Models of Computation

Chapter 13
Languages, Grammars and Machines

We will focus on an abstraction of computation which is generating and recognizing sets of strings over some alphabet.

A grammar is a set of rewriting rules that generate a set of strings.

A machine is an abstract computing device (with states) that decides if its input string is in a certain set.

Grammars

A grammar $G = (V, T, S, P)$ consists of:

- Set $V$ (vocabulary): set of symbols
- Set $T$ (terminal symbols)
- Element $S \in V$ (start symbol)
- Set $P$ (productions)

A production is a rule for replacing one substring with another substring.

$N = V - T$ is the set of nonterminal symbols.
Example

\[ G = (V, T, S, P) \] where:

Vocabulary \( V = \{a, b, A, B, S\} \)

Terminal symbols \( T = \{a, b\} \)

Nonterminal symbols \( N = V - T = \{A, B, S\} \)

Productions \( P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\} \)

Grammar Derivations

Let \( G \) be a grammar.

- If \( w = lzr \) is a string over \( V \)
  and \( z \rightarrow z' \) is a production of \( G \),
  then \( w' = lz'r \) is directly derivable from \( w \).
  Notation: \( w \rightarrow w' \)

- If \( w_0 \rightarrow w_1, w_1 \rightarrow w_2, \ldots, w_{n-1} \rightarrow w_n \),
  then \( w_n \) is derivable from \( w_0 \).
  Notation: \( w_0 \rightarrow ^* w_n \)
Grammar Derivations

Example: In vocabulary \( V = \{a, b, A, B, S\} \),

\[ ABa \Rightarrow *abababa \text{ why?} \]

\[ \begin{align*}
ABa & \rightarrow Aaba \quad \text{by production } B \rightarrow ab \\
& \rightarrow BBaba \quad \text{by production } A \rightarrow BB \\
& \rightarrow Bababa \quad \text{by production } B \rightarrow ab \\
& \rightarrow abababa \quad \text{by production } B \rightarrow ab
\end{align*} \]

Language Generated by a Grammar

The language \( L(G) \) generated by grammar \( G \) is the set of all strings of terminals derivable from the start symbol \( S \).

Depending on the specifics of the productions:

- \( L(G) \) might be \( \emptyset \)
- \( L(G) \) might be finite
- \( L(G) \) might be infinite

So, a grammar can be a concise (finite) way to describe an infinite set of strings.
Describing the Language of a Grammar

How to describe in words the language of a grammar.
Example: \( G = (V, T, S, P) \) where

\( V = \{S, A, a, b\} \)
\( T = \{a, b\} \)

Start symbol is \( S \)
\( P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\} \)
Then \( L(G) = \{b, aaa\} \).

Another Grammar Example

\( G = (V, T, S, P) \) where
\( V = \{S, 0, 1\} \)
\( T = \{0, 1\} \)
start symbol is \( S \)
\( P=\{S \rightarrow 0S, S \rightarrow S1, S \rightarrow \lambda\} \)

What is \( L(G) \)?
Another Grammar Example

Every string generated by this grammar consists of some number of 0’s followed by some number of 1’s.

Every string consisting of some number of 0’s followed by some number of 1’s can be generated by this grammar.

Thus \( L(G) = \{ 0^m1^n \mid m \in \mathbb{Z}, n \in \mathbb{Z} \} \).

Types of Grammars

By putting different constraints on the form of the productions, we get different types of grammars.

Regular grammar:

Every production must be in one of these forms:

\[ S \rightarrow \lambda \] (start symbol goes to the empty string)
\[ A \rightarrow aB \] (a single nonterminal goes to a single terminal followed by a single nonterminal)
\[ A \rightarrow a \] (a single nonterminal goes to a single terminal)
Types of Grammars

By putting different constraints on the form of the productions, we get different types of grammars.

**Context-free grammar:**
The LHS of every production must be a single nonterminal

**Unrestricted grammar:**
No restrictions on the productions

---

Derivation Trees

A derivation of a context-free (or regular) grammar can be represented by a tree, called a derivation tree or parse tree.

- The root is the start symbol
- Each internal vertex is a nonterminal symbol
- Each leaf is a terminal symbol
- Each vertex and its children correspond to the application of a production
Applications of Grammars

The earliest application of grammars was in the study of natural languages:

- Panini developed a grammar for Sanskrit (with almost 4000 rules) in 4th century B.C.
- String matching (search for a string in a document)
- Lexical analysis (transform input stream into a stream of tokens to be parsed)
- Algorithms to parse a string (steps in translating program into machine code)

Finite-State Machines

A finite-state machine is an abstract model of a computing entity. It has:

- A finite set of states
- A starting state
- A transition function that assigns a next state

Applications:

spell checkers, parts of compilers, speech recognition algorithms, network protocols etc.
Finite-State Automata

A finite-state automaton $M = (S, I, f, s_0, F)$ has

- **finite** set $S$ of states
- **finite** set $I$, the input alphabet
- transition function $f : S \times I \rightarrow S$ (assigns next state)
- **start state** $s_0$ in $S$
- subset $F$ of $S$, the final, or accepting states

Representing Finite-State Automata

We can represent a finite-state automaton with a **table** or a **state diagram** (a directed graph with labels on edges)

<table>
<thead>
<tr>
<th>current state</th>
<th>input</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>$S_1$</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>$S_3$</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>
Extending Transition Function Notation

We can also define the transition function \( f \) on ordered pairs of states and strings of input characters.

Suppose \( x = x_1x_2 \ldots x_k \) is a string over the input alphabet \( I \).

- Let \( s_1 \) be any state.
- Let \( s_2 \) be the result of applying \( f \) to \((s_1, x_1)\).
- Let \( s_3 \) be the result of applying \( f \) to \((s_2, x_2)\).
- Until reaching state \( s_{k+1} \), the result of applying \( f \) to \((s_k, x_k)\).
- We define \( f(s_1, x) \) to be \( s_{k+1} \).

Recognition by Finite-State Automata

Let \( M = (S, I, f, s_0, F) \) be a finite-state automaton. String \( x \) is recognized (or accepted) by \( M \) if \( f(s_0, x) \in F \).

The language \( L(M) \) recognized (or accepted) by \( M \) is the set of all strings recognized by \( M \).
Examples of Languages Recognized by FSAs

Equivalent Finite-State Automata

Two finite-state automata are equivalent if they recognize the same language.
The previous definition of a finite-state automaton is deterministic:

Given an input state and input character, the transition function produces only one next state.

Sometimes it is useful to allow a choice of next states!

A non-deterministic finite-state automaton $M = (S, I, f, s_0, F)$ has:

- finite set $S$ of states
- finite set $I$, the input alphabet
- transition function $f : S \times I \rightarrow P(S)$ (assigns a set of next states to every pair of state and input)
- start state $s_0$ in $S$
- subset $F$ of $S$, the final, or accepting states
Representing Nondeterministic FSAs

We can use either a table or a state diagram.

<table>
<thead>
<tr>
<th>current state</th>
<th>input</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>0</td>
<td>${S_0, S_1}$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>${S_3}$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>0</td>
<td>${S_0}$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>${S_1, S_3}$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>${S_0, S_2}$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>0</td>
<td>${S_0, S_1, S_2}$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>${S_1}$</td>
</tr>
</tbody>
</table>

Recognition by Nondeterministic FSA

Let $M = (S, I, f, s_0, F)$ be a nondeterministic finite-state automaton.

String $x$ is accepted by $M$ if there exists a state in $F$ that can be reached from $s_0$ using $x$.

The language $L(M)$ accepted by $M$ is the set of all strings recognized by $M$. 
Nondeterministic Recognition Example

The language recognized by this nondeterministic finite-state automaton is \( \{0^n, 0^n01, 0^n11 : n \geq 0\} \).

Regular Expressions

Regular expressions can also generate the same collection of sets as regular grammars and FSAs.

Regular expression over a finite set \( I \):

**Basis:** The symbols \( \emptyset, \lambda \), and \( x \in I \) are regular expressions.

**Inductive step:** If \( A \) and \( B \) are regular expressions, then so are the symbols \( (AB), (A \cup B), \) and \( A^* \).
Regular Expressions

Each regular expression represents a set of strings over $\Sigma$:

- $\emptyset$ represents the empty set (no strings)
- $\lambda$ represents the set $\{\lambda\}$ (one element, the empty string)
- $x$ represents the set $\{x\}$ (one element, the string $x$)
- $(AB)$ represents the concatenation of the sets represented by $A$ and $B$. (strings of the form $xy$, $x$ is in the set represented by $A$, and $y$ is in the set represented by $B$)

- $(A \cup B)$ represents the union of the sets represented by $A$ and $B$
- $A^*$ represents the Kleene closure of the set represented by $A$. (strings formed by concatenating any number of strings in the set represented by $A$)
Computer Sci. Applications of Regular Expressions

Regular expressions are a powerful method for describing patterns.
Mechanisms to describe patterns using regular expressions are provided in:
• Operating system utilities (awk and grep in Unix)
• Programming languages (Perl)
• Text editors

The Turing Machine

A Turing machine (TM) is a finite-state automaton with an infinitely long memory (“tape”).

TMs can be used to recognize sets and to compute functions.

TM is the most general model of computation and are a good formal model for real computers
Definition of the Turing Machine

A Turing machine $T = (S, I, f, s_0)$ consists of:

- **Finite** set $S$ of states
- **Finite** set $I$, the alphabet, containing the blank symbol $B$
- **Partial function** $f: S \times I \rightarrow S \times I \times \{R, L\}$, the transition rules (need not be defined for all pairs)
- Start state $s_0$ in $S$

Representation of a Turing Machine

Tape is infinite in both directions.
Only finitely many nonblank cells at any time.
How does a Turing Machine Work?

It starts with the control unit in its start state $s_0$, reading the left-most nonblank symbol on the tape.

At each step:
1. Read the current tape symbol $x$
2. Compute $f(s, x) = (s', x', d)$, ($s$ is the current state)
3. Write $x'$ into the current tape cell
4. Change the state of control unit to $s'$
5. Move one cell in direction $d$ (left or right)

If $f$ is not defined for $(s, x)$, then the Turing machine halts.

Example of Turing Machine Computation
Recognizing Sets and Computing Functions with Turing Machines

A Turing machine $T = (S, I, f, s_0)$ recognizes a string $x$ if, when starting in $s_0$ and with $x$ written on the tape, $T$ halts in a final state.

$T$ recognizes a set $A$ if it recognizes all, and only, strings in $A$.  

End