

Formal Languages

Andreas Klappenecker

Motivation

The syntax of programming languages (such as Algol, C, Java) are described by phrase structure grammars. The grammars are also used in the specification of data exchange formats.

A grammar specifies a formal language. The advantage is that a fairly small grammar can describe a fairly complex language.

Formal languages are also a convenient tool in computational complexity theory, as we will see.

Notation

Let V be a set.

We denote by V^* the set of strings over V .

A language over the alphabet V is a subset of V^* .

The empty string is denoted by λ .

Phrase-Structure Grammars

A phrase-structure grammar G consists of

- a set V called the **vocabulary**,
- a subset T of V consisting of **terminal symbols**,
[$N=V \setminus T$ is called the set of **nonterminal symbols**]
- a distinguished nonterminal element S in N , called **start symbol**
- and a finite set of **productions** (or rules).

We denote this data by $G=(V,T,S,P)$.

Productions

The productions are term-rewriting rules that specify how a part of string can be modified.

A **production rule** is of the form

$$A \rightarrow B$$

where A is string in V^* containing at least one **nonterminal symbol**, and B is a string in V^* .

The production rule $A \rightarrow B$ specifies that A can be replaced by B within a string.

Main Idea

We begin with the start symbol S , and then repeatedly apply the productions to transform the current string of symbols into a new string of symbols.

Once we reach a string s that consists only of terminal symbols, then this procedure terminates.

We say that s is derivable from S .

Derivable

Consider a grammar that contains the productions

$$A \rightarrow a \quad \text{and} \quad A \rightarrow aAa$$

Given a string $AAab$, we can apply the first production to obtain $Aaab$, and the second production to get $aAaaab$. Applying the first production, we get $aaaaab$.

We write $A \Rightarrow B$ iff the string B **can be derived** from A by applying a single production rule.

We will write $A \Rightarrow^* B$ if the string B can be derived from A by a finite sequence of production rules.

Thus, we have shown that $AAab \Rightarrow^* aaaaab$

Example 0

Consider the grammar $G = (V, T, S, P)$ with

$$V = \{S\}$$

$$T = \{1, (,), +, \}$$

$$P = \{S \rightarrow (S + S), S \rightarrow 1\}$$

We have $S \Rightarrow 1$,

$$S \Rightarrow (S + S) \Rightarrow (S + 1) \Rightarrow (1+1)$$

$$S \Rightarrow (S + S) \Rightarrow ((S + S) + S) \Rightarrow^* ((1+1)+1)$$

Derivability Relation

Notice that \Rightarrow is a relation on V^* . It relates a string in V^* with a string that can be obtained from it by applying a single production rule. Since many production rules may apply, \Rightarrow is in general not a function.

The relation \Rightarrow^* is the so-called transitive closure of \Rightarrow , that is, the smallest transitive relation containing the relation \Rightarrow .

Language

Let $G=(V,T,S,P)$ be a phrase-structure grammar.

The set $L(G)$ of strings that can be derived from the start symbol S using production from P is called the **language** of the grammar G .

In other words,

$$L(G) = \{ s \text{ in } T^* \mid S \Rightarrow^* s \}$$

Example 1

Let G be a grammar with vocabulary $V = \{S, A, a, b\}$,

set of terminal symbols $T = \{a, b\}$

start symbol S

set of production rules $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$

$S \Rightarrow b$

$S \Rightarrow aA \Rightarrow aaa$

Are b and aaa all terminal strings in $L(G)$?

Example 1: Derivation Tree

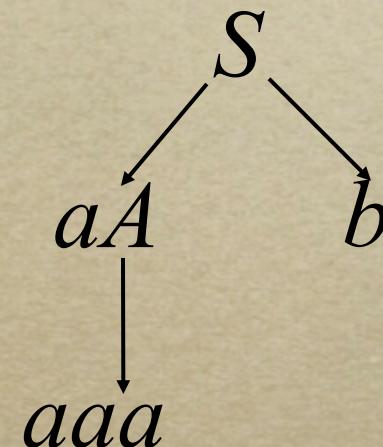
Recall that $G = (\{S, A, a, b\}, \{a, b\}, S, P)$

where

$$P = \{ S \rightarrow aA, S \rightarrow b, A \rightarrow aa \}$$

Simply form the tree with all derivations starting from S .

Thus, $L(G) = \{aaa, b\}$



Example 2

Let $G = (V, T, S, P)$ be a grammar with $T = \{a, b\}$ and
 $P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\}.$

$S \Rightarrow ABa \Rightarrow BBBa \Rightarrow abBBa \Rightarrow ababBa \Rightarrow abababa$

$S \Rightarrow ABa \Rightarrow Aaba \Rightarrow BBaba \Rightarrow^* abababa$

$S \Rightarrow ABa \Rightarrow ba$

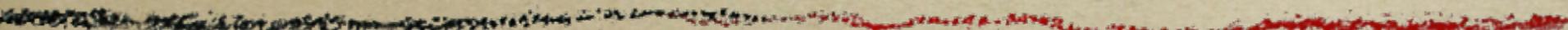
$L(G) = \{ ba, abababa \}$

Problem

In general, we would like to have an algorithm that can decide whether a given string s belongs to the language $L(G)$ or not. This is not always possible, not even in principle.

The problem is that the production rules are too flexible. We will now consider more restricted forms of grammars that allow one - at least in principle - to write an algorithm to decide membership in $L(G)$.

Regular Grammars



Regular Grammars

In a regular grammar, all productions are of the form

- a) $S \rightarrow$ empty string or
- b) $A \rightarrow aB$, where A and B are nonterminal symbols and a is a terminal symbol
- c) $A \rightarrow aB$, where A is a nonterminal symbol and a is a terminal symbol

Notation

Instead of writing multiple productions such as

$$A \rightarrow Ab$$

$$A \rightarrow Aba$$

we can combine them into a single production,
separating alternatives with |, as follows:

$$A \rightarrow Ab | Aba$$

Test Yourself...

Consider the grammar $G = (V, T, S, P)$ where
 $V = \{S, A, 0, 1\}$,
 $T = \{0, 1\}$,
 S is the start symbol, and
 P has the rules

$$S \rightarrow 0S \mid 1A \mid 1 \mid \text{empty-string}$$

$$A \rightarrow 1A \mid 1$$

Determine $L(G)$.

$$L(G) = \{ 0^m 1^n \mid m \geq 0, n \geq 0 \}$$

Regular Languages

A language generated by a regular grammar is called a **regular language**.

Warning: A regular language might also be generated by a different grammar that is not regular (and often it is more convenient to do so).

Example

Consider the grammar

$G = (\{S, 0, 1\}, \{0, 1\}, S, P)$ with

$P = \{S \rightarrow 11S, S \rightarrow 0\}$

Is G a regular grammar?

No, but $L(G)$ is a regular language,

since $L(G) = L(G')$ where

$G' = (\{S, T, 0, 1\}, \{0, 1\}, S, P')$

$P' = \{S \rightarrow 1T, T \rightarrow 1S, S \rightarrow 0\}$

Applications of Regular Grammars

Applications of regular grammars include:

- algorithms to search text for certain patterns (using regular expressions)
- part of a compiler that transforms an input stream into a stream of tokens (the so-called tokenizer). The purpose of the tokens is to group characters together into entities that have more meaning, such as "variable" or "signed integer".

Context Free Grammars

Context Free Grammars

A grammar $G = (V, T, S, P)$ is called **context free** if and only if all productions in P are of the form

$$A \rightarrow B$$

where A is a single nonterminal symbol and B is in V^* .

The reason this is called “context free” is that the production $A \rightarrow B$ can be applied whenever the symbol A occurs in the string, no matter what else is in the string.

Example 1

Consider the grammar $G = (\{S,a,b\}, \{a,b\}, S, P)$

where $P = \{ S \rightarrow ab \mid aSb \}$.

Then $L(G) = \{ a^n b^n \mid n \geq 1 \}$

The language $L(G)$ is not regular.

Thus, the set of regular languages are a proper subset of the set of context free languages.

Test Yourself

X1 Find a context-free grammar G for the language

$$L(G) = \{ ww^R \mid w \text{ in } \{a,b\}^* \}$$

where w^R is the string w reversed. For example, if $w=abc$, then $w^R = cba$.

$$G = (\{S,a,b\}, \{a,b\}, S, \{ S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \lambda \})$$

Context-Sensitive Grammars

Context Sensitive Grammars

In a context sensitive grammar, all productions are of the form

- a) $|A \rightarrow lwr$ where A is a nonterminal symbol, l, w, r are strings in V^* ; l and r can be empty, but w must be a nonempty string.
- b) Can contain $S \rightarrow \lambda$ if S does not occur on RHS of any production.

Example 1

The language $\{ 0^n 1^n 2^n \mid n \geq 0 \}$ is a context-sensitive language.

Indeed, the grammar $G = (V, T, S, P)$

with $V = \{0, 1, 2, S, A, B, C\}$, $T = \{0, 1, 2\}$, and

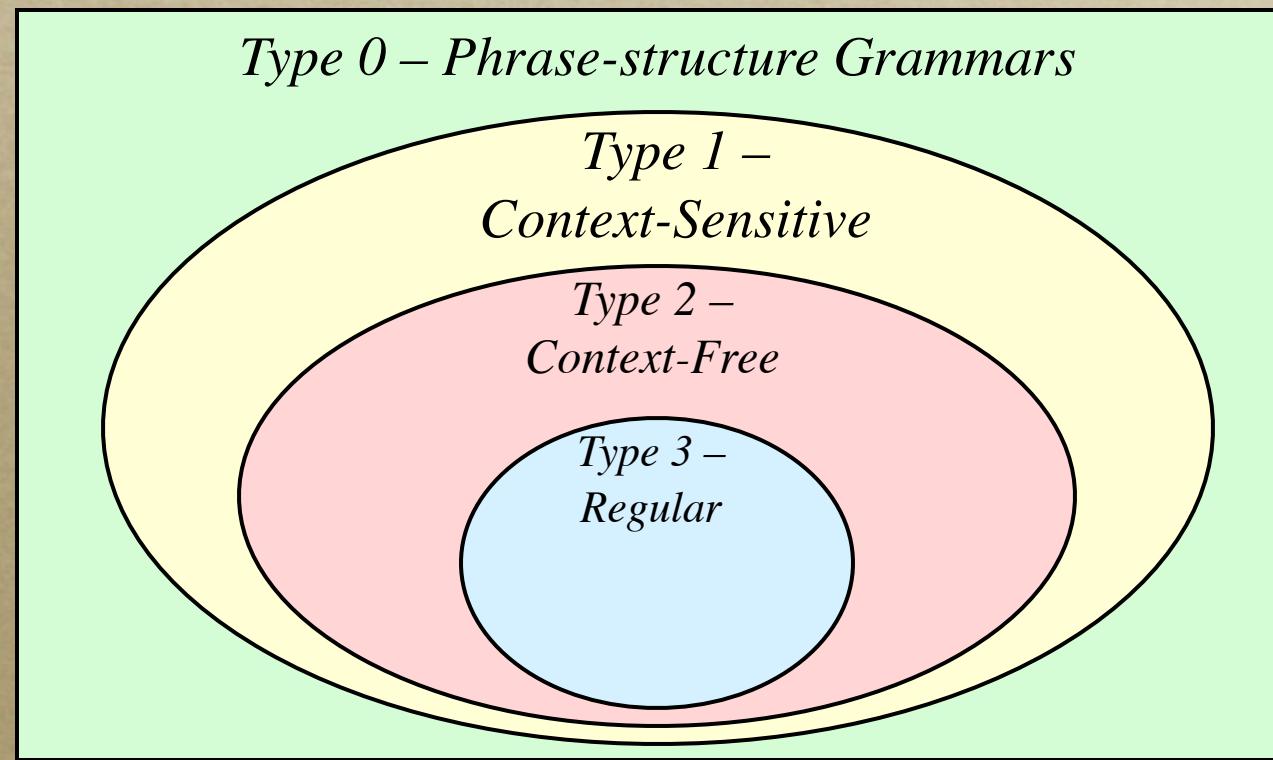
$P = \{ S \rightarrow C, C \rightarrow 0CAB, C \rightarrow OAB, S \rightarrow \lambda, BA \rightarrow AB,$
 $OA \rightarrow 01, 1A \rightarrow 11, 1B \rightarrow 12, 2B \rightarrow 22 \}$

generates this language.

$S \Rightarrow^* 0CAB \Rightarrow 00ABAB \Rightarrow 00AABB \Rightarrow 001ABB \Rightarrow 0011BB$
 $\Rightarrow 00112B \Rightarrow 001122.$

Types of Grammars

Chomsky Hierarchy



Defining the PSG Types

Type 0: Any PSG

Type 1: Context-Sensitive PSG:

Production are of the form $|Ar \rightarrow |w$ where A is a nonterminal symbol, and w a nonempty string in V^* . Can contain $S \rightarrow \lambda$ if S does not occur on RHS of any production.

Type 2: Context-Free PSG:

Productions are of the form $A \rightarrow B$ where A is a nonterminal symbol.

Type 3: Regular PSGs:

Productions are of the form $A \rightarrow aB$ or $A \rightarrow a$ where A,B are nonterminal symbols and a is a terminal symbol. Can contain $S \rightarrow \lambda$.

Test Yourself

X1 Recall that any formal language over an alphabet $\{0,1\}$ is a subset of $\{0,1\}^*$. Is every formal language of type 0?

Since the alphabet and the production rules are finite, the set of all type 0 languages is countable. On the other hand, the cardinality of the set $\{0,1\}^*$ is uncountable. Thus, many formal languages are not of type 0.

Modeling Computation

Transition Functions

Let S be the state space of a computer (that is, a state describes the contents of memory, cache, and registers).

Let I and O denote the set of input and output symbols. In each time step, the computer receives an input symbol i in I and produces an output symbol o in O .

The computer can be modeled by a transition function

$$T: S \times I \rightarrow S \times O$$

Given the old state and the input, the computer processes this information, creates a new state, and produces an output.

Language Recognition Problem

Language Recognition Problem:

Let $G = (V, T, S, P)$ be a grammar. Given a string s in T^* , is the string s contained in $L(G)$?

Computation vs. Language Recognition

Let T be the transition function of a computer. We may assume that the input, output, and state of the computer are given by bit strings. Let $B = \{0,1\}$. Then

$$T: B^* \rightarrow B^*$$

For a given input a , the output is $b=T(a)$. The i -th output bit has values 0 or 1. Let L_i be the language

$$L_i = \{ x \text{ in } I \mid T(x)_i = 1 \}$$

Thus, the language recognition problem a in L_i simply gives the value of the i -th output bit.

Conclusion

The language recognition problem is as general as our notion of computation!

Finite State Machines with Output

Vending Machine

A vending machine accepts nickels, dimes, and quarters. A drink will cost 30 cents. Change will be immediately given for any excess funds. When at least 30 cents have been deposited and any excess has been refunded, the customer can

- a) push an orange button to receive orange juice.
- b) push a red button to receive apple juice.

Vending Machine

We can model our vending machine as follows:

We have seven different states s_0, \dots, s_6 .

The state s_i means that $5i$ cents have been deposited for all i in the range $0 \leq i < 6$.

The vending machine will dispense a drink only in the state s_6 .

We can model the behavior of the vending machine by specifying the transition function. The inputs are:

5 (nickel), 10 (dime), 25 (quarter), O (orange button),
R (red button).

Vending Machine

© The McGraw-Hill Companies, Inc. all rights reserved.

Figure courtesy of McGrawHill

TABLE 1 State Table for a Vending Machine.

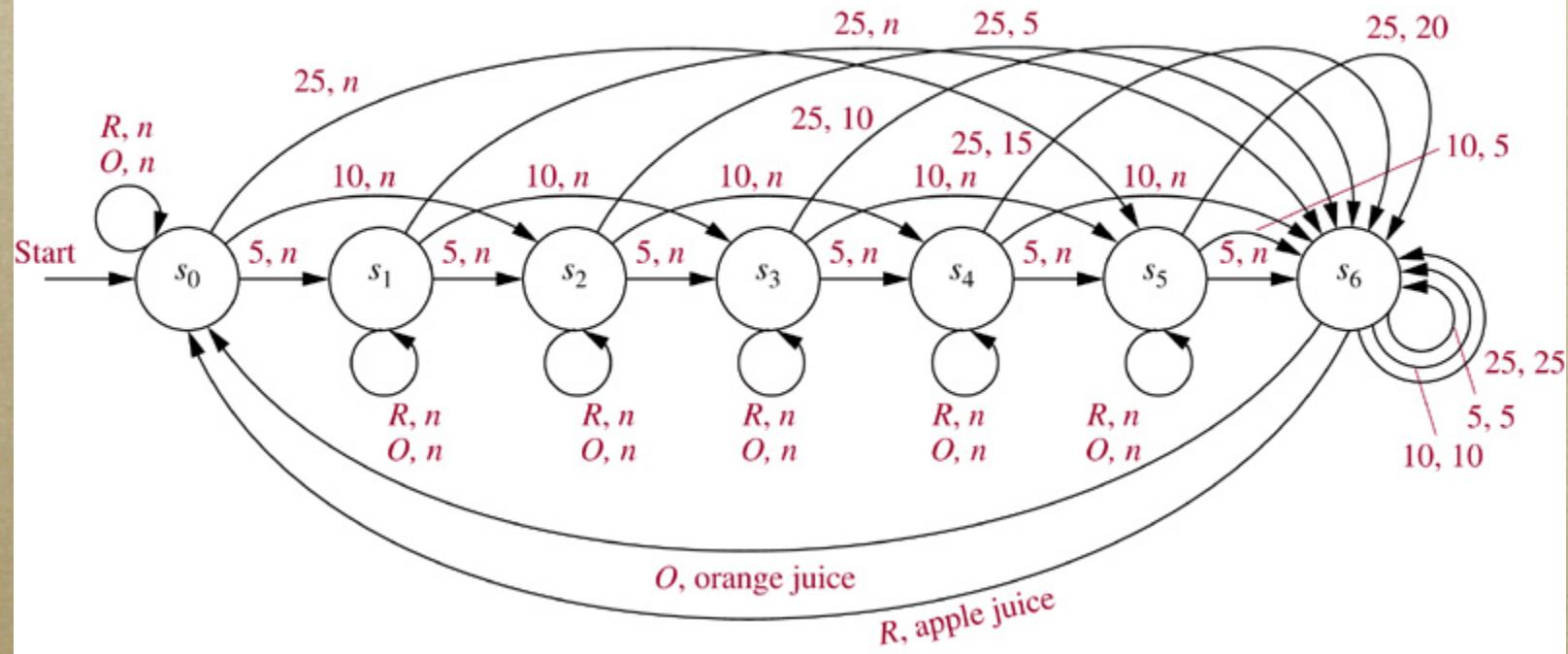
State	Next State					Output				
	5	10	25	O	R	5	10	25	O	R
s_0	s_1	s_2	s_5	s_0	s_0	n	n	n	n	n
s_1	s_2	s_3	s_6	s_1	s_1	n	n	n	n	n
s_2	s_3	s_4	s_6	s_2	s_2	n	n	5	n	n
s_3	s_4	s_5	s_6	s_3	s_3	n	n	10	n	n
s_4	s_5	s_6	s_6	s_4	s_4	n	n	15	n	n
s_5	s_6	s_6	s_6	s_5	s_5	n	5	20	n	n
s_6	s_6	s_6	s_6	s_0	s_0	5	10	25	OJ	AJ

Example: Machine is initially in state s_0 . If a dime is inserted, then it moves to the state s_2 and outputs nothing. If a quarter is then inserted, then it will move to s_6 and output a nickel of change. If you then press O, then machine will move to state s_0 and output

Vending Machine

© The McGraw-Hill Companies, Inc. all rights reserved.

Figure courtesy of McGrawHill



Finite State Machine

A finite state machine M is given by $M=(S, I, O, f, g, s_0)$,
where:

S is the set of states

I is input alphabet

O is output alphabet

f is the transition function that assigns each (state, input) pair a
new state

g is output function that assigns each (state, input) pair an
output.

s_0 is the initial state

Adder

When adding two binary numbers, we can process the numbers from the least significant bit to the most significant bit. For each bit, we carry out the addition. We keep the information about the carry in the state of the machine.

s_0 when the carry in is 0,

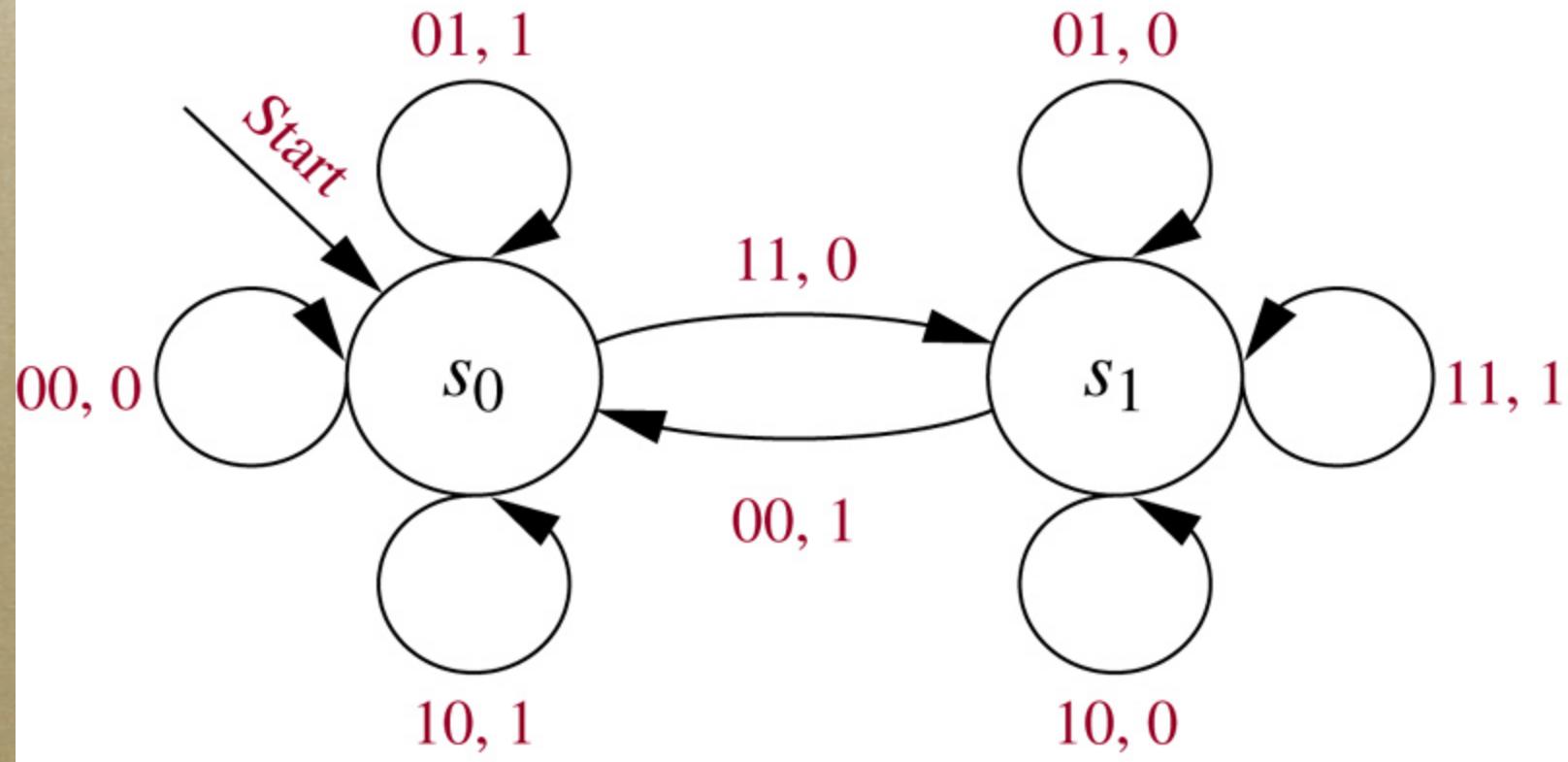
s_1 when the carry in is 1.

Curr. State	Input	Output	Next State
s_0	0 0	0	s_0
s_0	0 1	1	s_0
s_0	1 0	1	s_0
s_0	1 1	0	s_1

Curr. State	Input	Output	Next State
s_1	0 0	1	s_0
s_1	0 1	0	s_1
s_1	1 0	0	s_1
s_1	1 1	1	s_1

Adder

© The McGraw-Hill Companies, Inc. all rights reserved.



Language Recognition with FSMs

Let M be a finite state machine with input alphabet I .

Let L be a formal language with $L \subseteq I^*$.

We say that M accepts the language L if and only if
(x belongs to L if and only if the last output bit
produced by M when given x as an input is 1).

In other words, a) an input string that belongs to L
will get accepted by M , and b) an input string that
does not belong to L does not get accepted by M .

Remarks

A finite state machine can be a good tool to model simple applications such as the vending machine.

We now know how to accept a language with finite state machines. However, for this application, we can simplify matters a little bit (\Rightarrow finite state automata).

We need a more descriptive way to describe the languages that are accepted by a finite state machine (\Rightarrow regular expressions).

Finite State Machines with No Output

Motivation

Suppose that we want to use a finite state machine simply for the purpose of language recognition. Recall that all output bits were ignored with the exception of the last output bit. The value of the last bit decided whether or not the input read belongs to the language that is accepted by the FSM.

Last output bit is 1 if and only if the input string is accepted (i.e., belongs to the language).

We might as well do away with any output and decide whether or not an input string belongs to the language depending on the value of the last state. We will have accepting (and rejecting) states.

Finite State Automata

A **finite state automaton** $M=(S, I, f, s_0, F)$ consists of a finite set S of states, a finite input alphabet I , a transition function $f: S \times I \rightarrow S$ that assigns to a given current state and input the next state of the automaton, an initial state s_0 , and a subset F of S consisting of accepting (or final) states.

Finite State Machines vs Automata

A finite state machine M is given by $M=(S, I, O, f, g, s_0)$,
where:

S is the set of states

I is input alphabet

O is output alphabet

f is the transition function that assigns each (state, input) pair a new state

g is output function that assigns each (state, input) pair an output.

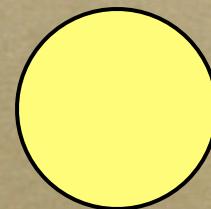
s_0 is the initial state

*Missing in case of finite
state automata*

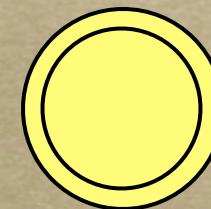
Notation

In state diagrams, the accepting states are denoted by double circles.

rejecting state

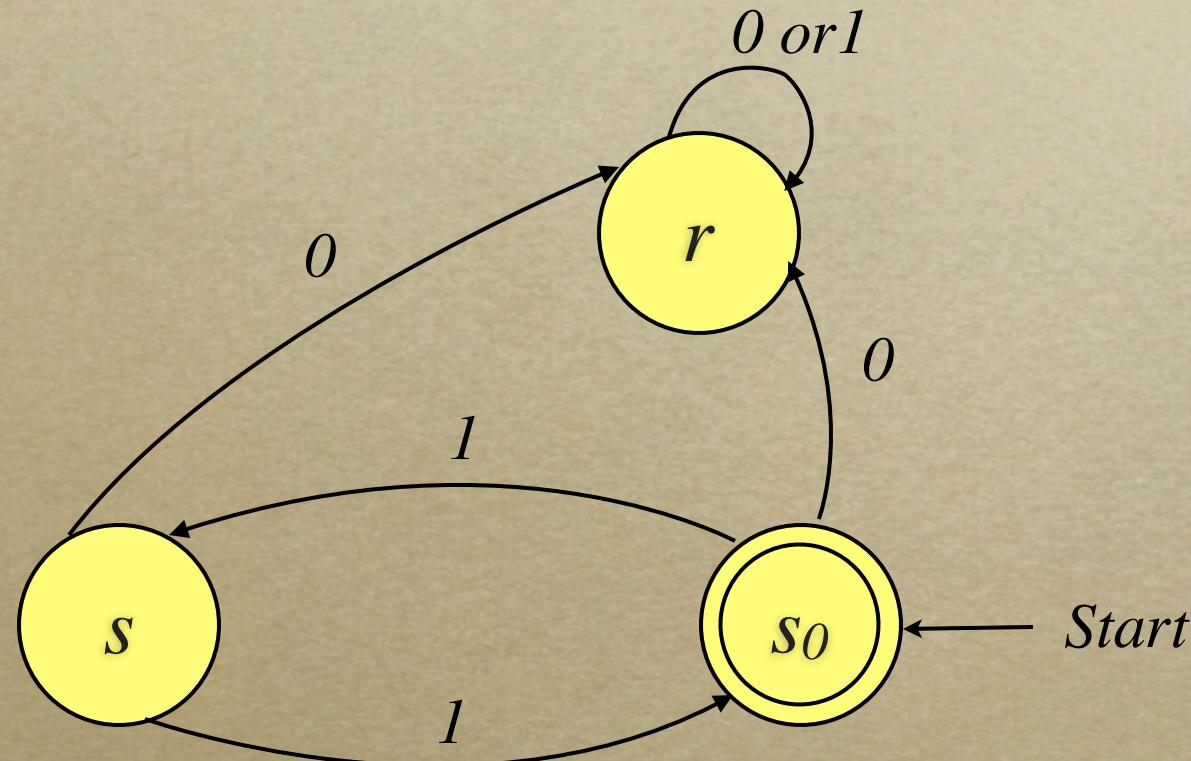


accepting state



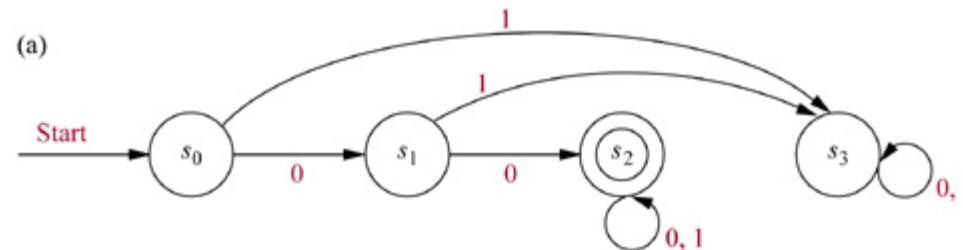
Example 1

What language does this automaton accept?



Example 2

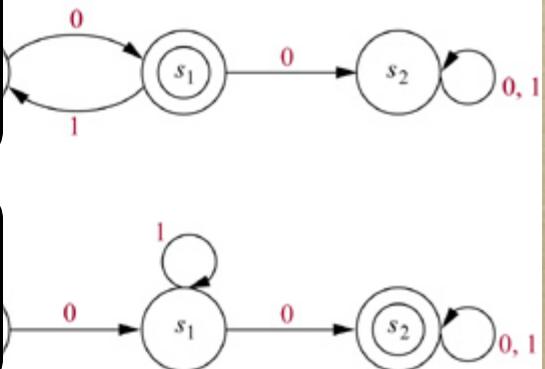
© The McGraw-Hill Companies, Inc. all rights reserved.



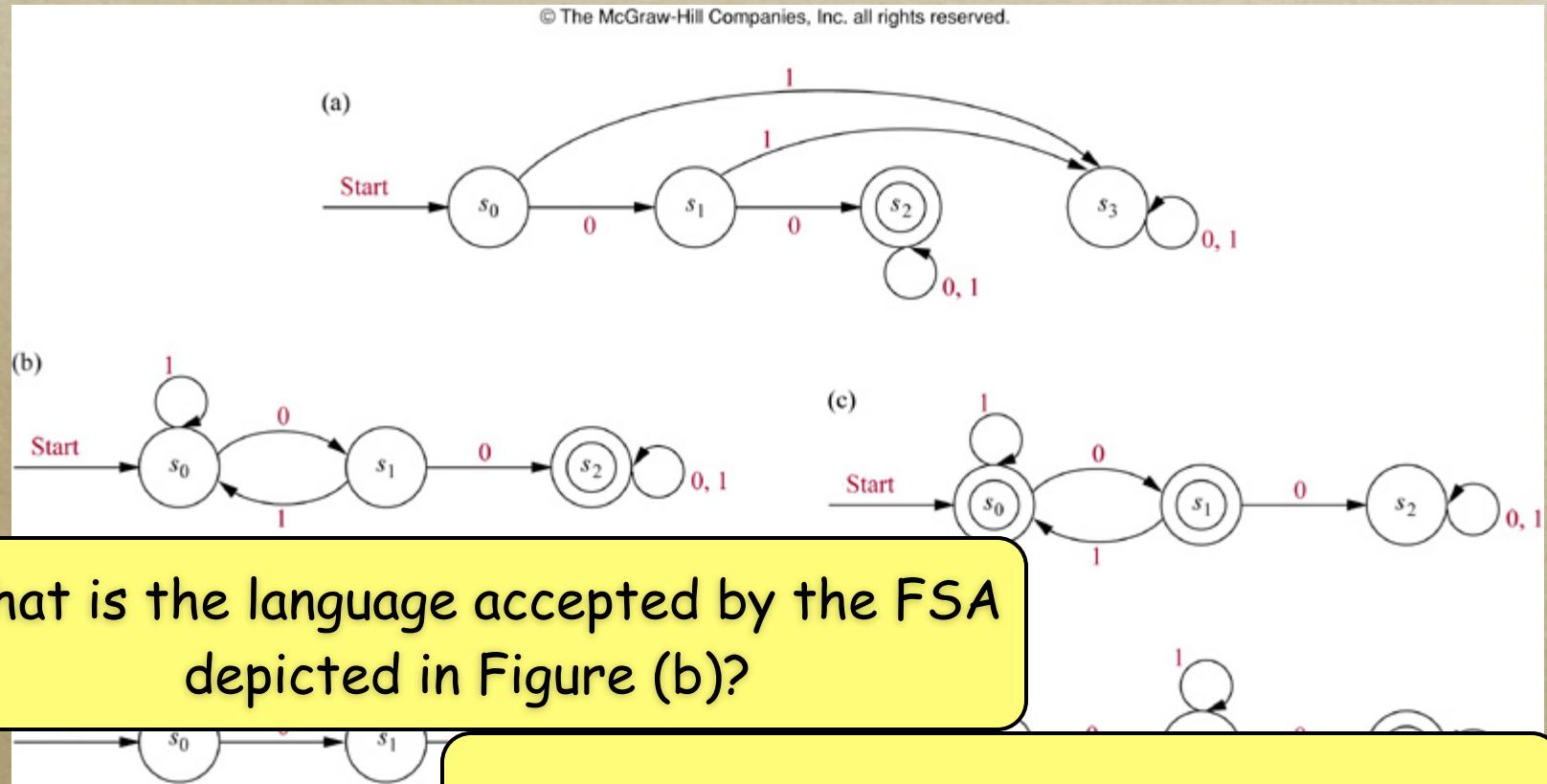
(b)

What is the language accepted by the FSA depicted in Figure (a)?

The set of bit strings that begin with 00



Example 3

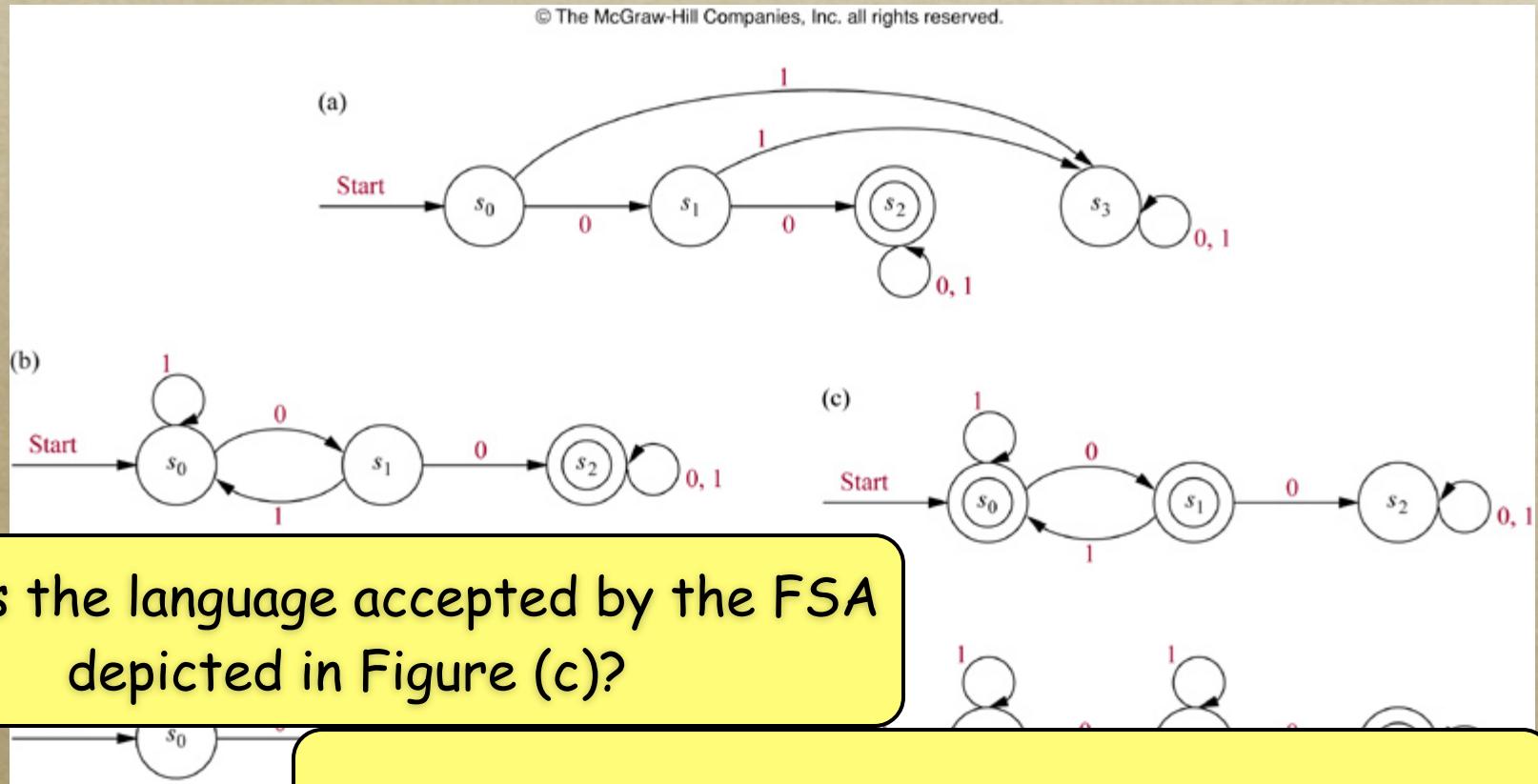


What is the language accepted by the FSA depicted in Figure (b)?



The set of bit strings that contain 00

Example 4



What is the language accepted by the FSA depicted in Figure (c)?

The set of bit strings that do not contain 00

Fundamental Theorem

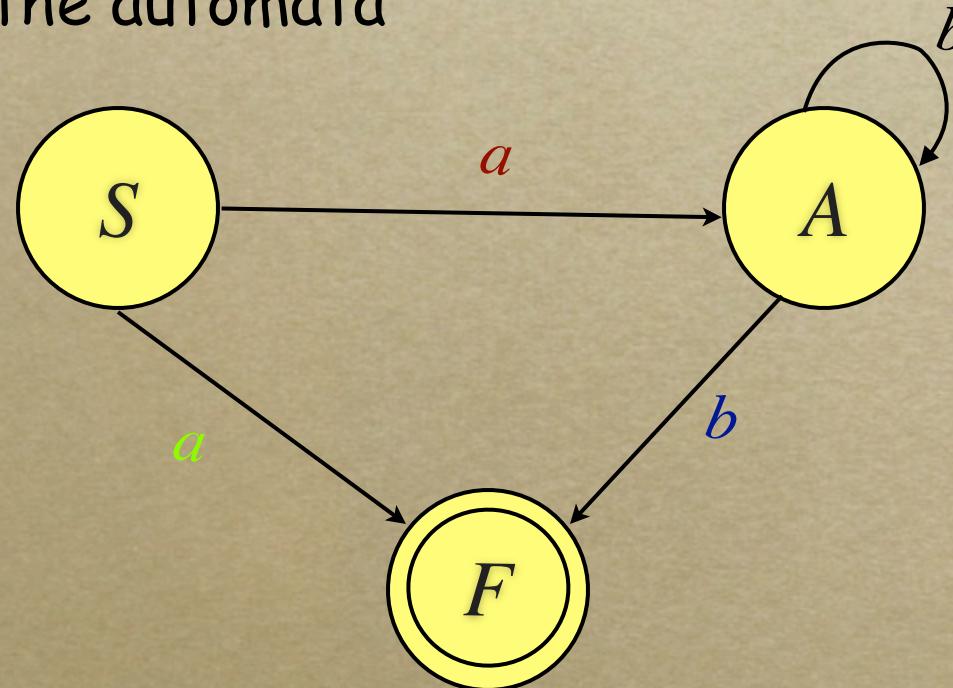
Theorem: A formal language L is regular if and only if there exists a finite state automata M accepting L .

Proof: Given in CSCE 433, but we will at least illustrate the main idea with the help of an example.

Proof Idea

Suppose that a regular grammar has the production rules: $P = \{S \rightarrow aA, S \rightarrow a, A \rightarrow bA, A \rightarrow b\}$.

Define the automata



A node for each non-terminal symbol, and an additional node for an accepting state F .

Each production $A \rightarrow aB$ yields an edge labeled with a from A to B .

A production $A \rightarrow a$ yields an edge from A to F labeled by a .

Regular Expressions

We will now introduce an algebraic description of formal languages with the help of regular expressions.

This will lead to yet another characterization of regular languages.

Operations on Languages: Concatenation

Suppose that V is an alphabet.

Let A and B be subsets of V^* .

Denote by AB the set $\{ xy \mid x \text{ in } A, y \text{ in } B\}$.

Example: $A = \{0,11\}$ and $B = \{1, 10, 110\}$

Then $AB = \{ 01, 010, 0110, 111, 1110, 11110 \}$

Operations on Languages: Exponent Notation

Suppose that V is an alphabet.

Let A be a subset of V^* .

Define $A^0 = \{ \lambda \}$ and $A^{n+1} = A^n A$

Example: $A = \{1, 00\}$

$A^2 = \{ 11, 100, 001, 0000 \}$

Operations on Languages: Kleene Closure

Let V be an alphabet, and A a subset of V^* .

The Kleene closure A^* of A is defined as

$$A^* = \bigcup_{k=0}^{\infty} A^k$$

Operations on Languages: Union

The union of two formal languages L_1 and L_2 is the formal language $\{ x \mid x \text{ in } L_1 \text{ or } x \text{ in } L_2 \}$.

Closure Properties

Theorem: The class of regular languages is closed under the operations: concatenation, Kleene closure, union, intersection, complement.

Corollary: All finite languages are regular.

Corollary: The complement of a finite language is regular.

Grammar of Regular Expressions

Let V be a finite alphabet.

Terminals: $T = V \cup \{ \emptyset, \lambda, \cup, *, (,) \}$

Nonterminals: $N = \{S\}$ with start symbol S

$P = \{ S \rightarrow b \text{ for all } b \text{ in } V \cup \{ \emptyset, \lambda \}, S \rightarrow (S \cup S), S \rightarrow (SS), S \rightarrow S^* \}$

The grammar $G = (N \cup T, T, S, P)$ is called the **grammar of regular expressions**. The elements in $L(G)$ are called **regular expressions** over the alphabet V .

Regular Expressions

The semantic of the language $L(G)$ of regular expressions is given by associating a formal language with each regular expression as follows:

$$E(\emptyset) = \emptyset,$$

$$E(\lambda) = \{\lambda\}$$

$$E(a) = \{a\} \text{ for all } a \text{ in } V$$

$$E((X \cup Y)) = E(X) \cup E(Y)$$

$$E((XY)) = E(X) E(Y)$$

$$E(X^*) = E(X)^*$$

Fundamental Theorem

Theorem (Kleene-Myhill): The class of regular languages coincides with the languages that can be described by regular expressions.

Example

$$E(a^*b^*) = \{ a^k b^l \mid k \geq 0, l \geq 0 \}$$

A Nonregular Language

Theorem: The language $L = \{ 0^n 1^n \mid n \geq 0 \}$ is not regular.

Proof: Seeking a contradiction, suppose that M is a finite state automaton accepting L . Let m denote the number of states of M . On input of $0^m 1^m$, the finite state automaton makes $2m$ transitions, say

$$s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_{2m}.$$

until ending up in the accepting state s_{2m} . Since there are just m different states, by the pigeonhole principle at least two of the states s_0, s_1, \dots, s_m must be the same.

A Nonregular Language

Therefore, there is a loop from one of these state back to itself of, say, length k. Since all of these transitions occur in input of 0, this means that the automaton also accepts the input string $0^{m+k}1^m$. Since this string does not belong to the language L, we get a contradiction. Therefore, we can conclude that there does not exist an automaton accepting the language $L = \{ 0^n1^n \mid n \geq 0 \}$.