $N(s)$ is the NFA for regular expression $s$, and $N(t)$ is the NFA for regular expression $t$, then $N(st)$ is
If $N(s)$ is the NFA for regular expression $s$, then $N(s^*)$ is
If $N(s)$ is the NFA for regular expression $s$, then $N((s)) = N(s)$ is
NFA $\rightarrow$ DFA

- NFAs are difficult to simulate in a computer program
  
  Non-determinism on a deterministic machine

- Fortunately, any NFA can be converted into an equivalent DFA
  
  - A process known as *subset construction* is used to create the DFA
  - Each state in the DFA is derived from the subset of the states in the NFA
  - If the NFA has $n$ states, its corresponding DFA may have up to $2^n$ states
    
    Fortunately, this theoretical maximum is rare in practice
Subset Construction

NFAtoDFA() {
  \( E \leftarrow \epsilon\text{-closure}\{s_0\} \); \( E\).mark \leftarrow \text{false}; \( D \leftarrow \{E\} \);
  \text{while} \: \exists \: T \in D \: \text{such that} \: T\text{.mark} = \text{false} \: \text{do} \{ \}
  \quad T\text{.mark} \leftarrow \text{true};
  \quad \text{for each} \: a \in \Sigma \: \text{do} \{ \}
  \quad \quad U \leftarrow \epsilon\text{-closure(move}(T, a))\};
  \quad \text{if} \: U \notin D \{ \}
  \quad \quad U\text{.mark} \leftarrow \text{false};
  \quad \quad D \leftarrow D \cup U;
  \quad \} \}
  \quad \text{DTran}[T][a] \leftarrow U; \}
  \}
}
DFA Minimization

Goal: Given a DFA \( M \), find a DFA \( M' \) such that \( M' \) exhibits the same external behavior as \( M \), but \( M' \) has fewer states than \( M \)

Reason: \( M' \) will be simpler and more efficient
DFA Minimization Procedure

1. Remove states unreachable from the start state

2. Ensure that all states have a transition on every input symbol (i.e., every element of \( \Sigma \))
   - Introduce a new “dead state” \( d \) if necessary
   - \( \forall a \in \Sigma, \phi(d, a) = d \) (i.e., \( \text{move}(d, a) = d \), for all \( a \))
   - \( \forall s \in S, \text{if } \exists a \text{ such that } \phi(s, a) \text{ is undefined, define } \phi(s, a) = d \)

3. Collapse equivalent states into a single, representative state
Equivalent States

- We say string $w$ distinguishes state $s$ from state $t$ if
  1. starting DFA $M$ in state $s$ and feeding it string $w$ we arrive at an accepting state, and
  2. starting DFA $M$ in state $t$ and feeding it string $w$ we arrive at an non-final state

  or vice-versa

- $w = \epsilon$ distinguishes any final state from any non-final state

- We must find all sets of states that can be distinguished by some input string

- Two states that cannot be distinguished by any input string are called equivalent states
DFA Minimization Algorithm (1)

\textbf{DFA minimize}(\text{DFA } M) \{ \\
\text{Part 1: Find equivalent states} \\
\Sigma \leftarrow M \cdot \Sigma; \quad \text{M’s alphabet} \\
S \leftarrow M \cdot S; \quad \text{M’s states} \\
F \leftarrow M \cdot F; \quad \text{M’s final states} \\
\phi \leftarrow M \cdot \phi; \quad \text{M’s transition function} \\
\Pi \leftarrow \{F, S - F\}; \quad \text{Partition states into two blocks: final and non-final states} \\
\Pi_{\text{old}} \leftarrow \emptyset; \\
\text{Iteratively partition the blocks until no further partitioning occurs} \\
\text{while } \Pi \neq \Pi_{\text{old}} \{ \\
\Pi_{\text{old}} \leftarrow \Pi; \\
\text{for each block } B \in \Pi \text{ do } \{ \\
\text{Partition } B \text{ into sub-blocks } B_1, B_2, \ldots, B_k \text{ such that two states } s \text{ and } t \\
\text{are in the same sub-block iff } \forall a \in \Sigma \text{ states } s \text{ and } t \\
\text{have transitions on } a \text{ to states in the same block of } \Pi; \\
\Pi \leftarrow (\Pi - B) \cup \{B_1, B_2, \ldots, B_k\} \\
\} \\
\}

DFA Minimization Algorithm (2)

Part 2: Build near-minimal DFA

\[ M' \cdot \Sigma \leftarrow \Sigma; \quad M' \cdot \mathcal{S} \leftarrow \emptyset; \quad M' \cdot \mathcal{F} \leftarrow \emptyset; \quad M' \cdot \phi \leftarrow \emptyset; \]

for each block \( B \in \Pi \) do \{ 
    Basically a block in \( \Pi \) becomes a state in \( M' \)
    Choose one state \( s \) in \( B \) to be the \textit{representative} of that block;
    \( M' \cdot \mathcal{S} \leftarrow M' \cdot \mathcal{S} \cup s; \)
\}

for each state \( s \in M' \cdot \mathcal{S} \) do \{ 
    Construct in the transition function for \( M' \)
    for each \( a \in \Sigma \) do \{ 
        if \( \phi(s, a) = t \) \{ 
            \( M' \cdot \phi(s, a) \leftarrow t' \in M' \cdot \mathcal{S} \) such that \( t' \)
            is the representative state of the block in \( \Pi \) that contains \( t; \)
        \}
    \}

The start state of \( M' \) is the representative state of the block in \( \Pi \) that contains
the start state of \( M; \)

for each state \( s \in M' \cdot \mathcal{S} \) do \{ 
    Assign final states
    if \( s \in \mathcal{F} \) \{ 
        \( M' \cdot \mathcal{F} \leftarrow M' \cdot \mathcal{F} \cup s; \) 
    \}
\}
**DFA Minimization Algorithm (3)**

**Part 3: Remove superfluous states**

if $M'.S$ contains a dead state $d$ {
    $M'.S \leftarrow M'.S - d$;
    for all $s \in M'.S$ do {
        if $\exists a \in \Sigma$ such that $M'.\phi(s,a) = d$ {
            $M'.\phi(s,a) \leftarrow$ undefined;
        }
    }
    for all $s \in M'.S$ do {
        if $s$ is unreachable from the start state in $M'$ {
            $M'.S \leftarrow M'.S - s$;
        }
    }
} return $M'$;

Remove any dead states

Prune unreachable states

The minimized DFA
Minimization Example

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State $a$</th>
<th>Next State $b$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- $a$ transitions are in red
- $b$ transitions are in blue

\[
\Pi_1 = \{\{2, 4\}, \{0, 1, 3\}\}
\]

\[
\Pi_2 = \{\{2\}, \{4\}, \{0, 1, 3\}\}
\]

\[
\Pi_3 = \{\{2\}, \{4\}, \{0, 1, 3\}\}
\]

\[
\Pi_2 = \Pi_3
\]
Minimal DFA

\[ \Pi_1 = \{\{2,4\},\{0,1,3\}\} \]

\[ \Pi_2 = \{\{2\},\{4\},\{0,1,3\}\} \]

\[ \Pi_3 = \{\{2\},\{4\},\{0,1,3\}\} \]

\[ \Pi_2 = \Pi_3 \]

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0'</td>
<td>2'</td>
<td>0'</td>
<td>1</td>
</tr>
<tr>
<td>2'</td>
<td>4'</td>
<td>0'</td>
<td>0</td>
</tr>
<tr>
<td>4'</td>
<td>4'</td>
<td>0'</td>
<td>0</td>
</tr>
</tbody>
</table>

- \(a\) transitions are in red
- \(b\) transitions are in blue
- \(\{0,1,3\} \Rightarrow \text{state } 0' \text{ in } M'\)
- \(\{2\} \Rightarrow \text{state } 2' \text{ in } M'\)
- \(\{4\} \Rightarrow \text{state } 4' \text{ in } M'\)
If $L \subseteq \Sigma^*$ is a language, the following four statements are equivalent:

1. $L$ is a regular language

2. $L$ can be represented by a regular expression

3. $L$ is accepted by some NFA

4. $L$ is accepted by some DFA
Limitations of Regular Languages

- Build a DFA to recognize
  \[ L = L(0^*1^*) \]

- Build a DFA to recognize
  \[ L = \{0^n1^n \mid n \in \mathbb{N}\} \]

- Not all languages are regular

- See the Pumping Lemma
The syntax of programming language constructs can be described by *context-free grammars* (CFGs).

- Relatively simple and widely used

- More powerful grammars exist
  - Context-sensitive grammars (CSG)
  - Type-0 grammars

  Both are too complex and inefficient for general use

- Backus-Naur Form (BNF) and extended BNF (EBNF) are a convenient way to represent CFGs
Advantages of CFGs

- Precise, easy-to-understand syntactic specification of a programming language

- Efficient parsers can be automatically generated for some classes of CFGs

- This automatic generation process can reveal ambiguities that might otherwise go undetected during the language design

- A well-designed grammar makes translation to object code easier

- Language evolution is expedited by an existing grammatical language description
Context-free Grammar (CFG) is a 4-tuple

\[ \langle V_N, V_T, s, P \rangle \]

- \( V_N \) is a set of non-terminal symbols
- \( V_T \) is a set of terminal symbols
- \( s \) is a distinguished element of \( V_N \) called the start symbol
- \( P \) is a set of productions or rules that specify how legal strings are built

\[ P \subseteq V_N \times (V_N \cup V_T)^* \]
CFG Elements

- **Terminals**: basic symbols from which strings are formed (typically corresponds to tokens from lexer)

- **Non-terminals**: syntactic variables that denote sets of strings and, in particular, denoting language constructs

- **Start symbol**: a non-terminal; the set of strings denoted by the start symbol is the language defined by the grammar

- **Productions**: set of rules that define how terminals and non-terminals can be combined to form strings in the language

\[ A \rightarrow bXYz \]
Symbol table interpreter

\[
G = \langle V_N, V_T, s, P \rangle
\]

\[
\begin{align*}
V_N &= \{ S \} \\
V_T &= \{ \text{new, id, num, insert, lookup, quit} \} \\
s &= S \\
P : S &\rightarrow \text{new id num} \\
\quad \quad \quad &\mid \text{insert id id num} \\
\quad \quad \quad &\mid \text{lookup id id} \\
\quad \quad \quad &\mid \text{quit}
\end{align*}
\]
Example

An arithmetic expression language

\[ G = \langle V_N, V_T, s, P \rangle \]

\[
\begin{align*}
V_N &= \{ E \} \\
V_T &= \{ \text{id}, +, *, (, ), - \} \\
s &= E \\
P : E &\rightarrow E + E \\
&\quad \mid E * E \\
&\quad \mid (E) \\
&\quad \mid -E \\
&\quad \mid \text{id}
\end{align*}
\]
A programming language construct

\[
stmt \; \rightarrow \; ;
\]

\[
| \; \textbf{if} \; ( \; expr \; ) \; stmt \; \textbf{else} \; stmt
\]

\[
| \; \textbf{while} \; ( \; expr \; ) \; stmt
\]

\[
| \; blk
\]

\[
| \; \textbf{id} \; = \; expr \; ;
\]

\[
blk \; \rightarrow \; \{ \; stmt^* \; \}
\]
• All regular languages are context-free

• Consider the regular expression

\[ a^*b^* \]

Let \( G = \langle \{A, B\}, \{a, b\}, A, \{A \to aA \mid B, B \to bB \mid \varepsilon\} \rangle \)
Producing a Grammar from a Regular Language

1. Construct an NFA from the regular expression

2. Each state in the NFA corresponds to a non-terminal symbol

3. For a transition from state $A$ to state $B$ given input symbol $x$, add a production of the form

   $A \rightarrow xB$

4. If $A$ is a final state, add the production

   $A \rightarrow \epsilon$
Parse Trees

- A graphical representation of a sequence of derivations
- Each interior node is a non-terminal and its children are the right side of one of the non-terminal’s productions
• If you read the leaves of the tree from left to right they form a sentential form
  – Also called the “yield” or “frontier” of the parse tree

• All the leaves need not be terminals; the parse tree may be incomplete

• Valid sentential forms can contain non-terminals
Comparing Context-free Grammars

- CFGs
- LR(k)
- LALR(1)
- SLR(1)
- LL(1)
Consider productions of the form $\alpha \rightarrow \beta$

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Criteria</th>
<th>Recognizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 3</td>
<td>Regular</td>
<td>$A \rightarrow a \mid aB$</td>
<td>Finite automaton</td>
</tr>
<tr>
<td>Type 2</td>
<td>Context-free</td>
<td>$A \rightarrow \alpha$</td>
<td>Push-down automaton</td>
</tr>
<tr>
<td>Type 1</td>
<td>Context-sensitive</td>
<td>$</td>
<td>\alpha</td>
</tr>
<tr>
<td>Type 0</td>
<td>Unrestricted</td>
<td>$\alpha \neq \epsilon$</td>
<td>Turing machine</td>
</tr>
</tbody>
</table>
Grammar Hierarchy

Unrestricted

Context-sensitive

Context-free

Regular

Type 3

Type 2

Type 1

Type 0