# 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: $L = \{1^m 0^n \mid m \in \mathbf{N} \wedge n \in \mathbf{N}\}$



## 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 $uyv$ from string $uxz$.

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 $0101010$.
$S \Rightarrow AB0 \Rightarrow 10$
$S \Rightarrow AB0 \Rightarrow BBB0 \Rightarrow \ldots 0101010$

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?

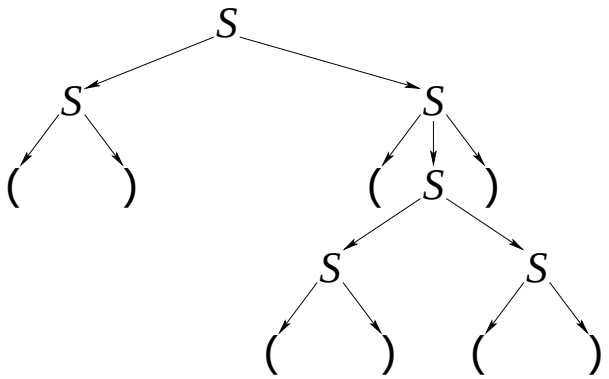
| | |
|---|---|
| $S \rightarrow 1S$ | Any $S$ can be replaced with $1S$ |
| $S \rightarrow 0S$ | Any $S$ can be replaced with $0S$ |
| $S \rightarrow 1$ | Any $S$ can be replaced with $1$ |

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

| | |
|---|---|
| $S \rightarrow SS$ | Here is a derivation of ()(()()) |
| $S \rightarrow (S)$ | $S \Rightarrow SS \Rightarrow ()S \Rightarrow ()(S) \Rightarrow$ |
| $S \rightarrow ()$ | $()(SS) \Rightarrow ()(()S) \Rightarrow ()(()())$ |

## 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: $\{1^n 0^n 1^n \mid n \in \mathbf{N}\}$
- *Context-free grammars* (Type 2): The left side is exactly one nonterminal symbol. Example: $\{1^n 0^n \mid n \in \mathbf{N}\}$
- *Regular grammars* (Type 3): A context-free grammar where the right side can have only one nonterminal symbol, only at the end. Example: $\{1^m 0^n \mid m \in \mathbf{N} \wedge n \in \mathbf{N}\}$





# Finite-State Machines



## 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:

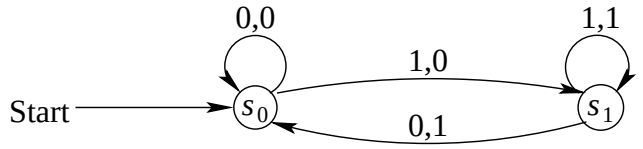
- $S$: a set of states
- $I$: an input alphabet (input symbols)
- $O$: an output alphabet (output symbols)
- $f : S \times I \to S$: a transition function from a state and input to a state.
- $g : S \times I \to O$: an output function from a state and input to an output.
- $s_o$: an initial state

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.

| | | Input | | | |
|---|---|---|---|---|---|
| | | $f$ | | $g$ | |
| Represents | State | 0 | 1 | 0 | 1 |
| last input was 0 | $s_0$ | $s_0$ | $s_1$ | 0 | 0 |
| last input was 1 | $s_1$ | $s_0$ | $s_1$ | 1 | 1 |

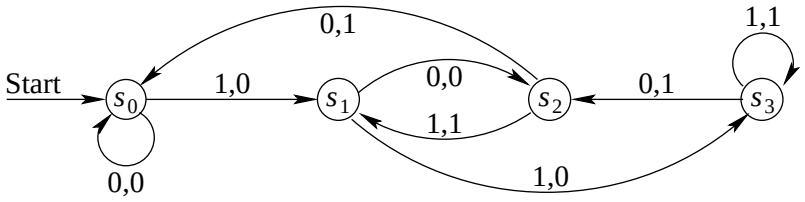





## FSM Example

The output equals the input delayed by two bits.

| | | Input | | | |
|---|---|---|---|---|---|
| | | $f$ | | $g$ | |
| Represents | State | 0 | 1 | 0 | 1 |
| last inputs were 00 | $s_0$ | $s_0$ | $s_1$ | 0 | 0 |
| last inputs were 01 | $s_1$ | $s_2$ | $s_3$ | 0 | 0 |
| last inputs were 10 | $s_2$ | $s_0$ | $s_1$ | 1 | 1 |
| last inputs were 11 | $s_3$ | $s_2$ | $s_3$ | 1 | 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.
- $s_o$: an initial state
- $F$: final states (a subset of $S$)

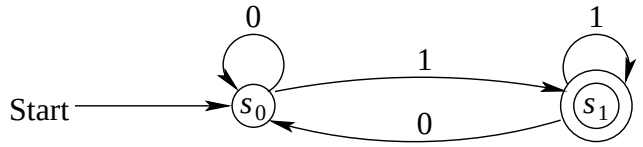
A FSA inputs a string and accepts the string if the last state is a final state.



## Finite-State Automaton Example 1

This accepts all bit strings ending with a 1.

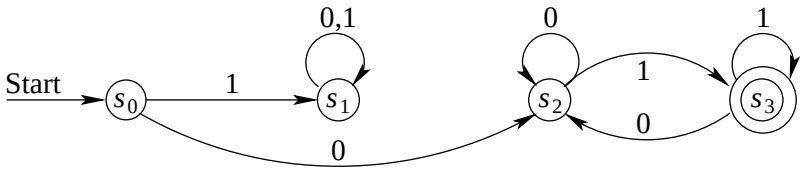
| | | Input | | |
|---|---|---|---|---|
| Represents | State | 0 | 1 | Final? |
| last input was a 0 | $s_0$ | $s_0$ | $s_1$ | no |
| last input was a 1 | $s_1$ | $s_0$ | $s_1$ | yes |



## Finite-State Automaton Example 2

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

| Represents | State | Input 0 | Input 1 | Final? |
|---|---|---|---|---|
| initial state | $s_0$ | $s_2$ | $s_1$ | no |
| first input was not a 0 | $s_1$ | $s_1$ | $s_1$ | no |
| first input 0, last input 0 | $s_2$ | $s_2$ | $s_3$ | no |
| first input 0, last input 1 | $s_3$ | $s_2$ | $s_3$ | yes |






# Regular Expressions



## Regular Expressions

The set of regular expressions over a set of input symbols $I$ is defined recursively as:

- $\emptyset$ is a regular expression.
- $\lambda$ is a regular expression.
- $w$ is a regular expression if $w$ is a string using symbols in $I$.
- $(E_1E_2)$, $(E_1 \cup E_2)$, and $E_1^*$ are regular expressions if $E_1$ and $E_2$ are regular expressions.

Each regular expression represents a set of strings.





## Regular Sets

- $\emptyset$ represents the empty set.
- $\lambda$ represents the set $\{\lambda\}$, consisting of the empty string.
- $w$ represents the set $\{w\}$, consisting of the string $w$.
- $(E_1E_2)$ represents the set $\{w_1w_2 \mid w_1 \in E_1 \wedge w_2 \in E_2\}$
- $(E_1 \cup E_2)$ represents the set $\{w \mid w \in E_1 \vee w \in E_2\}$
- $E^*$ represents the set $\{w^n \mid w \in E \wedge n \in \mathbf{N}\}$

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



## Examples

| Expression | Represents |
|---|---|
| $01$ | one string $01$ |
| $00 \cup 010$ | two strings $00$ and $010$ |
| $0(1 \cup \lambda)0$ | two strings $00$ and $010$ |
| $0^*1^*$ | any number of 0s followed by any number of 1s |
| $0^*(1 \cup \lambda)0^*$ | strings with at most one $1$ |
| $(0^* \cup 0^*10^*)$ | strings with at most one $1$ |
| $(0 \cup (1(01^*0)^*1))^*$ | binary numbers divisible by $3$ |



# Turing Machines



## 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.



## Turing Machine Example 1

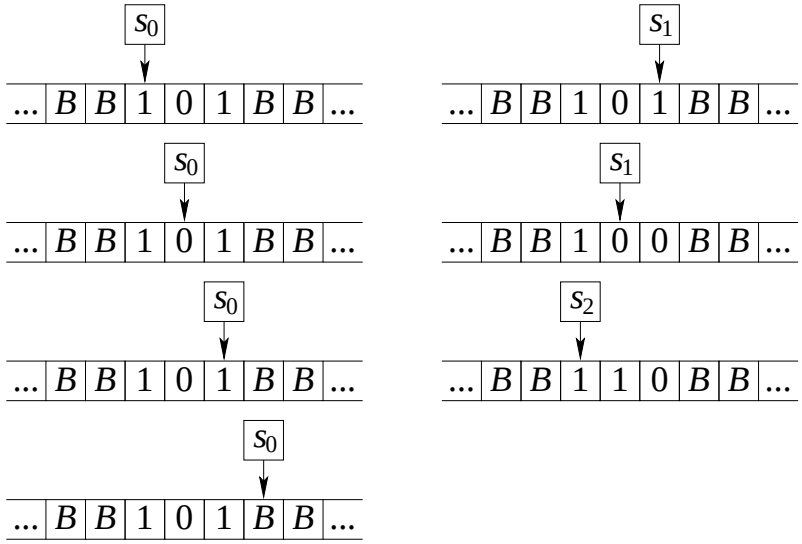
Here is a Turing machine for incrementing a binary string.

| | |
|---|---|
| $f(s_0, 0) = (s_0, 0, R)$ | 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. |





## Example 1 Continued






## 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 $M$. |
| $f(s_1, M) = (s_1, M, R)$ | In state $s_1$, change the first |
| $f(s_1, 0) = (s_1, 0, R)$ | 1 to an $M$, and then switch to |
| $f(s_1, 1) = (s_3, M, L)$ | state $s_3$. |
| $f(s_2, M) = (s_2, M, R)$ | In state $s_2$, change the first |
| $f(s_2, 1) = (s_2, 1, R)$ | 0 to an $M$, and then switch to |
| $f(s_2, 0) = (s_3, M, L)$ | state $s_3$. |
| $f(s_3, 0) = (s_3, 0, L)$ | In state $s_3$, move back to the |
| $f(s_3, 1) = (s_3, 1, L)$ | beginning of the string, and then |
| $f(s_3, M) = (s_3, M, L)$ | switch to state $s_0$. |
| $f(s_3, B) = (s_0, B, R)$ | |

## 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.