Languages, Grammars, and Machines

To understand a program, you must become both the machine and the program. (A. Perlis)

Some people, when confronted with a problem, think “I know, I’ll use regular expressions”. Now they have two problems. (Jamie Zawinski)
Languages and Grammars

Strings and Languages
Vocabulary ($V$): a finite set of symbols, e.g., $V = \{0, 1\}$
String ($w$): a finite sequence of symbols from $V$, e.g., $w_1 = 010$ and $w_2 = 11110$
Empty String ($\lambda$): the string of length 0.
Concatenation ($vw$): $vw$ is string $v$ then string $w$.
$w^n$: string $w$ repeated $n$ times, e.g., $0^3 = 000$
$V^*$: all possible strings using $V$.
Language ($L$): A subset of all possible strings $V^*$

Example: $L = \{1^m0^n \mid m \in \mathbb{N} \land n \in \mathbb{N}\}$

Phrase-Structure Grammars
A phrase-structure grammar $G$ consists of:
- vocabulary $V$, divided into two subsets: terminal symbols $T$ and nonterminal symbols $N$
- start symbol $S \in N$
- a finite set of productions $P$
Each production has the form $x \rightarrow y$, where $x \in V^*$ and $y \in V^*$. This means we can derive string $wyv$ from string $wxz$.

A string $w$ is an element of the language generated by $G$ if $w$ can be derived from $S$ by applying a sequence of productions.

Example Grammar 1
By convention, $S$ is the start symbol, and nonterminal symbols are in uppercase italics.

$S \rightarrow AB0$ Any $S$ can be replaced with $AB0$
$A \rightarrow BB$ Any $A$ can be replaced with $BB$
$B \rightarrow 01$ Any $B$ can be replaced with $01$
$AB \rightarrow 1$ Any $AB$ can be replaced with $1$

This grammar can only generate $10$ and $0101010$.

$S \Rightarrow AB0 \Rightarrow 10$
$S \Rightarrow AB0 \Rightarrow B BB0 \Rightarrow \ldots 0101010$

An equivalent grammar is $S \rightarrow 10$ and $S \rightarrow 0101010$.

Example Grammars 2
This grammar generates odd binary numbers. How would you change it to generate even numbers?

$S \rightarrow 1S$ Any $S$ can be replaced with $1S$
$S \rightarrow 0S$ Any $S$ can be replaced with $0S$
$S \rightarrow 1$ Any $S$ can be replaced with $1$

This grammar generates balanced parentheses. How would you include balanced braces?

$S \rightarrow SS$ Here is a derivation of $(())()$
$S \rightarrow (S) \Rightarrow SS \Rightarrow ()S \Rightarrow ()(S) \Rightarrow$
$S \rightarrow ()(SS) \Rightarrow ()(()S) \Rightarrow ()(()())$
Derivation Tree

A derivation tree or parse tree can represent a derivation. Here is a derivation tree for ( ) ( ) ( ).

Types of Grammars

Each type imposes restrictions on all productions.

- **Type 0**: No restrictions on productions.
- **Context-sensitive grammars** (Type 1): The right side cannot be shorter than the left side. Example: \( \{1^n 0^n 1^n \mid n \in \mathbb{N} \} \)
- **Context-free grammars** (Type 2): The left side is exactly one nonterminal symbol. Example: \( \{1^n 0^n \mid n \in \mathbb{N} \} \)
- **Regular grammars** (Type 3): A context-free grammar where the right side can have only one nonterminal symbol, only at the end. Example: \( \{1^m 0^n \mid m \in \mathbb{N} \land n \in \mathbb{N} \} \)

Finite-State Machines

- A **finite-state machine** inputs a string one symbol at a time.
- The machine starts in an initial state and can move to another state after each symbol.
- The next state is a function of the current state and the symbol that is input.
- Examples: two-way light switch, vending machine, combination lock, ATM, self-checkout, automated customer service

We discuss two types of FSMs: one type produces an output string; the other type accepts or rejects strings.

Finite-State Machines with Output

A finite-state machine with output consists of:

- \( S \): a set of states
- \( I \): an input alphabet (input symbols)
- \( O \): an output alphabet (output symbols)
- \( f : S \times I \rightarrow S \): a transition function from a state and input to a state.
- \( g : S \times I \rightarrow O \): an output function from a state and input to an output.
- \( s_0 \): an initial state

This type of FSM inputs a string and outputs a string.
State Tables and State Diagrams

A FSM can be represented by a state table or a state diagram. In this FSM, the output equals the input delayed by one bit.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( f )</td>
</tr>
<tr>
<td>( s_0 )</td>
<td>0</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>1</td>
</tr>
</tbody>
</table>

Start

\( s_0 \)

0,0

1,0

1,1

0,1

Finite-State Machines for Languages

A finite-state automaton (a FSM for recognizing a language) consists of:
- \( S \): a set of states
- \( I \): an input alphabet (input symbols)
- \( f : S \times I \rightarrow S \): a transition function from a state and input to a state.
- \( s_0 \): an initial state
- \( F \): final states (a subset of \( S \))

A FSA inputs a string and accepts the string if the last state is a final state.

Finite-State Automaton Example 1

This accepts all bit strings ending with a 1.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( f )</td>
</tr>
<tr>
<td>( s_0 )</td>
<td>0</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>1</td>
</tr>
</tbody>
</table>

Start

\( s_0 \)

0

1

\( s_1 \)

0,0

1,0

1,1

0,1

0,1

1,1

1,0

CS 2233 Discrete Mathematical Structures
Languages, Grammars, and Machines – 10

CS 2233 Discrete Mathematical Structures
Languages, Grammars, and Machines – 11

CS 2233 Discrete Mathematical Structures
Languages, Grammars, and Machines – 12

CS 2233 Discrete Mathematical Structures
Languages, Grammars, and Machines – 13
**Finite-State Automaton Example 2**

This accepts bit strings starting with a 0 and ending with a 1.

<table>
<thead>
<tr>
<th>Input</th>
<th>State</th>
<th>0</th>
<th>1</th>
<th>Final?</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial state</td>
<td>s0</td>
<td>s2</td>
<td>s1</td>
<td>no</td>
</tr>
<tr>
<td>first input was not a 0</td>
<td>s1</td>
<td>s2</td>
<td>s1</td>
<td>no</td>
</tr>
<tr>
<td>first input 0, last input 0</td>
<td>s2</td>
<td>s2</td>
<td>s3</td>
<td>no</td>
</tr>
<tr>
<td>first input 0, last input 1</td>
<td>s3</td>
<td>s2</td>
<td>s3</td>
<td>yes</td>
</tr>
</tbody>
</table>

---

**Regular Sets**

- ∅ represents the empty set.
- λ represents the set {λ}, consisting of the empty string.
- w represents the set {w}, consisting of the string w.
- \((E_1E_2)\) represents the set \(\{w_1w_2 \mid w_1 \in E_1 \land w_2 \in E_2\}\)
- \((E_1 \cup E_2)\) represents the set \(\{w \mid w \in E_1 \lor w \in E_2\}\)
- \(E^*\) represents the set \(\{w^n \mid w \in E \land n \in \mathbb{N}\}\)

The set of strings represented by a regular expression is called a regular set.

---

**Regular Expressions**

The set of regular expressions over a set of input symbols \(I\) is defined recursively as:

- ∅ is a regular expression.
- λ is a regular expression.
- \(w\) is a regular expression if \(w\) is a string using symbols in \(I\).
- \((E_1E_2), (E_1 \cup E_2), \text{ and } E_1^*\) are regular expressions if \(E_1\) and \(E_2\) are regular expressions.

Each regular expression represents a set of strings.

---

**Turing Machines**

A definition of computation is needed to study computation mathematically. A Turing machine is a primitive, yet general, computer with an infinite tape. In each “cycle”:

- the control unit reads the current tape symbol,
- writes a symbol on the tape,
- moves one position to the left or right, and
- switches to the next state.

The last three actions depend on the current state and tape symbol.
**Formal Definition**

Formally, a Turing machine $T$ consists of:

- $S$, a finite set of states.
- $I$, an alphabet, which is a finite set of symbols including the blank symbol $B$, and
- $f$, a state transition function, which is a partial function from $S \times I$ to $S \times I \times \{R, L\}$.
- $s_0$, the start state.

The Turing machine starts in state $s_0$ with the control unit reading the first nonblank symbol of the input string. There are an infinite number of blanks to the left and right of the input.

**Example 1 Continued**

Here is a Turing machine for incrementing a binary string.

- $f(s_0, 0) = (s_0, 0, R)$ in state $s_0$, move to the right
- $f(s_0, 1) = (s_0, 1, R)$ until you reach a blank, and
- $f(s_0, B) = (s_1, B, L)$ then switch to state $s_1$.
- $f(s_1, 1) = (s_1, 0, L)$ state $s_1$ moves to the left
- $f(s_1, 0) = (s_2, 1, L)$ changing 1s to 0s until a 0 or
- $f(s_1, B) = (s_2, 1, L)$ blank, changing it to a 1.

There are no transitions from $s_2$, so this is where you halt.

**Turing Machine Example 2**

Accept strings with equal numbers of 0s and 1s.

- $f(s_0, 0) = (s_1, M, R)$ in state $s_0$, change the first
- $f(s_0, 1) = (s_2, M, R)$ 0 or 1 to an $M$, and then switch
- $f(s_0, M) = (s_0, M, R)$ to state $s_1$ or $s_2$.
- $f(s_0, B) = $ accept Accept if all symbols are $M$.
- $f(s_1, 0) = (s_1, 0, R)$ in state $s_1$, change the first
- $f(s_1, 1) = (s_1, M, R)$ 1 to an $M$, and then switch to state $s_1$.
- $f(s_2, M) = (s_2, M, R)$ in state $s_2$, change the first
- $f(s_2, 1) = (s_2, 1, R)$ 0 to an $M$, and then switch to state $s_1$.
- $f(s_2, 0) = (s_2, M, L)$
- $f(s_1, 0) = (s_3, 0, L)$ in state $s_3$, move back to the
- $f(s_1, 1) = (s_3, 1, L)$ beginning of the string, and then
- $f(s_1, M) = (s_3, M, L)$ switch to state $s_0$.
- $f(s_1, B) = (s_3, B, R)$
Properties of Turing Machines

- A Turing machine can recognize a language iff it can be generated by a phrase-structure grammar.
- The Church-Turing Thesis: A function can be computed by an algorithm iff it can be computed by a Turing machine.