I. Lecture Notes

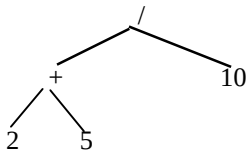
The Three Hour Tour Through Automata Theory

Read Supplementary Materials: The Three Hour Tour Through Automata Theory
Read Supplementary Materials: Review of Mathematical Concepts
Read K & S Chapter 1
Do Homework 1.

Let's Look at Some Problems

```
int alpha, beta;
alpha = 3;
beta = (2 + 5) / 10;
```

(1) **Lexical analysis**: Scan the program and break it up into variable names, numbers, etc.
(2) **Parsing**: Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



(3) **Optimization**: Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.
(4) **Termination**: Decide whether the program is guaranteed to halt.
(5) **Interpretation**: Figure out what (if anything) it does.

A Framework for Analyzing Problems

We need a single framework in which we can analyze a very diverse set of problems.
The framework we will use is **Language Recognition**

A ***language*** is a (possibly infinite) set of finite length strings over a finite alphabet.

Languages

(1) $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
L = $\{w \in \Sigma^*$: w represents an odd integer$\}$
= $\{w \in \Sigma^*$: the last character of w is 1,3,5,7, or 9$\}$
= $(0 \cup 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9)^*$
$(1 \cup 3 \cup 5 \cup 7 \cup 9)$

(2) $\Sigma = \{(,)\}$
L = $\{w \in \Sigma^*$: w has matched parentheses$\}$
= the set of strings accepted by the grammar:
$S \rightarrow (S)$
$S \rightarrow SS$
$S \rightarrow \varepsilon$

(3) L = {w: w is a sentence in English}
Examples: Mary hit the ball.
Colorless green ideas sleep furiously.
The window needs fixed.

(4) L = {w: w is a C program that halts on all inputs}

Encoding Output in the Input String

(5) Encoding multiplication as a single input string

L = {w of the form: <integer>x<integer>=<integer>, where <integer> is any well formed integer, and the third integer is the product of the first two}

12x9=108 12=12 12x8=108

(6) Encoding prime decomposition

L = {w of the form: <integer1>/<integer2>,<integer3> …, where integers 2 - n represent the prime decomposition of integer 1.

15/3,5 2/2

More Languages

(7) Sorting as a language recognition task:

L = {w_1 # w_2: $\exists n \geq 1$,

w_1 is of the form int_1, int_2, … int_n,

w_2 is of the form int_1, int_2, … int_n, and

w_2 contains the same objects as w_1 and w_2 is sorted}

Examples:

1,5,3,9,6#1,3,5,6,9 $\in$ L

1,5,3,9,6#1,2,3,4,5,6,7 $\notin$ L

(8) Database querying as a language recognition task:

L = {d # q # a:

d is an encoding of a database,

q is a string representing a query, and

a is the correct result of applying q to d}

Example:

(name, age, phone), (John, 23, 567-1234) (Mary, 24, 234-9876)# (select name age=23) # (John) $\in$ L

The Traditional Problems and their Language Formulations are Equivalent

By equivalent we mean:

If we have a machine to solve one, we can use it to build a machine to do the other using just the starting machine and other functions that can be built using a machine of equal or lesser power.

Consider the multiplication example:

L = {w of the form:

<integer>x<integer>=<integer>, where

<integer> is any well formed integer, and

the third integer is the product of the first two}

Given a multiplication machine, we can build the language recognition machine:

Given the language recognition machine, we can build a multiplication machine:

A Framework for Describing Languages

Clearly, if we are going to work with languages, each one must have a finite description.

Finite Languages: Easy. Just list the elements of the language.
L = {June, July, August}

Infinite Languages: Need a finite description.

Grammars let us use recursion to do this.

Grammars 1

(1) The Language of Matched Parentheses

$S \rightarrow (S)$
$S \rightarrow SS$
$S \rightarrow \varepsilon$

(2) The Language of Odd Integers

$S \rightarrow 1$
$S \rightarrow 3$
$S \rightarrow 5$
$S \rightarrow 7$
$S \rightarrow 9$
$S \rightarrow 0\ S$
$S \rightarrow 1\ S$
$S \rightarrow 2\ S$
$S \rightarrow 3\ S$
$S \rightarrow 4\ S$
$S \rightarrow 5\ S$
$S \rightarrow 6\ S$
$S \rightarrow 7\ S$
$S \rightarrow 8\ S$
$S \rightarrow 9\ S$

Grammars 2

$S \rightarrow O$
$S \rightarrow A\ O$
$A \rightarrow A\ D$
$A \rightarrow D$
$D \rightarrow O$
$D \rightarrow E$
$O \rightarrow 1$
$O \rightarrow 3$
$O \rightarrow 5$
$O \rightarrow 7$
$O \rightarrow 9$
$E \rightarrow 0$
$E \rightarrow 2$
$E \rightarrow 4$
$E \rightarrow 6$
$E \rightarrow 8$

Grammars 3

(3) The Language of Simple Arithmetic Expressions

<exp> → <number>
<exp> → (<exp>)
<exp> → - <exp>
<exp> → <exp> <op> <exp>
<op> → + | - | * | /
<number> → <digit>
<number> → <digit> <number>
<digit > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Grammars as Generators and Acceptors

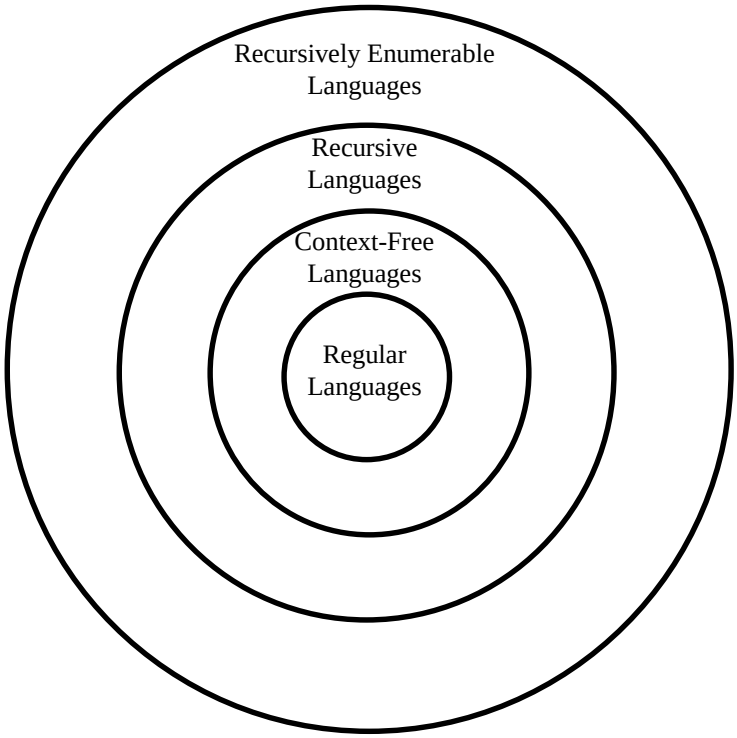
Top Down Parsing

4 + 3

Bottom Up Parsing

4 + 3

The Language Hierarchy



Regular Grammars

In a regular grammar, all rules must be of the form:

<one nonterminal> → <one terminal> or ε

or

<one nonterminal> → <one terminal><one nonterminal>

So, the following rules are okay:

$S \rightarrow \varepsilon$
$S \rightarrow a$
$S \rightarrow aS$

But these are not:

$S \rightarrow ab$
$S \rightarrow SS$
$aS \rightarrow b$

Regular Expressions and Languages

Regular expressions are formed from ∅ and the characters in the target alphabet, plus the operations of:

- Concatenation: $\alpha\beta$ means α followed by β
- Or (Set Union): $\alpha \cup \beta$ means α Or (Union) β
- Kleene *: α^* means 0 or more occurrences of α concatenated together.
- At Least 1: α^+ means 1 or more occurrences of α concatenated together.
- (): used to group the other operators

Examples:

(1) Odd integers:
(0∪1∪2∪3∪4∪5∪6∪7∪8∪9)*(1∪3∪5∪7∪9)

(2) Identifiers:
$(A\text{-}Z)^+((A\text{-}Z) \cup (0\text{-}9))^*$

(3) Matched Parentheses

Context Free Grammars

(1) The Language of Matched Parentheses

$S \rightarrow (S)$
$S \rightarrow SS$
$S \rightarrow \varepsilon$

(2) The Language of Simple Arithmetic Expressions

<S> → <exp>
<exp> → <number>
<exp> → (<exp>)
<exp> → - <exp>
<exp> → <exp> <op> <exp>
<op> → + | - | * | /
<number> → <digit>
<number> → <digit> <number>
<digit > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Not All Languages are Context-Free

English:
S → NP VP
NP → the NP1 | NP1
NP1 → ADJ NP1 | N
N → boy | boys
VP →V | V NP
V → run | runs
What about "boys runs"

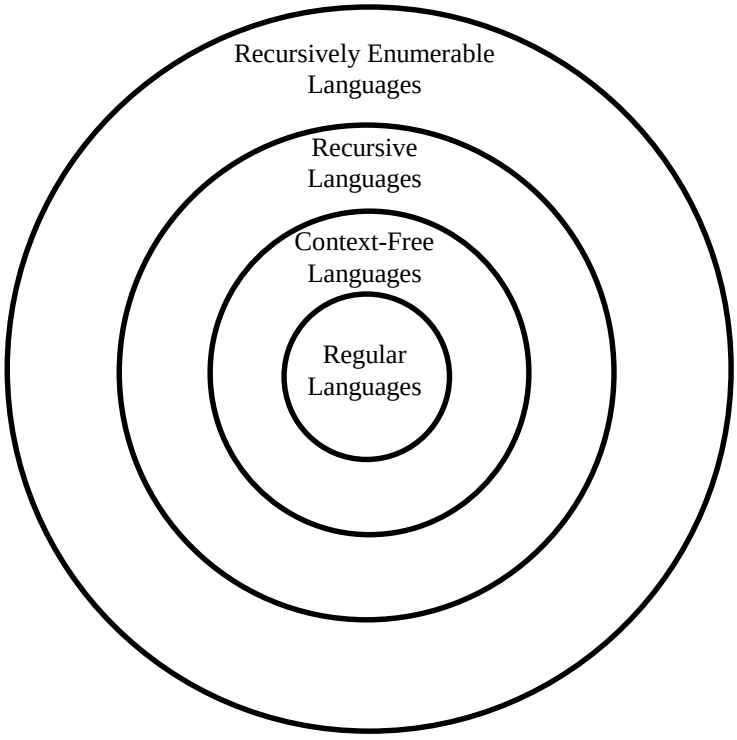
A much simpler example: $a^n b^n c^n$, $n \geq 1$

Unrestricted Grammars

Example: A grammar to generate all strings of the form $a^n b^n c^n$, $n \geq 1$

S → aBSc
S → aBc
Ba → aB
Bc → bc
Bb → bb

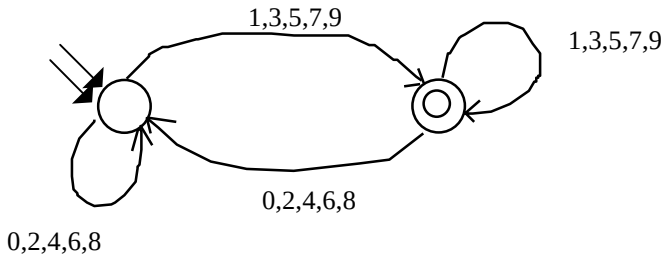
The Language Hierarchy



A Machine Hierarchy

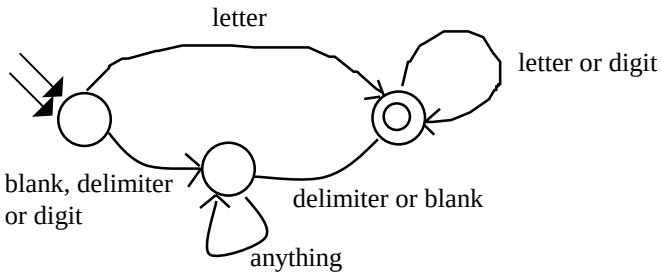
Finite State Machines 1

An FSM to accept odd integers:



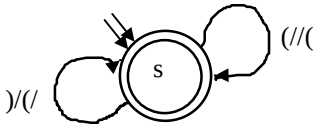
Finite State Machines 2

An FSM to accept identifiers:



Pushdown Automata

A PDA to accept strings with balanced parentheses:

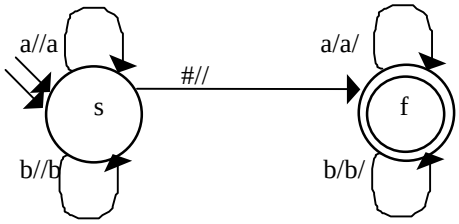


Example: (())()

Stack:

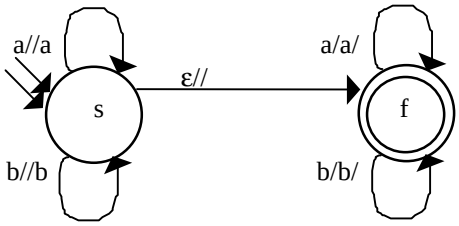
Pushdown Automaton 2

A PDA to accept strings of the form $w\#w^R$:



A Nondeterministic PDA

A PDA to accept strings of the form ww^R

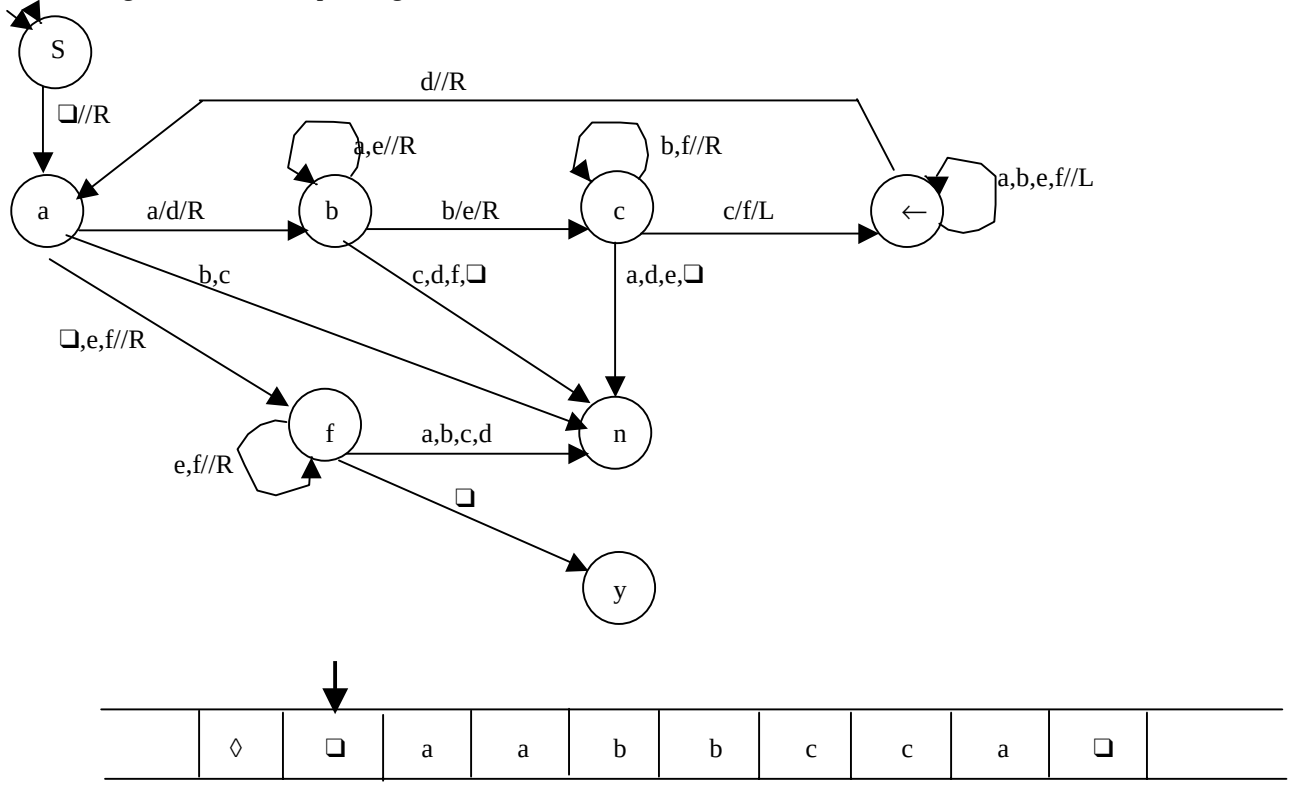


PDA 3

A PDA to accept strings of the form $a^n b^n c^n$

Turing Machines

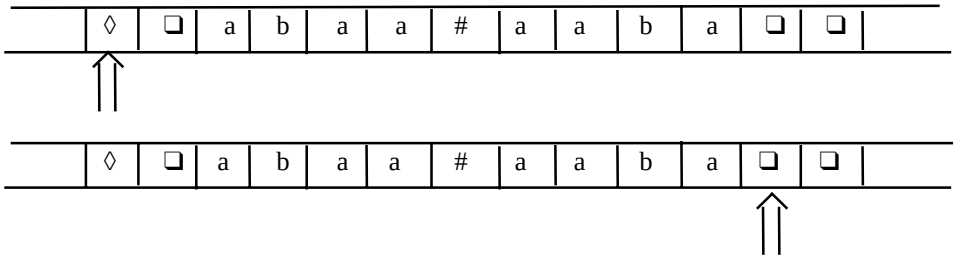
A Turing Machine to accept strings of the form $a^n b^n c^n$



A Two Tape Turing Machine

A Turing Machine to accept $\{w\#w^R\}$

A Two Tape Turing Machine to do the same thing



Simulating k Tapes with One

A multitrack tape:

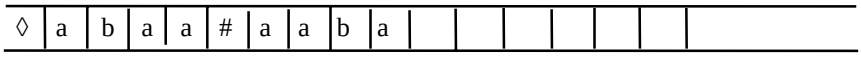
◊	◊	❑	a	b	a	❑	❑	❑	❑
	0	0	1	0	0	0	0	
	◊	a	b	b	a	b	a	
	0	1	0	0	0	0	0	

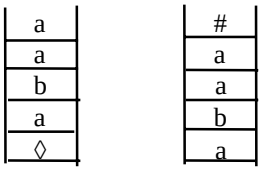
Can be encoded on a single tape with an alphabet consisting of symbols corresponding to :

$\{\{◊,a,b,\#,❑\} \times \{0,1\} \times \{◊,a,b,\#,❑\} \times \{0,1\}\}$

Example: 2nd square: (❑,0,a,1))

Simulating a Turing Machine with a PDA with Two Stacks





The Universal Turing Machine
Encoding States, Symbols, and Transitions

Suppose the input machine M has 5 states, 4 tape symbols, and a transition of the form:

(s,a,q,b), which can be read as:

in state s, reading an a, go to state q, and write b.

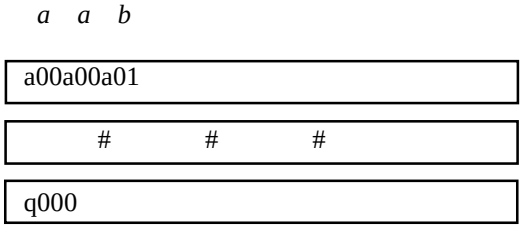
We encode this transition as:

q000,a00,q010,a01

A series of transitions that describe an entire machine will look like

q000,a00,q010,a01#q010,a00,q000,a00

The Universal Turing Machine



Church's Thesis
(Church-Turing Thesis)

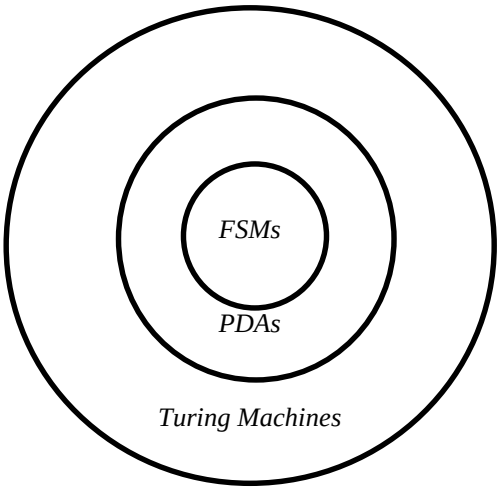
An algorithm is a formal procedure that halts.

The Thesis: Anything that can be computed by any algorithm can be computed by a Turing machine.

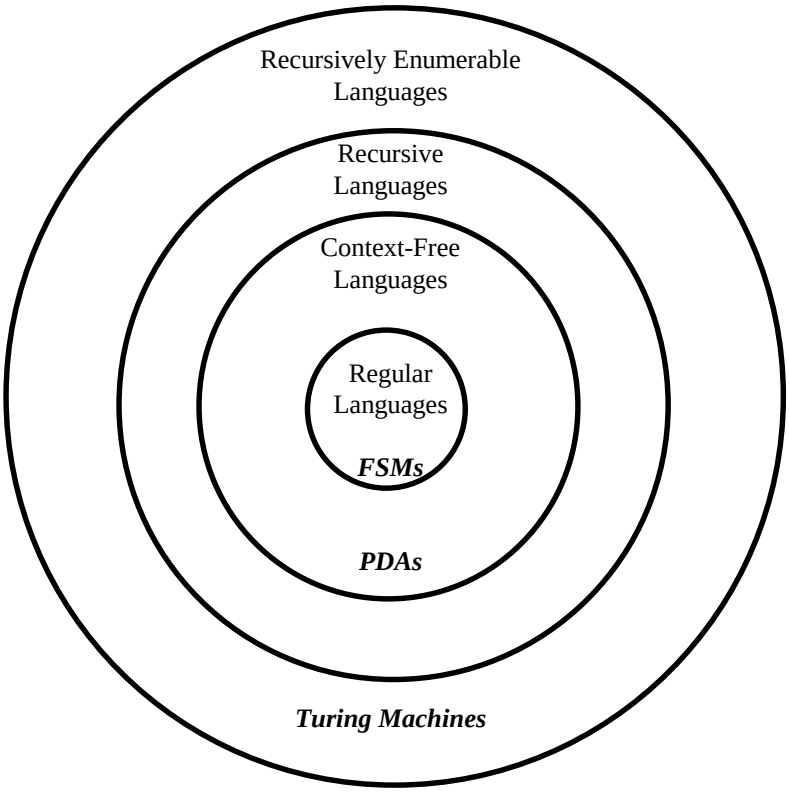
Another way to state it: All "reasonable" formal models of computation are equivalent to the Turing machine. This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

Example: unrestricted grammars

A Machine Hierarchy



Languages and Machines



Where Does a Particular Problem Go?

Showing what it is -- generally by construction of:

- A grammar, or a machine

Showing what it isn't -- generally by contradiction, using:

- Counting
  Example: $a^n b^n$
- Closure properties
- Diagonalization
- Reduction

Closure Properties

Regular Lanugages are Closed Under:

- Union
- Concatenation
- Kleene closure
- Complementation
- Reversal
- Intersection

Context Free Languages are Closed Under:

- Union
- Concatenation
- Kleene Closure
- Reversal
- Intersection with regular languages

Etc.

Using Closure Properties

Example:
$L = \{a^n b^m c^p : n \neq m \text{ or } m \neq p\}$ is not deterministic context-free.

Two theorems we'll prove later:

Theorem 3.7.1: The class of deterministic context-free languages is closed under complement.

Theorem 3.5.2: The intersection of a context-free language with a regular language is a context-free language.

If L were a deterministic CFL, then the complement of L (L') would be a deterministic CFL.

But $L' \cap a^*b^*c^* = \{a^n b^n c^n\}$, which we know is not context-free, much less deterministic context-free. Thus a contradiction.

Diagonalization

The power set of the integers is not countable.
Imagine that there were some enumeration:

	1	2	3	4	5
Set 1	1				
Set 2		1		1	
Set 3	1		1		
Set 4		1			
Set 5	1	1	1	1	1

But then we could create a new set

New Set				1	

But this new set must necessarily be different from all the other sets in the supposedly complete enumeration. Yet it should be included. Thus a contradiction.

More on Cantor

Of course, if we're going to enumerate, we probably want to do it very systematically, e.g.,

	1	2	3	4	5	6	7
Set 1	1						
Set 2		1					
Set 3	1	1					
Set 4			1				
Set 5	1		1				
Set 6		1	1				
Set 7	1	1	1				

Read the rows as bit vectors, but read them backwards. So Set 4 is 100. Notice that this is the binary encoding of 4.
This enumeration will generate all **finite** sets of integers, and in fact the set of all finite sets of integers is countable.
But when will it generate the set that contains all the integers except 1?

The Unsolvability of the Halting Problem

Suppose we could implement

HALTS(M,x)

M: string representing a Turing Machine
x: string representing the input for M
If M(x) halts then True
else False

Then we could define

TROUBLE(x)

x: string
If HALTS(x,x) then loop forever
else halt

So now what happens if we invoke TROUBLE(TROUBLE), which invokes HALTS(TROUBLE,TROUBLE)

If HALTS says that TROUBLE halts on itself then TROUBLE loops. IF HALTS says that TROUBLE loops, then TROUBLE halts.

Viewing the Halting Problem as Diagonalization

First we need an enumeration of the set of all Turing Machines. We'll just use lexicographic order of the encodings we used as inputs to the Universal Turing Machine. So now, what we claim is that HALTS can compute the following table, where 1 means the machine halts on the input:

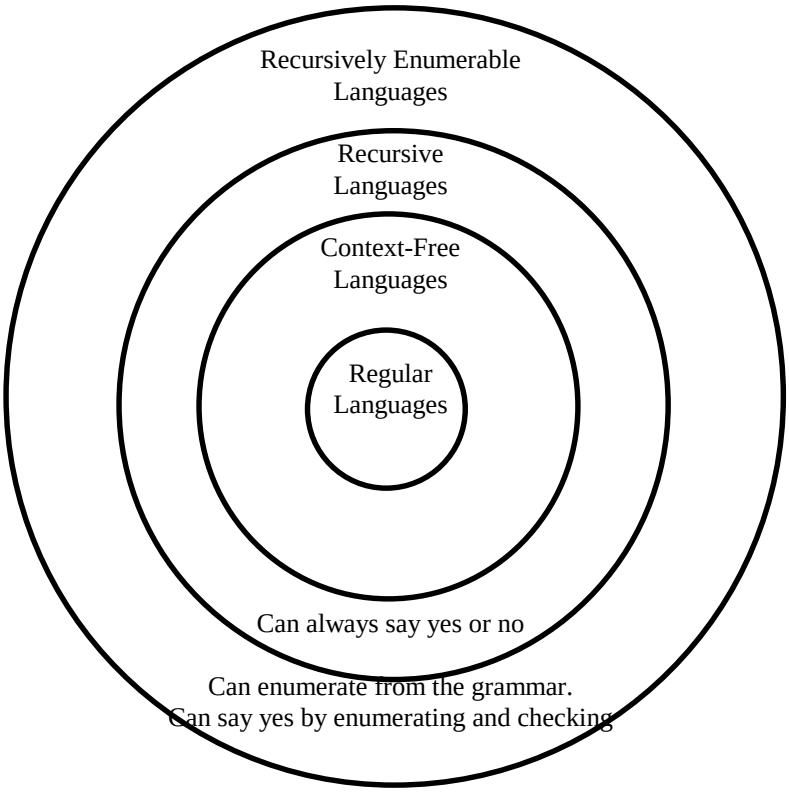
	I1	I2	I3	TROUBLE	I5
Machine 1	1				
Machine 2		1		1	
Machine 3					
TROUBLE			1		1
Machine 5	1	1	1	1	

But we've defined TROUBLE so that it will actually behave as:

TROUBLE			1	1	1

Or maybe HALT said that TROUBLE(TROUBLE) would halt. But then TROUBLE would loop.

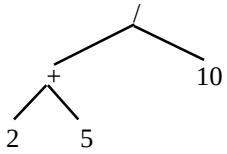
Decidability



Let's Revisit Some Problems

```
int alpha, beta;
alpha = 3;
beta = (2 + 5) / 10;
```

(1) **Lexical analysis**: Scan the program and break it up into variable names, numbers, etc.
(2) **Parsing**: Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



(3) **Optimization**: Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.
(4) **Termination**: Decide whether the program is guaranteed to halt.
(5) **Interpretation**: Figure out what (if anything) useful it does.

So What's Left?

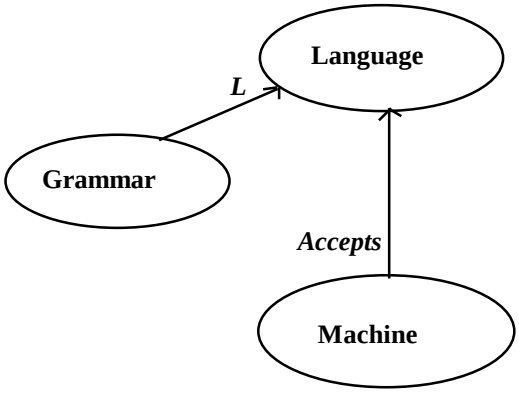
- Formalize and Prove Things

- Regular Languages and Finite State Machines
    - FSMs
        - Nondeterminism
        - State minimization
        - Implementation
    - Equivalence of regular expressions and FSMs
    - Properties of Regular Languages
- Context-Free Languages and PDAs
    - Equivalence of CFGs and nondeterministic PDAs
    - Properties of context-free languages
    - Parsing and determinism
- Turing Machines and Computability
    - Recursive and recursively enumerable languages
    - Extensions of Turing Machines
    - Undecidable problems for Turing Machines and unrestricted grammars

What Is a Language?

Do Homework 2.

Grammars, Languages, and Machines



Strings: the Building Blocks of Languages

An **alphabet** is a finite set of **symbols**: English alphabet: {A, B, C, …, Z}
Binary alphabet: {0, 1}

A **string** over an alphabet is a finite sequence of symbols drawn from the alphabet.

English string: happynewyear
binary string: 1001101

We will generally omit " " from strings unless doing so would lead to confusion.

The set of all possible strings over an alphabet Σ is written Σ*.
binary string: 1001101 ∈ {0,1}*

The shortest string contains no characters. It is called the **empty string** and is written " " or ε (epsilon).

The set of all possible strings over an alphabet Σ is written Σ*.

More on Strings

The **length** of a string is the number of symbols in it.
|ε| = 0
|1001101| = 7

A string a is a **substring** of a string b if a occurs contiguously as part of b.
aaa is a substring of aaabbbaaa
aaaaaa is not a substring of aaabbbaaa

Every string is a substring (although not a proper substring) of itself.

ε is a substring of every string. Alternatively, we can match ε anywhere.

Notice the analogy with sets here.

Operations on Strings

Concatenation: The **concatenation** of two strings *x* and *y* is written $x \| y$, $x \cdot y$, or xy and is the string formed by appending the string *y* to the string *x*.

$|xy| = |x| + |y|$

If $x = \varepsilon$ and $y =$ "food", then $xy =$
If $x =$ "good" and $y =$ "bye", then $|xy| =$

Note: $x \cdot \varepsilon = \varepsilon \cdot x = x$ for all strings x.

Replication: For each string *w* and each natural number *i*, the string w^i is defined recursively as

$w^0 = \varepsilon$
$w^i = w^{i-1} w$ for each $i \geq 1$

Like exponentiation, the replication operator has a high precedence.

Examples:
$a^3 =$
$(bye)^2 =$
$a^0 b^3 =$

String Reversal

An inductive definition:

(1) If $|w| = 0$ then $w^R = w = \varepsilon$
(2) If $|w| \geq 1$ then $\exists\, a \in \Sigma$: $w = u \cdot a$
(a is the last character of w)
and
$w^R = a \cdot u^R$

Example:
$(abc)^R =$

More on String Reversal

Theorem: If w, x are strings, then $(w \cdot x)^R = x^R \cdot w^R$

Example: $(dogcat)^R = (cat)^R \cdot (dog)^R =$ tacgod

Proof (by induction on |x|):

Basis: $|x| = 0$. Then $x = \varepsilon$, and $(w \cdot x)^R = (w \cdot \varepsilon)^R = (w)^R = \varepsilon \cdot w^R = \varepsilon^R \cdot w^R = x^R \cdot w^R$

Induction Hypothesis: If $|x| \leq n$, then $(w \cdot x)^R = x^R \cdot w^R$

Induction Step: Let $|x| = n + 1$. Then $x = u\,a$ for some character a and $|u| = n$

$(w \cdot x)^R = (w \cdot (u \cdot a))^R$	
$= ((w \cdot u) \cdot a)^R$	associativity
$= a \cdot (w \cdot u)^R$	definition of reversal
$= a \cdot u^R \cdot w^R$	induction hypothesis
$= (u \cdot a)^R \cdot w^R$	definition of reversal
$= x^R \cdot w^R$	

$\overline{d\ o\ g}^{\,w}\ \overline{c\ a\ t}^{\,x}$

$\overline{u}\ \overline{a}$

Defining a Language

A **language** is a (finite or infinite) set of finite length strings over a finite alphabet Σ.

Example: Let Σ = {a, b}

Some languages over Σ: ∅, {ε}, {a, b}, {ε, a, aa, aaa, aaaa, aaaaa}

The language Σ* contains an infinite number of strings, including: ε, a, b, ab, ababaaa

Example Language Definitions

L = {x ∈ {a, b}* : all a's precede all b's}

L = {x : ∃y ∈ {a, b}* : x = ya}

L = {a^n, n ≥ 0 }

L = a^n (If we say nothing about the range of n, we will assume that it is drawn from N, i.e., n ≥ 0.)

L = {x#y: x,y ∈ {0-9}* and square(x) = y}

L = {} = ∅ (the empty language—not to be confused with {ε}, the language of the empty string)

Techniques for Defining Languages

Languages are sets. Recall that, for sets, it makes sense to talk about ***enumerations*** and ***decision procedures***. So, if we want to provide a computationally effective definition of a language we could specify either a

- Language **generator**, which enumerates (lists) the elements of the language, or a
- Language **recognizer**, which decides whether or not a candidate string is in the language and returns True if it is and False if it isn't.

Example: The logical definition: L = {x : ∃y ∈ {a, b}* : x = ya} can be turned into either a language generator or a language recognizer.

How Large are Languages?

- The smallest language over any alphabet is ∅. |∅| = 0
- The largest language over any alphabet is Σ*. |Σ*| = ?
  - If Σ = ∅ then Σ* = {ε} and |Σ*| = 1
  - If Σ ≠ ∅ then |Σ*| is countably infinite because its elements can be enumerated in 1 to 1 correspondence with the integers as follows:
    1. Enumerate all strings of length 0, then length 1, then length 2, and so forth.
    2. Within the strings of a given length, enumerate them lexicographically. E.g., aa, ab, ba, bb

- So all languages are either finite or countably infinite. Alternatively, all languages are *countable*.

Operations on Languages 1

Normal set operations: **union**, **intersection**, **difference**, **complement**…

Examples: Σ = {a, b} L_1 = strings with an even number of a's
L_2 = strings with no b's

$L_1 \cup L_2$ =
$L_1 \cap L_2$ =
$L_2 - L_1$ =
$\neg(L_2 - L_1)$ =

Operations on Languages 2

Concatenation: (based on the definition of concatenation of strings)

If L_1 and L_2 are languages over Σ, their concatenation $L = L_1 L_2$, sometimes $L_1 \cdot L_2$, is
$\{w \in \Sigma^*: w = x\ y \text{ for some } x \in L_1 \text{ and } y \in L_2\}$

Examples:
$L_1 = \{cat, dog\}$ $L_2 = \{apple, pear\}$ $L_1 L_2 = \{catapple, catpear, dogapple, dogpear\}$
$L_1 = \{a^n: n \geq 1\}$ $L_2 = \{a^n: n \leq 3\}$ $L_1 L_2 =$

Identities:
$L\varnothing = \varnothing L = \varnothing \quad \forall L$ (analogous to multiplication by 0)
$L\{\varepsilon\} = \{\varepsilon\}L = L \quad \forall L$ (analogous to multiplication by 1)

Replicated concatenation:
$L^n = L \cdot L \cdot L \cdot \ \ldots \cdot L$ (n times)
$L^1 = L$
$L^0 = \{\varepsilon\}$

Example:
L = {dog, cat, fish}
$L^0 = \{\varepsilon\}$
L^1 = {dog, cat, fish}
L^2 = {dogdog, dogcat, dogfish, catdog, catcat, catfish, fishdog, fishcat, fishfish}

Concatenating Languages Defined Using Variables

$L_1 = a^n = \{a^n : n \geq 0\}$ $L_2 = b^n = \{b^n : n \geq 0\}$
$L_1 L_2 = \{a^n : n \geq 0\}\{b^n : n \geq 0\} = \{ a^n b^m : n,m \geq 0\}$ (common mistake:) $\neq a^n b^n = \{ a^n b^n : n \geq 0\}$

Note: The scope of any variable used in an expression that invokes replication will be taken to be the entire expression.

$L = 1^n 2^m$
$L = a^n b^m a^n$

Operations on Languages 3

Kleene Star (or Kleene closure): $\mathbf{L^*} = \{w \in \Sigma^* : w = w_1\ w_2 \ldots w_k \text{ for some } k \geq 0 \text{ and some } w_1, w_2, \ldots w_k \in L\}$

Alternative definition: $\mathbf{L^*} = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

Note: $\forall L, \varepsilon \in L^*$

Example:
L = {dog, cat, fish}
L* = {ε, dog, cat, fish, dogdog, dogcat, fishcatfish, fishdogdogfishcat, …}

Another useful definition: $\mathbf{L^+} = L\ L^*$ (L^+ is the **closure** of L under concatenation)

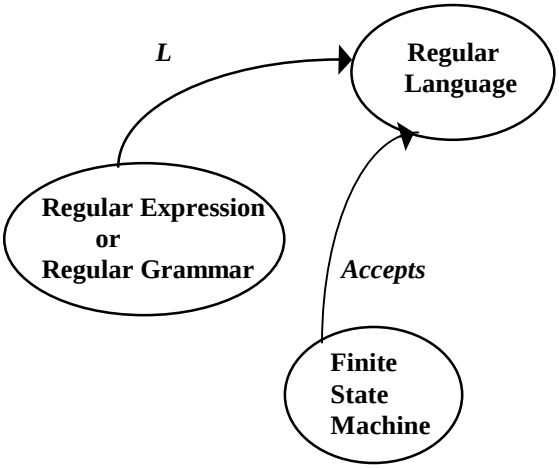
Alternatively, $\mathbf{L^+} = L^1 \cup L^2 \cup L^3 \cup \ldots$

$L^+ = L^* - \{\varepsilon\}$ if $\varepsilon \notin L$
$L^+ = L^*$ if $\varepsilon \in L$

Regular Languages

Read Supplementary Materials: Regular Languages and Finite State Machines: Regular Languages
Do Homework 3.

Regular Grammars, Languages, and Machines



"Pure" Regular Expressions

The **regular expressions** over an alphabet Σ are all strings over the alphabet $\Sigma \cup \{\text{“(“, “)”}, \varnothing, \cup, *\}$ that can be obtained as follows:

1. ∅ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is αβ
3. If α , β are regular expressions, then so is α∪β.
4. If α is a regular expression, then so is α*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

If Σ = {a,b} then these are regular expressions: ∅, a, bab, a∪b , (a∪b)*a*b*

So far, regular expressions are just (finite) strings over some alphabet, $\Sigma \cup \{\text{“(“, “)”}, \varnothing, \cup, *\}$.

Regular Expressions Define Languages

Regular expressions define languages via a **semantic interpretation function** we'll call L:
1. $L(\varnothing) = \varnothing$ and $L(a) = \{a\}$ for each $a \in \Sigma$
2. If α , β are regular expressions, then
   $L(\alpha\beta) = L(\alpha)\cdot L(\beta)$
   = all strings that can be formed by concatenating to some string from L(α) some string from L(β).
   Note that if either α or β is ∅, then its language is ∅, so there is nothing to concatenate and the result is ∅.
3. If α , β are regular expressions, then $L(\alpha\cup\beta) = L(\alpha) \cup L(\beta)$
4. If α is a regular expression, then $L(\alpha^*) = L(\alpha)^*$
5. $L((\alpha)) = L(\alpha)$

A language is **regular** if and only if it can be described by a regular expression.

A regular expression is always finite, but it may describe a (countably) infinite language.

Regular Languages

An equivalent definition of the class of regular languages over an alphabet Σ:
The ***closure*** of the languages

{a} ∀a ∈ Σ and ∅ [1]

with respect to the functions:

- concatenation, [2]
- union, and [3]
- Kleene star. [4]

In other words, the class of regular languages is the smallest set that includes all elements of [1] and that is closed under [2], [3], and [4].

"Closure" and "Closed"

Informally, a set can be defined in terms of a (usually small) starting set and a group of functions over elements from the set. The functions are applied to members of the set, and if anything new arises, it's added to the set. The resulting set is called the **closure** over the initial set and the functions. Note that the functions(s) may only be applied a **finite** number of times.

Examples:

The set of natural numbers **N** can be defined as the closure over {0} and the successor (succ(n) = n+1) function.
Regular languages can be defined as the closure of {a} ∀a∈Σ and ∅ and the functions of concatenation, union, and Kleene star.

We say a set is **closed** over a function if applying the function to arbitrary elements in the set does not yield any new elements.

Examples:

The set of natural numbers **N** is closed under multiplication.
Regular languages are closed under intersection.

See *Supplementary Materials—Review of Mathematical Concepts* for more formal definitions of these terms.

Examples of Regular Languages

L(a*b*) =
L((a ∪ b)) =
L((a ∪ b)*) =
L((a∪b)*a*b*) =
L = {w ∈ {a,b}* : |w| is even}
L = {w ∈ {a,b}* : w contains an odd number of a's}

Augmenting Our Notation

It would be really useful to be able to write ε in a regular expression.

Example: (a ∪ ε) b (Optional a followed by b)

But we'd also like a minimal definition of what constitutes a regular expression. Why?

Observe that

$\emptyset^0$ = {ε} (since 0 occurrences of the elements of any set generates the empty string), so
∅* = {ε}

So, without changing the set of languages that can be defined, we can add ε to our notation for regular expressions if we specify that

L(ε) = {ε}

We're essentially treating ε the same way that we treat the characters in the alphabet.
Having done this, you'll probably find that you rarely need ∅ in any regular expression.

More Regular Expression Examples

L((aa*) $\cup$ ε) =
L((a $\cup$ ε)*) =
L = { w $\in$ {a,b}* : there is no more than one b}
L = { w $\in$ {a,b}* : no two consecutive letters are the same}

Further Notational Extensions of Regular Expressions

- A fixed number of concatenations: α^n means $\alpha\alpha\alpha\alpha\ldots\alpha$ (n times).
- At Least 1: α^+ means 1 or more occurrences of α concatenated together.
- Shorthands for denoting sets, such as ranges, e.g., (A-Z) or (letter-letter)
  Example: L = $(A\text{-}Z)^+((A\text{-}Z)\cup(0\text{-}9))^*$

- A replicated regular expression α^n, where n is a constant.
  Example: L = $(0 \cup 1)^{20}$

- Intersection: $\alpha\cap\beta$ (we'll prove later that regular languages are closed under intersection)
  Example: L = $(a^3)^* \cap (a^5)^*$

Operator Precedence in Regular Expressions

Regular expressions are strings in the language of regular expressions. Thus to interpret them we need to:
1. Parse the string
2. Assign a meaning to the parse tree

Parsing regular expressions is a lot like parsing arithmetic expressions. To do it, we must assign precedence to the operators:

		Regular Expressions	Arithmetic Expressions
Highest		Kleene star	exponentiation
	↕	concatenation	multiplication
		intersection	
Lowest		union	addition

a b* $\cup$ c d*

$x\,y^2 + i\,j^2$

Regular Expressions and Grammars

Recall that grammars are language generators. A grammar is a recipe for creating strings in a language.

Regular expressions are analogous to grammars, but with two special properties:

1. The have limited power. They can be used to define only regular languages.
2. They don't look much like other kinds of grammars, which generally are composed of sets of production rules.

But we can write more "standard" grammars to define exactly the same languages that regular expressions can define. Specifically, any such grammar must be composed of rules that:

- have a left hand side that is a single nonterminal
- have a right hand side that is ε, or a single terminal, or a single terminal followed by a single nonterminal.

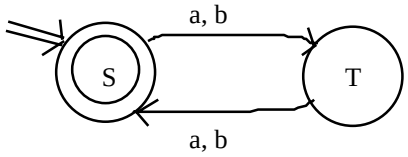
Regular Grammar Example

L={w ∈ {a, b}* : |w| is even}

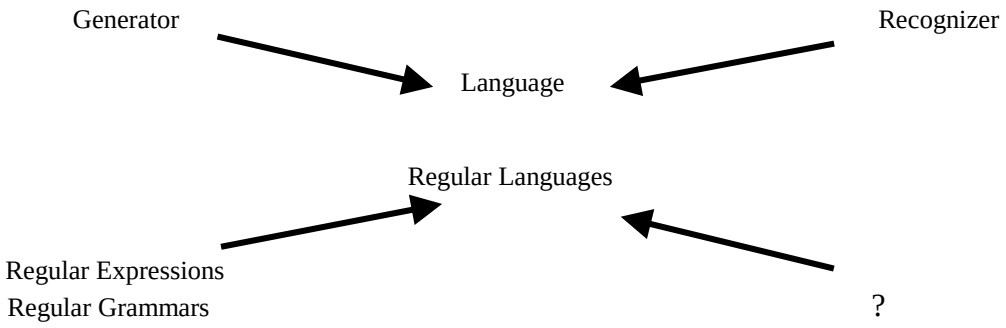
((aa) ∪ (ab) ∪ (ba) ∪ (bb))*

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$
$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$

Notice how these rules correspond naturally to a FSM:



Generators and Recognizers



Finite State Machines

Read K & S 2.1
Do Homeworks 4 & 5.

Finite State Machines 1

A DFSM to accept odd integers:

Definition of a Deterministic Finite State Machine (DFSM)

M = (K, Σ, δ, s, F), where
- K is a finite set of states
- Σ is an alphabet
- $s \in K$ is the initial state
- $F \subseteq K$ is the set of final states, and
- δ is the transition function. It is function from $(K \times \Sigma)$ to K

i.e., each element of δ maps from: a state, input symbol pair to a new state.

Informally, M **accepts** a string w if M winds up in some state that is an element of F when it has finished reading w (if not, it **rejects** w).

The **language accepted by M**, denoted L(M), is the set of all strings accepted by M.

Deterministic finite state machines (DFSMs) are also called deterministic finite state automata (DFSAs or DFAs).

Computations Using FSMs

A **computation** of A FSM is a sequence of configurations, where a configuration is any element of $K \times \Sigma^*$.
The **yields** relation $|-_M$:
(q, w) $|-_M$ (q', w') iff
- w = a w' for some symbol $a \in \Sigma$, and
- δ (q, a) = q'

(The yields relation effectively runs M one step.)

$|-_M^*$ is the reflexive, transitive closure of $|-_M$.
(The $|-_M^*$ relation runs M any number of steps.)

Formally, a FSM M **accepts** a string w iff
(s, w) $|-_M^*$ (q, ε), for some $q \in F$.

An Example Computation

A DFSM to accept odd integers:
On input 235, the configurations are:
(q0, 235) $|-_M$ (q0, 35)
$|-_M$
$|-_M$

Thus (q0, 235) $|-_{M^*}$ (q1, ε). (What does this mean?)

Finite State Machines 2

A DFSM to accept $.50 in change:

More Examples

((aa) ∪ (ab) ∪ (ba) ∪ (bb))*

(b ∪ ε)(ab)*(a ∪ ε)

More Examples

L1 = {w ∈ {a, b}* : every a is immediately followed a b}

A regular expression for L1:

A DFSM for L1:

L2 = {w ∈ {a, b}* : every a has a matching b somewhere before it}

A regular expression for L2:

A DFSM for L2:

Another Example: Socket-based Network Communication

Client	Server	Σ = {Open, Req, Reply, Close}
open socket		
send request		
	send reply	L = Open (Req Reply)* (Req $\cup$ ε) Close
send request		
	send reply	
…		M =
close socket		

Definition of a Deterministic Finite State Transducer (DFST)

M = (K, Σ, O, δ, s, F), where

K is a finite set of states
Σ is an input alphabet
O is an output alphabet
$s \in K$ is the initial state
$F \subseteq K$ is the set of final states, and
δ is the transition function. It is function from

$(K \times \Sigma)$ to $(K \times O^*)$

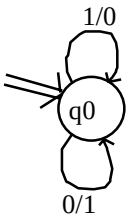
i.e., each element of δ maps from: a state, input symbol pair
to : a new state and zero or more output symbols (an output string)

M **computes a function M(w)** if, when it reads w, it outputs M(w).

Theorem: The output language of a deterministic finite state transducer (on final state) is regular.

A Simple Finite State Transducer

Convert 1's to 0's and 0's to 1's (this isn't just a finite state task -- it's a one state task)



An Odd Parity Generator

After every three bits, output a fourth bit such that each group of four bits has odd parity.

Nondeterministic Finite State Machines

Read K & S 2.2, 2.3
Read Supplementary Materials: Regular Languages and Finite State Machines: Proof of the Equivalence of Nondeterministic and Deterministic FSAs.
Do Homework 6.

Definition of a Nondeterministic Finite State Machine (NDFSM/NFA)

$M = (K, \Sigma, \Delta, s, F)$, where

K is a finite set of states
Σ is an alphabet
$s \in K$ is the initial state
$F \subseteq K$ is the set of final states, and
Δ is the transition *relation*. It is a finite subset of

$$(K \times (\Sigma \cup \{\varepsilon\})) \times K$$

i.e., each element of Δ contains:
a configuration (state, input symbol or ε), and a new state.

M accepts a string w if there exists *some path* along which w drives M to some element of F.

The language accepted by M, denoted L(M), is the set of all strings accepted by M, where computation is defined analogously to DFSMs.

A Nondeterministic FSA

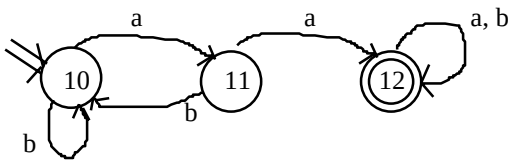
$L = \{w :$ there is a symbol $a_i \in \Sigma$ not appearing in w$\}$

The idea is to guess (nondeterministically) which character will be the one that doesn't appear.

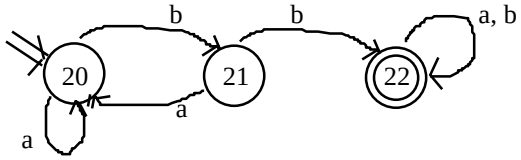
Another Nondeterministic FSA

$L_1 = \{w :$ aa occurs in w$\}$
$L_2 = \{x :$ bb occurs in x$\}$
$L_3 = \{y : \in L_1$ or $L_2\}$

M_1 =

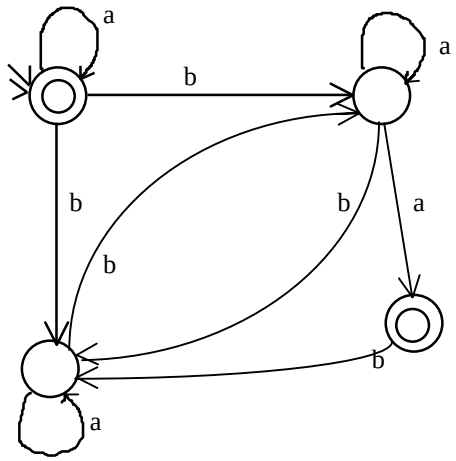


M_2=



M_3=

Analyzing Nondeterministic FSAs

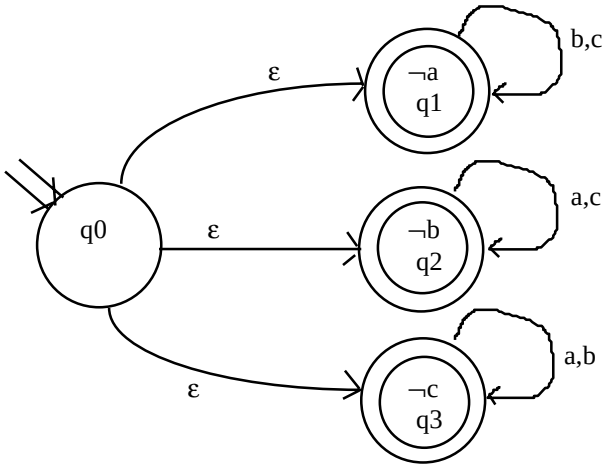


Does this FSA accept: baaba
Remember: we just have to find one accepting path.

Nondeterministic and Deterministic FSAs

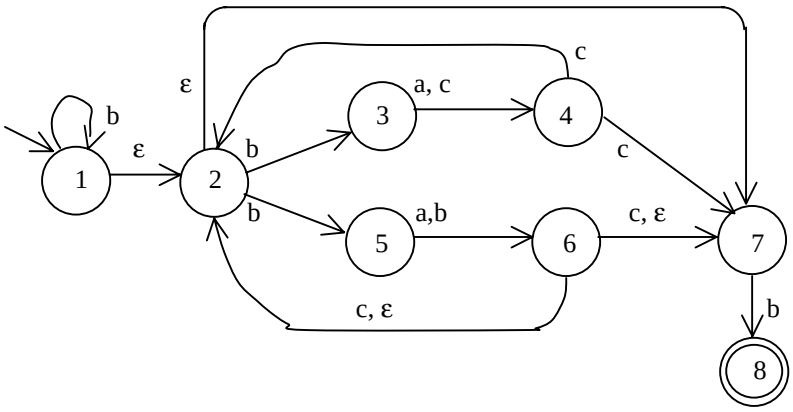
Clearly, {Languages accepted by a DFSA} ⊆ {Languages accepted by a NDFSA}
(Just treat δ as Δ)

More interestingly, **Theorem**: For each NDFSA, there is an equivalent DFSA.
Proof: By construction

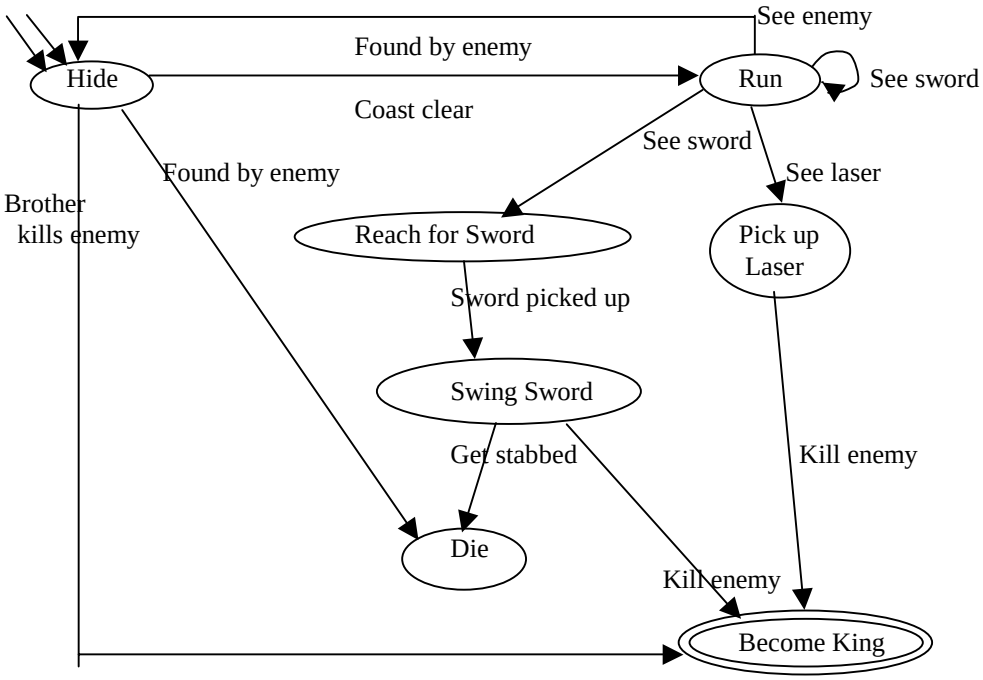


Another Nondeterministic Example

b* (b(a ∪ c)c ∪ b(a ∪ b) (c ∪ ε))* b



A "Real" Example



Dealing with ε Transitions

$E(q) = \{p \in K : (q,w) \vdash^*_M (p, w\}$. $E(q)$ is the closure of $\{q\}$ under the relation $\{(p,r) :$ there is a transition $(p, \varepsilon, r) \in \Delta\}$
An algorithm to compute $E(q)$:

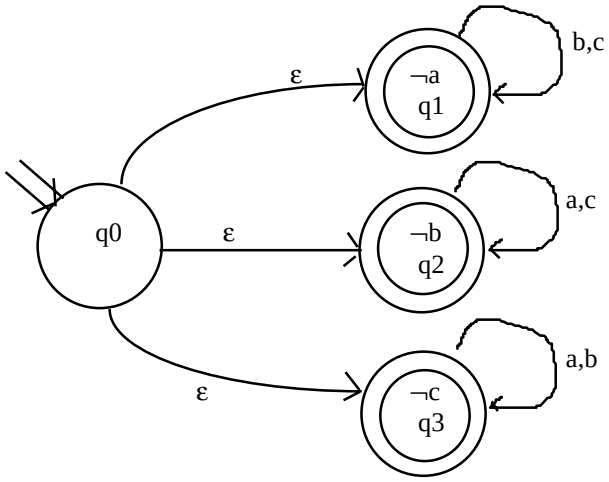
Defining the Deterministic FSA

Given a NDFSA $M = (K, \Sigma, \Delta, s, F)$,
we construct $M' = (K', \Sigma, \delta', s', F')$, where
$K' = 2^K$
$s' = E(s)$
$F' = \{Q \subseteq K : Q \cap F \neq \varnothing\}$
$\delta'(Q, a) = \cup\{E(p) : p \in K$ and $(q, a, p) \in \Delta$
for some $q \in Q\}$

Example: computing δ' for the missing letter machine

$s' =$ $\{q0, q1, q2, q3\}$

$\delta' =$ { $(\{q0, q1, q2, q3\}, a, \{q2, q3\})$,
$(\{q0, q1, q2, q3\}, b, \{q1, q3\})$,
$(\{q0, q1, q2, q3\}, c, \{q1, q2\})$,
$(\{q1, q2\}, a, \{q2\})$, $(\{q1, q2\}, b, \{q1\})$, $(\{q1, q2\}, c, \{q1, q2\})$
$(\{q1, q3\}, a, \{q3\})$, $(\{q1, q3\}, b, \{q1, q3\})$, $(\{q1, q3\}, c, \{q1\})$
$(\{q2, q3\}, a, \{q2, q3\})$, $(\{q2, q3\}, b, \{q3\})$, $(\{q2, q3\}, c, \{q2\})$
$(\{q1\}, b, \{q1\})$, $(\{q1\}, c, \{q1\})$
$(\{q2\}, a, \{q2\})$, $(\{q2\}, c, \{q2\})$
$(\{q3\}, a, \{q3\})$, $(\{q3\}, b, \{q3\})$ }

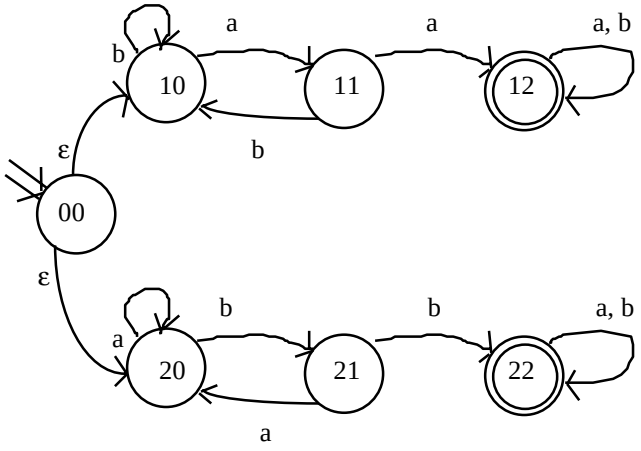


An Algorithm for Constructing the Deterministic FSA

1. Compute the E(q)s:
2. Compute s' = E(s)
3. Compute δ':
   δ' (Q, a) = ∪{E(p) : p ∈ K and (q, a, p) ∈ Δ for some q ∈ Q}
4. Compute K' = a subset of 2^K
5. Compute F' = {Q ∈ K' : Q ∩ F ≠ ∅ }

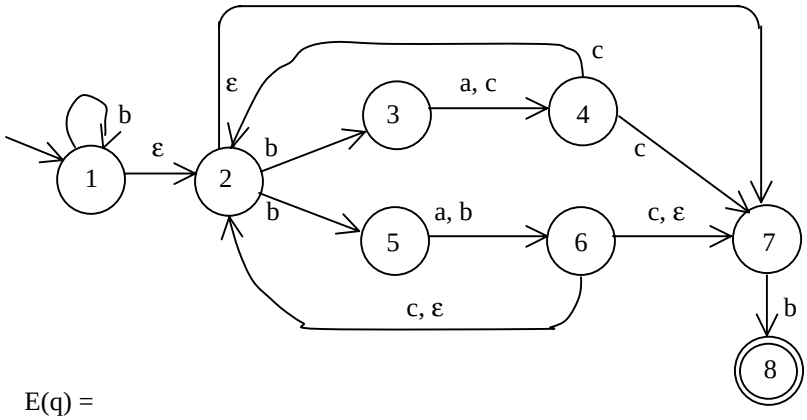
An Example - The Or Machine

L_1= {w : aa occurs in w}
L_2= {x : bb occurs in x}
L_3= {y : ∈ L_1 or L_2 }



Another Example

b* (b(a ∪ c)c ∪ b(a ∪ b) (c ∪ ε))* b



E(q) =

δ' =

Sometimes the Number of States Grows Exponentially

Example: The missing letter machine, with $|\Sigma| = n$

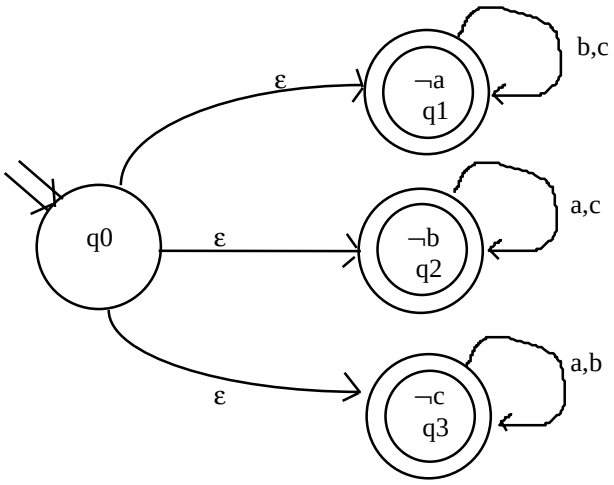
No. of states after 0 chars: 1

No. of new states after 1 char: $\binom{n}{n-1} = n$

No. of new states after 2 chars: $\binom{n}{n-2} = n(n-1)/2$

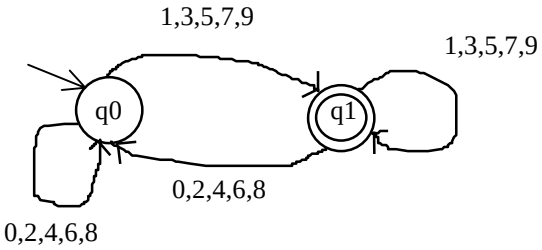
No. of new states after 3 chars: $\binom{n}{n-3} = n(n-1)(n-2)/6$

Total number of states after n chars: 2^n



What If The Original FSA is Deterministic?

M=



1. Compute the E(q)s:
2. s' = E(q0) =
3. Compute δ'
   ({q0}, odd, {q1})
   ({q0}, even, {q0})
   ({q1}, odd, {q1})
   ({q1}, even, {q0})
4. K' = {{q0}, {q1}}
5. F' = {{q1}}

M' = M

The real meaning of "determinism"

A FSA is **deterministic** if, for each input and state, there is at most one possible transition.

DFSAs are always deterministic. Why?

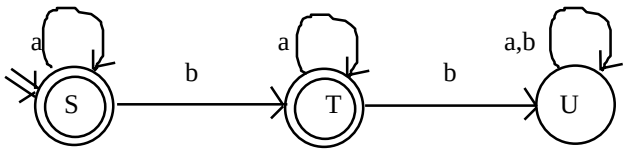
NFSAs can be deterministic (even with ε-transitions and implicit dead states), but the formalism allows nondeterminism, in general.

Determinism implies uniquely defined machine behavior.

Interpreters for Finite State Machines

Deterministic FSAs as Algorithms

Example: No more than one b




```
S:  s := get-next-symbol;
    if s = end-of-file then accept;
    else if s = a then go to S;
    else if s = b then go to T;
T:  s:= get-next-symbol;
    if s = end-of-file then accept;
    else if s = a then go to T;
    else if s = b then go to U;
etc.
```

Length of Program: $|K| \times (|\Sigma| + 2)$
Time required to analyze string w: $O(|w| \times |\Sigma|)$

We have to write new code for every new FSM.

Until accept or reject do:

A Deterministic FSA Interpreter

To simulate M = $(K, \Sigma, \delta, s, F)$:

```
ST := s;
Repeat
        i := get-next-symbol;
        if i ≠ end-of-string then
                ST := δ(ST, i)
Until i = end-of-string;
If ST ∈ F then accept else reject
```

Simulate the no more than one b machine on input: aabaa

Nondeterministic FSAs as Algorithms

Real computers are deterministic, so we have three choices if we want to execute a nondeterministic FSA:

1. Convert the NDFSA to a deterministic one:
   - Conversion can take time and space 2^K.
   - Time to analyze string w: $O(|w|)$

2. Simulate the behavior of the nondeterministic one by constructing sets of states "on the fly" during execution
   - No conversion cost
   - Time to analyze string w: $O(|w| \times K^2)$

3. Do a depth-first search of all paths through the nondeterministic machine.

A Nondeterministic FSA Interpreter

To simulate $M = (K, \Sigma, \Delta, s, F)$:

```
SET ST;
ST := E(s);
Repeat
        i := get-next-symbol;
        if  i ≠ end-of-string then
                ST1 := ∅
                For all q ∈ ST do
                        For all r ∈ Δ(q, i) do
                                ST1 := ST1 ∪ E(r);
                ST := ST1;

Until i = end-of-string;
If  ST ∩  F ≠ ∅ then accept else reject
```

A Deterministic Finite State Transducer Interpreter

To simulate $M = (K, \Sigma, O, \delta, s, F)$, given that:

δ_1(state, symbol) returns a single new state (i.e., M is deterministic), and

δ_2(state, symbol) returns an element of O^*, the string to be output.

```
ST := s;
Repeat:
        i := get-next-symbol;
        if  i ≠ end-of-string then
                write(δ2(ST, i));
                ST := δ1(ST, i)
Until i = end-of-string;
If  ST ∈ F then accept else reject
```

Equivalence of Regular Languages and FSMs

Read K & S 2.4
Read Supplementary Materials: Regular Languages and Finite State Machines: Generating Regular Expressions from Finite State Machines.
Do Homework 8.

Equivalence of Regular Languages and FSMs

Theorem: The set of languages expressible using regular expressions (the regular languages) equals the class of languages recognizable by finite state machines. Alternatively, a language is regular if and only if it is accepted by a finite state machine.

Proof Strategies

Possible Proof Strategies for showing that two sets, *a* and *b* are equal (also for iff):

1. Start with *a* and apply valid transformation operators until *b* is produced.

Example:
Prove:
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

$A \cap (B \cup C)$	$= (B \cup C) \cap A$	commutativity
	$= (B \cap A) \cup (C \cap A)$	distributivity
	$= (A \cap B) \cup (A \cap C)$	commutativity

2. Do two separate proofs: (1) a $\Rightarrow$ b, and (2) b $\Rightarrow$a, possibly using totally different techniques. In this case, we show first (by construction) that for every regular expression there is a corresponding FSM. Then we show, by induction on the number of states, that for every FSM, there is a corresponding regular expression.

For Every Regular Expression There is a Corresponding FSM

We'll show this by construction.

Example:

$a^*(b \cup \varepsilon)a^*$

Review - Regular Expressions

The regular expressions over an alphabet Σ^* are all strings over the alphabet $\Sigma \cup \{(,), \varnothing, \cup, *\}$ that can be obtained as follows:

1. $\varnothing$ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is $\alpha\beta$.
3. If α , β are regular expressions, then so is $\alpha \cup \beta$.
4. If α is a regular expression, then so is α^*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

We also allow ε and α^+, etc. but these are just shorthands for $\varnothing^*$ and $\alpha\alpha^*$, etc. so they do not need to be considered for completeness.

For Every Regular Expression There is a Corresponding FSM

Formalizing the Construction: The class of regular languages is the smallest class of languages that contains ∅ and each of the singleton strings drawn from Σ, and that is closed under

- Union
- Concatenation, and
- Kleene star

Clearly we can construct an FSM for any finite language, and thus for ∅ and all the singleton strings. If we could show that the class of languages accepted by FSMs is also closed under the operations of union, concatenation, and Kleene star, then we could recursively construct, for any regular expression, the corresponding FSM, starting with the singleton strings and building up the machine as required by the operations used to express the regular expression.

FSMs for Primitive Regular Expressions

An FSM for ∅:

An FSM for ε (∅*):

An FSM for a single element of Σ:

Closure of FSMs Under Union

To create a FSM that accepts the union of the languages accepted by machines M1 and M2:

1. Create a new start state, and, from it, add ε-transitions to the start states of M1 and M2.

Closure of FSMs Under Concatenation

To create a FSM that accepts the concatenation of the languages accepted by machines M1 and M2:

1. Start with M1.
2. From every final state of M1, create an ε-transition to the start state of M2.
3. The final states are the final states of M2.

Closure of FSMs Under Kleene Star

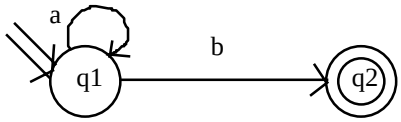
To create an FSM that accepts the Kleene star of the language accepted by machine M1:
1. Start with M1.
2. Create a new start state S0 and make it a final state (so that we can accept ε).
3. Create an ε-transition from S0 to the start state of M1.
4. Create ε-transitions from all of M1's final states back to its start state.
5. Make all of M1's final states final.

Note: we need a new start state, S0, because the start state of the new machine must be a final state, and this may not be true of M1's start state.

Closure of FSMs Under Complementation

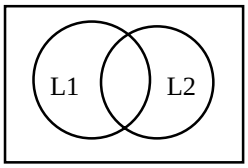
To create an FSM that accepts the complement of the language accepted by machine M1:
1. Make M1 deterministic.
2. Reverse final and nonfinal states.

A Complementation Example



Closure of FSMs Under Intersection

$L1 \cap L2 =$



Write this in terms of operations we have already proved closure for:

- Union
- Concatenation
- Kleene star
- Complementation

An Example

$(b \cup ab^*a)^*ab^*$

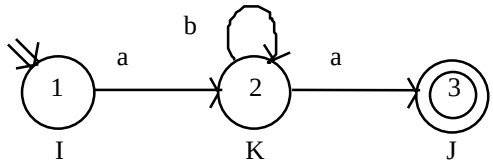
For Every FSM There is a Corresponding Regular Expression

Proof:
(1) There is a trivial regular expression that describes the strings that can be recognized in going from one state to itself ({ε} plus any other single characters for which there are loops) or from one state to another directly (i.e., without passing through any other states), namely all the single characters for which there are transitions.

(2) Using (1) as the base case, we can build up a regular expression for an entire FSM by induction on the number assigned to possible intermediate states we can pass through. By adding them in only one at a time, we always get simple regular expressions, which can then be combined using union, concatenation, and Kleene star.

Key Ideas in the Proof

Idea 1: Number the states and, at each induction step, increase by one the states that can serve as intermediate states.

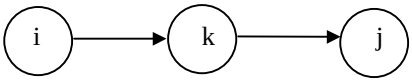


Idea 2: To get from state I to state J without passing through any intermediate state numbered greater than K, a machine may either:
1. Go from I to J without passing through any state numbered greater than K-1 (which we'll take as the induction hypothesis), or
2. Go from I to K, then from K to K any number of times, then from K to J, in each case without passing through any intermediate states numbered greater than K-1 (the induction hypothesis, again).

So we'll start with no intermediate states allowed, then add them in one at a time, each time building up the regular expression with operations under which regular languages are closed.

The Formula

Adding in state k as an intermediate state we can use to go from i to j, described using paths that don't use k:



R(i, j, k) = R(i, j, k - 1) /* what you could do without k

∪

R(i, k, k-1) /* go from i to the new intermediate state without using k or higher

∘

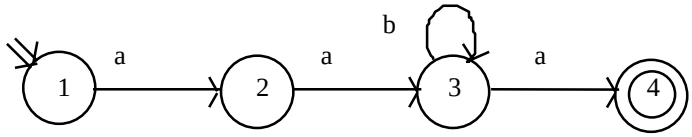
R(k, k, k-1)* /* then go from the new intermediate state back to itself as many times as you want

∘

R(k, j, k-1) /* then go from the new intermediate state to j without using k or higher

Solution: $\bigcup R(s, q, N) \ \forall q \in F$

An Example of the Induction



Going through no intermediate states:

(1,1,0) = ε (1,2,0) = a (1, 3, 0) = ∅ (2,3,0) = a (3,3,0) = ε ∪ b (3,4,0) = a

Allow 1 as an intermediate state:

Allow 2 as an intermediate state:

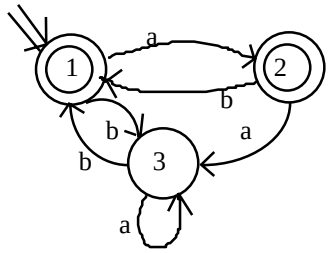
(1, 3, 2) = (1, 3, 1) ∪ (1, 2, 1)(2, 2, 1)*(2, 3, 1)
= ∅ ∪ a ε* a
= aa

Allow 3 as an intermediate state:

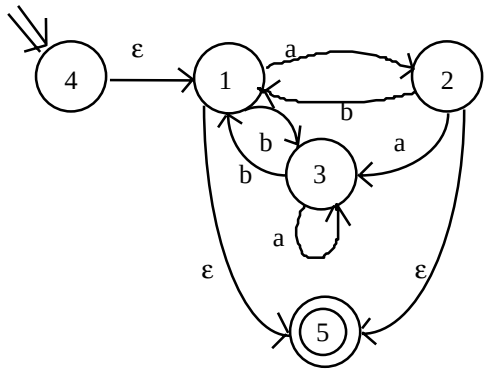
(1, 3, 3) = (1, 3, 2) ∪ (1, 3, 2)(3, 3, 2)*(3, 3, 2)
= aa ∪ aa (ε ∪ b)* (ε ∪ b)
= aab*

(1, 4, 3) = (1, 4, 2) ∪ (1, 3, 2)(3, 3, 2)*(3, 4, 2)
= ∅ ∪ aa (ε ∪ b)* a
= aab*a

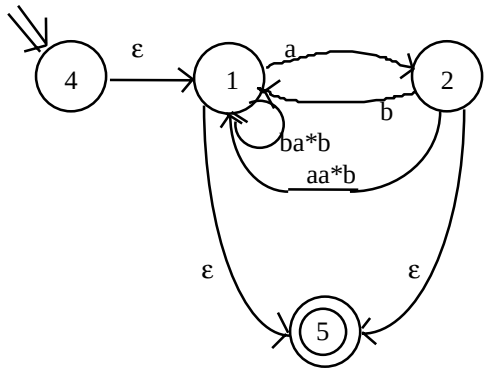
An Easier Way - See Packet



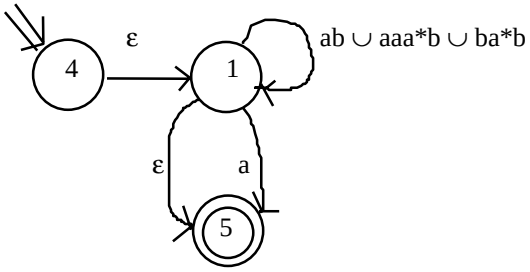
(1) Create a new initial state and a new, unique final state, neither of which is part of a loop.

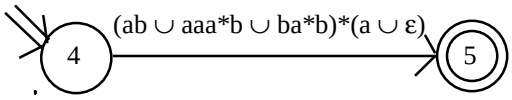


(2) Remove states and arcs and replace with arcs labelled with larger and larger regular expressions. States can be removed in any order, but don't remove either the start or final state.



(Notice that the removal of state 3 resulted in two new paths because there were two incoming paths to 3 from another state and 1 outgoing path to another state, so 2×1 = 2.) The two paths from 2 to 1 should be coalesced by unioning their regular expressions (not shown).





Thus, the equivalent regular expression is:

(ab ∪ aaa*b ∪ ba*b)*(a ∪ ε)

Using Regular Expressions in the Real World (PERL)

Matching floating point numbers:

-? ([0-9]+(\.[0-9]*)? | \.[0-9]+)

Matching IP addresses:

([0-9]+ (\ . [0-9]+) {3})

Finding doubled words:

\< ([A-Za-z]+) \s+ \1 \>

From Friedl, J., Mastering Regular Expressions, O'Reilly,1997.

Note that some of these constructs are more powerful than regular expressions.

Regular Grammars and Nondeterministic FSAs

Any regular language can be defined by a regular grammar, in which all rules

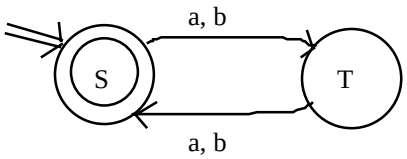
- have a left hand side that is a single nonterminal
- have a right hand side that is ε, a single terminal, a single nonterminal, or a single terminal followed by a single nonterminal.

Example: $L=\{w \in \{a, b\}^* : |w| \text{ is even}\}$

$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$

$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$



An Algorithm to Generate the NDFSM from a Regular Grammar

1. Create a nonterminal for each state in the NDFSM.
2. s is the start state.
3. If there are any rules of the form $X \rightarrow w$, for some $w \in \Sigma$, then create an additional state labeled #.
4. For each rule of the form $X \rightarrow w\ Y$, add a transition from X to Y labeled w ($w \in \Sigma \cup \varepsilon$).
5. For each rule of the form $X \rightarrow w$, add a transition from X to # labeled w ($w \in \Sigma$).
6. For each rule of the form $X \rightarrow \varepsilon$, mark state X final.
7. Mark state # final.

Example 1 - Even Length Strings

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$

$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$

Example 2 - One Character Missing

$S \rightarrow \varepsilon$
$S \rightarrow aB$
$S \rightarrow aC$
$S \rightarrow bA$
$S \rightarrow bC$
$S \rightarrow cA$
$S \rightarrow cB$

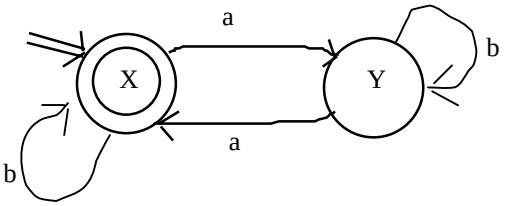
$A \rightarrow bA$
$A \rightarrow cA$
$A \rightarrow \varepsilon$
$B \rightarrow aB$
$B \rightarrow cB$
$B \rightarrow \varepsilon$

$C \rightarrow aC$
$C \rightarrow bC$
$C \rightarrow \varepsilon$

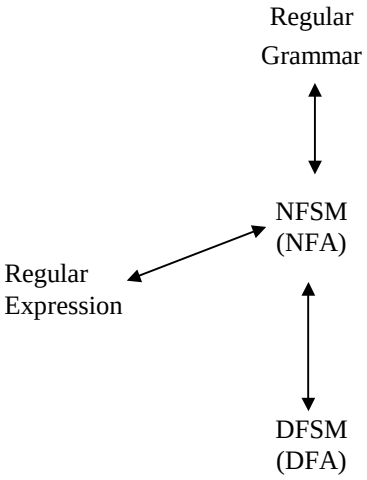
An Algorithm to Generate a Regular Grammar from an NDFSM

1. Create a nonterminal for each state in the NDFSM.
2. The start state becomes the starting nonterminal
3. For each transition $\delta(T, a) = U$, make a rule of the form $T \rightarrow aU$.
4. For each final state T, make a rule of the form $T \rightarrow \varepsilon$.

Example:



Conversion Algorithms between Regular Language Formalisms



Languages That Are and Are Not Regular

Read L & S 2.5, 2.6
Read Supplementary Materials: Regular Languages and Finite State Machines: The Pumping Lemma for Regular Languages.
Do Homework 9.

Deciding Whether a Language is Regular

Theorem: There exist languages that are not regular.
Lemma: There are an uncountable number of languages.
Proof of Lemma:
Let: Σ be a finite, nonempty alphabet, e.g., {a, b, c}.

Then Σ^* contains all finite strings over Σ.
e.g., {ε, a, b, c, aa, ab, bc, abc, bba, bbaa, bbbaac}

Σ^* is countably infinite, because its elements can be enumerated one at a time, shortest first.

Any language L over Σ is a subset of Σ^*, e.g., L1 = {a, aa, aaa, aaaa, aaaaa, …}
L2 = {ab, abb, abbb, abbbb, abbbbb, …}
The set of all possible languages is thus the power set of Σ^*.

The power set of any countably infinite set is not countable. So there are an uncountable number of languages over Σ^*.

Some Languages Are Not Regular

Theorem: There exist languages that are not regular.
Proof:
(1) There are a countably infinite number of regular languages. This true because every description of a regular language is of finite length, so there is a countably infinite number of such descriptions.
(2) There are an uncountable number of languages.

Thus there are more languages than there are regular languages. So there must exist some language that is not regular.

Showing That a Language is Regular

Techniques for showing that a language L is regular:
1. Show that L has a finite number of elements.
2. Exhibit a regular expression for L.
3. Exhibit a FSA for L.
4. Exhibit a regular grammar for L.
5. Describe L as a function of one or more other regular languages and the operators $\cdot, \cup, \cap, *, -, \neg$. We use here the fact that the regular languages are closed under all these operations.
6. Define additional operators and prove that the regular languages are closed under them. Then use these operators as in 5.

Example

Let Σ = {0, 1, 2, … 9}
Let L $\subseteq \Sigma^*$ be the set of decimal representations for nonnegative integers (with no leading 0's) divisible by 2 or 3.

L_1 = decimal representations of nonnegative integers without leading 0's.
$L_1 = 0 \cup \{1, 2, \ldots 9\}\{0 - 9\}^*$
So L_1 is regular.

L_2 = decimal representations of nonnegative integers without leading 0's divisible by 2
$L_2 = L_1 \cap \Sigma^*\{0, 2, 4, 6, 8\}$
So L_2 is regular.

Example, Continued

$L_3 = L_1$ and divisible by 3

Recall that a number is divisible by 3 if and only if the sum of its digits is divisible by 3. We can build a FSM to determine that and accept the language L_{3a}, which is composed of strings of digits that sum to a multiple of 3.

$L_3 = L_1 \cap L_{3a}$

Finally, $L = L_2 \cup L_3$

Another Example

$\Sigma = \{0 - 9\}$
L = {w : w is the social security number of a living US resident}

Finiteness - Theoretical vs. Practical

Any finite language is regular. The size of the language doesn't matter.

Parity ⟵ ⟶ Soc. Sec. #

Checking Checking

But, from an implementation point of view, it very well may.

When is an FSA a good way to encode the facts about a language?

What are our alternatives?

FSA's are good at looking for repeating patterns. They don't bring much to the table when the language is just a set of unrelated strings.

Showing that a Language is Not Regular

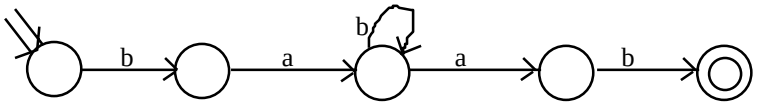
The argument, "I can't find a regular expression or a FSM", won't fly. (But a proof that there cannot exist a FSM is ok.)

Instead, we need to use two fundamental properties shared by regular languages:

1. We can only use a finite amount of memory to record essential properties.
   Example:
   $a^n b^n$ is not regular

2. The only way to generate/accept an infinite language with a finite description is to use Kleene star (in regular expressions) or cycles (in automata). This forces some kind of simple repetitive cycle within the strings.
   Example:
   ab*a generates aba, abba, abbba, abbbba, etc.
   Example:
   $\{a^n : n \geq 1$ is a prime number$\}$ is not regular.

Exploiting the Repetitive Property



If a FSM of n states accepts any string of length ≥ n, how many strings does it accept?

L = bab*ab

n

b a b b b b a b

x y z

xy*z must be in L.

So L includes: baab, babab, babbab, babbbbbbbbbbab

The Pumping Lemma for Regular Languages

If L is regular, then
- ∃ N ≥ 1, such that
  - ∀ strings w ∈ L, where |w| ≥ N,
    - ∃ x, y, z, such that w = xyz
      - and |xy| ≤ N,
      - and y ≠ ε,
      - and ∀ q ≥ 0, xy^qz is in L.

Example: L = a^nb^n

a a a a a a a a a b b b b b b b b b b

x y z

∃ N ≥ 1	Call it N
∀ long strings w	We pick one
∃ x, y, z	We show no x, y, z

Example: a^nb^n is not Regular

N is the number from the pumping lemma (or one more, if N is odd).

Choose w = $a^{N/2}b^{N/2}$. (Since this is what it takes to be "long enough": |w| ≥ N)

1 2

a a a a a a a a a a | b b b b b b b b b b

x y z

We show that there is no x, y, z with the required properties:

|xy| ≤ N,
y ≠ ε,
∀ q ≥ 0, xy^qz is in L.

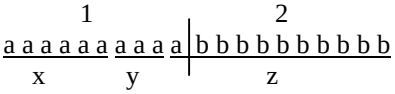
Three cases to consider:

- y falls in region 1:
- y falls across regions 1 and 2:
- y falls in region 3:

Example: $a^n b^n$ is not Regular

Second try:

Choose w to be be $a^N b^N$. (Since we get to choose any w in L.)



We show that there is no x, y, z with the required properties:

$|xy| \le N$,
$y \ne \varepsilon$,
$\forall\ q \ge 0$, $xy^q z$ is in L.

Since $|xy| \le N$, y must be in region 1. So $y = a^g$ for some $g \ge 1$. Pumping in or out (any q but 1) will violate the constraint that the number of a's has to equal the number of b's.

A Complete Proof Using the Pumping Lemma

Proof that $L = \{a^n b^n\}$ is not regular:

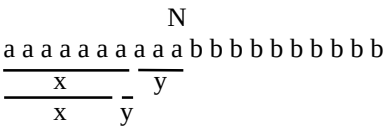
Suppose L is regular. Since L is regular, we can apply the pumping lemma to L. Let N be the number from the pumping lemma for L. Choose $w = a^N b^N$. Note that $w \in L$ and $|w| \ge N$. From the pumping lemma, there exists some x, y, z where xyz = w and $|xy| \le N$, $y \ne \varepsilon$, and $\forall\ q \ge 0$, $xy^q z \in L$. Because $|xy| \le N$, $y = a^{|y|}$ (y is all a's). We choose q = 2 and $xy^q z = a^{N+|y|} b^N$. Because $|y| > 0$, then $xy^2 z \notin L$ (the string has more a's than b's). Thus for all possible x, y, z: xyz = w, $\exists q$, $xy^q z \notin L$. Contradiction. $\therefore$ L is not regular.

Note: the underlined parts of the above proof is "boilerplate" that can be reused. A complete proof should have this text or something equivalent.

You get to choose w. Make it a single string that depends only on N. Choose w so that it makes your proof easier.
You may end up with various cases with different q values that reach a contradiction. You have to show that all possible cases lead to a contradiction.

Proof of the Pumping Lemma

Since L is regular it is accepted by some DFSA, M. Let N be the number of states in M. Let w be a string in L of length N or more.



Then, in the first N steps of the computation of M on w, M must visit N+1 states. But there are only N different states, so it must have visited the same state more than once. Thus it must have looped at least once. We'll call the portion of w that corresponds to the loop y. But if it can loop once, it can loop an infinite number of times. Thus:

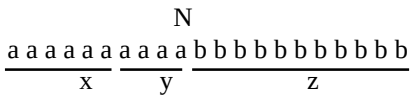
- M can recognize $xy^q z$ for all values of $q \ge 0$.
- $y \ne \varepsilon$ (since there was a loop of length at least one)
- $|xy| \le N$ (since we found y within the first N steps of the computation)

.

Another Pumping Example

$L = \{w=a^Jb^K : K > J\}$ (more b's than a's)

Choose $w = a^Nb^{N+1}$

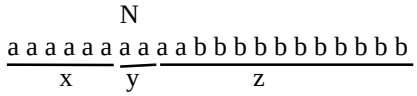


We are guaranteed to pump only a's, since $|xy| \leq N$. So there exists a number of copies of y that will cause there to be more a's than b's, thus violating the claim that the pumped string is in L.

A Slightly Different Example of Pumping

$L = \{w=a^Jb^K : J > K\}$ (more a's than b's)

Choose $w = a^{N+1}b^N$



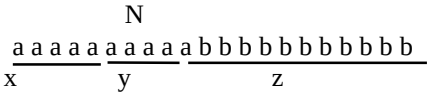
We are guaranteed that y is a string of at least one a, since $|xy| \leq N$. But if we pump in a's we get even more a's than b's, resulting in strings that are in L.

What can we do?

Another Slightly Different Example of Pumping

$L = \{w=a^Jb^K : J \geq K\}$

Choose $w = a^{N+1}b^N$



We are guaranteed that y is a string of at least one a, since $|xy| \leq N$. But if we pump in a's we get even more a's than b's, resulting in strings that are in L.

If we pump out, then if y is just a then we still have a string in L.

What can we do?

Another Pumping Example

$L = aba^nb^n$

Choose $w = aba^Nb^N$

$$\underbrace{a\ b}_{x}\ \underbrace{a}_{y}\ \underbrace{a\ a\ a\ a\ a\ a\ a\ a\ b\ b\ b\ b\ b\ b\ b\ b\ b}_{z} \qquad (N)$$

What are the choices for (x, y):
(ε, a)
(ε, ab)
(ε, aba^+)
(a, b)
(a, ba^+)
(aba*, a^+)

What if L is Regular?

Given a language L that is regular, pumping will work: L = (ab)* Choose $w = (ab)^N$

There must exist an x, y, and z where y is pumpable.

$$\underbrace{abababab}_{x}\ \underbrace{ababab}_{y}\ \underbrace{abababababab}_{z}$$

Suppose y = ababab Then, for all $q \geq 0$, $x\,y^q z \in L$

Note that this does not prove that L is regular. It just fails to prove that it is not.

Using Closure Properties

Once we have some languages that we can prove are not regular, such as a^nb^n, we can use the closure properties of regular languages to show that other languages are also not regular.

Example: $\Sigma = \{a, b\}$
$L = \{w : w$ contains an equal number of a's and b's $\}$
a*b* is regular. So, if L is regular, then $L_1 = L \cap a^*b^*$ is regular.

But L_1 is precisely a^nb^n. So L is not regular.

Don't Try to Use Closure Backwards

One Closure Theorem:
If L_1 and L_2 are regular, then so is $L_3 = L_1 \cap L_2$.

But what if L_3 and L_1 are regular? What can we say about L_2?

$L_3 = L_1 \cap L_2$.

Example: $ab = ab \cap a^nb^n$

A Harder Example of Pumping

$\Sigma = \{a\}$
$L = \{w = a^K : K \text{ is a prime number}\}$

N
a a a a a a a a a a a a a
x y z

$|x| + |z|$ is prime.
$|x| + |y| + |z|$ is prime.
$|x| + 2|y| + |z|$ is prime.
$|x| + 3|y| + |z|$ is prime, and so forth.

Distribution of primes:

Distribution of $|x| + q|y| + |z|$:

But the Prime Number Theorem tells us that the primes "spread out", i.e., that the number of primes not exceeding x is asymptotic to x/ln x.

Note that when $q = |x| + |z|$, $|xy^q z| = (|y| + 1) \times (|x| + |z|)$, which is composite (non-prime) if both factors are > 1. If you're careful about how you choose N in a pumping lemma proof, you can make this true for both factors.

Automata Theory is Just the Scaffolding

Our results so far give us tools to:

- Show a language is regular by:
  - Showing that it has a finite number of elements,
  - Providing a regular expression that defines it,
  - Constructing a FSA that accepts it, or
  - Exploiting closure properties
- Show a language is not regular by:
  - Using the pumping lemma, or
  - Exploiting closure properties.

But to use these tools effectively, we may also need domain knowledge (e.g., the Prime Number Theorem).

More Examples

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$
$L = \{w = \text{the octal representation of a number that is divisible by 7}\}$

Example elements of L:
7, 16 (14), 43 (35), 61 (49), 223 (147)

More Examples

$\Sigma = \{W, H, Q, E, S, T, B \text{ (measure bar)}\}$
$L = \{w = w \text{ represents a song written in 4/4 time}\}$

Example element of L:
WBWBHHBHQQBHHBQEEQEEB

More Examples

$\Sigma = \{0 - 9\}$
L = {w = is a prime Fermat number}

The Fermat numbers are defined by

$$F_n = 2^{2^n} + 1, n = 1, 2, 3, \ldots$$

Example elements of L:
$F_1 = 5$, $F_2 = 17$, $F_3 = 257$, $F_4 = 65{,}537$

Another Example

$\Sigma = \{0 - 9, *, =\}$
L = {w = a*b=c: a, b, c $\in$ $\{0\text{-}9\}^+$ and int(a) * int(b) = int(c)}

The Bottom Line

A language is regular if:

OR

The Bottom Line (Examples)

- The set of decimal representations for nonnegative integers divisible by 2 or 3
- The social security numbers of living US residents.
- Parity checking
- $a^n b^n$
- $a^j b^k$ where k>j
- a^k where k is prime
- The set of strings over {a, b} that contain an equal number of a's and b's.
- The octal representations of numbers that are divisible by 7
- The songs in 4/4 time
- The set of prime Fermat numbers

Decision Procedures

A **decision procedure** is an algorithm that answers a question (usually "yes" or "no") and terminates. The whole idea of a decision procedure itself raises a new class of questions. In particular, we can now ask,

1. Is there a decision procedure for question X?
2. What is that procedure?
3. How efficient is the best such procedure?

Clearly, if we jump immediately to an answer to question 2, we have our answer to question 1. But sometimes it makes sense to answer question 1 first. For one thing, it tells us whether to bother looking for answers to questions 2 and 3.

Examples of Question 1:

Is there a decision procedure, given a regular expression E and a string S, for determining whether S is in L(E)?

Is there a decision procedure, given a Turing machine T and an input string S, for determining whether T halts on S?

Decision Procedures for Regular Languages

Let M be a deterministic FSA. There is a decision procedure to determine whether:

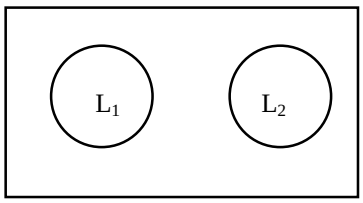
- $w \in L(M)$ for some fixed w
- L(M) is empty
- L(M) is finite
- L(M) is infinite

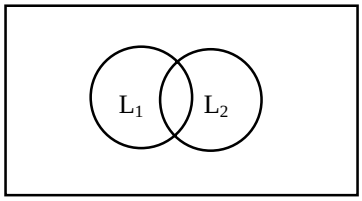
Let M_1 and M_2 be two deterministic FSAs. There is a decision procedure to determine whether M_1 and M_2 are equivalent. Let L_1 and L_2 be the languages accepted by M_1 and M_2. Then the language

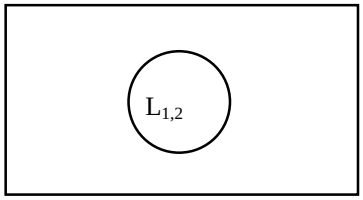
$L = (L_1 \cap \neg L_2) \cup (\neg L_1 \cap L_2)$

$= (L_1 - L_2) \cup (L_2 - L_1)$

must be regular. L is empty iff $L_1 = L_2$. There is a decision procedure to determine whether L is empty and thus whether $L_1 = L_2$ and thus whether M_1 and M_2 are equivalent.







A Review of Equivalence Relations

Do Homework 7.

A Review of Equivalence Relations

A relation R is an equivalence relation if it is: reflexive, symmetric, and transitive.

Example: R = the reflexive, symmetric, transitive closure of:
(Bob, Bill), (Bob, Butch), (Butch, Bud),
(Jim, Joe), (Joe, John), (Joe, Jared),
(Tim, Tom), (Tom, Tad)

An equivalence relation on a nonempty set A creates a partition of A. We write the elements of the partition as $[a_1]$, $[a_2]$, …
Example:

Another Equivalence Relation

Example: R = the reflexive, symmetric, transitive closure of:
(apple, pear), (pear, banana), (pear, peach),
(peas, mushrooms), (peas, onions), (peas, zucchini)
(bread, rice), (rice, potatoes), (rice, pasta)

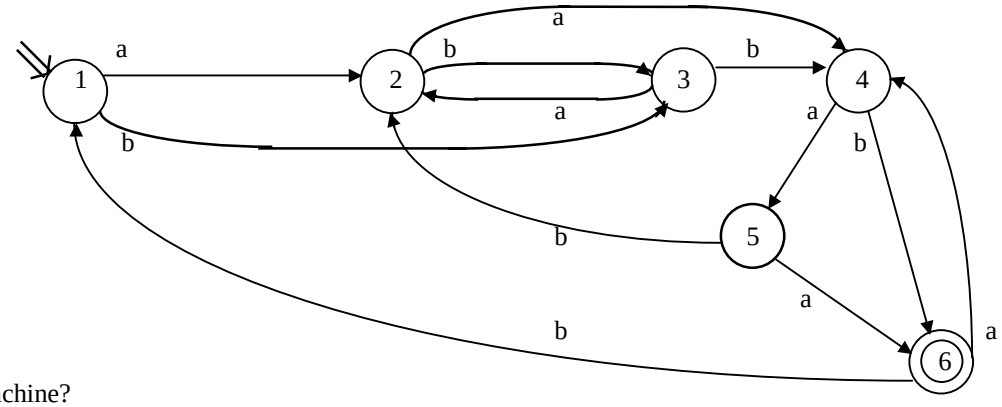
Partition:

State Minimization for DFAs

Read K & S 2.7
Do Homework 10.

State Minimization

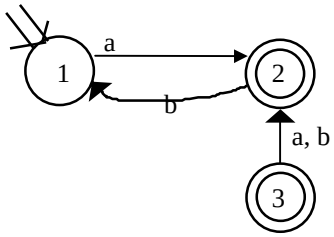
Consider:



Is this a minimal machine?

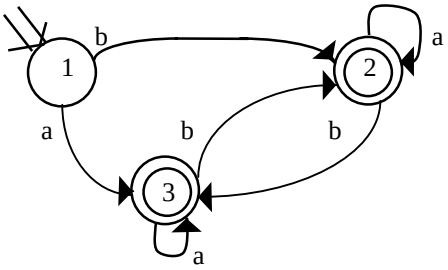
State Minimization

Step (1): Get rid of unreachable states.



State 3 is unreachable.

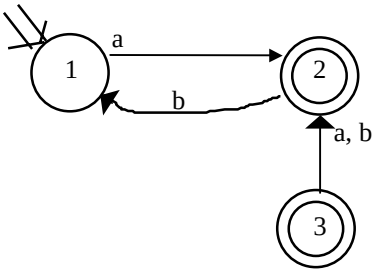
Step (2): Get rid of redundant states.



States 2 and 3 are redundant.

Getting Rid of Unreachable States

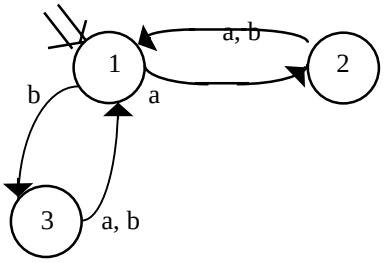
We can't easily find the unreachable states directly. But we can find the reachable ones and determine the unreachable ones from there. An algorithm for finding the reachable states:



Getting Rid of Redundant States

Intuitively, two states are equivalent to each other (and thus one is redundant) if all strings in Σ* have the same fate, regardless of which of the two states the machine is in. But how can we tell this?

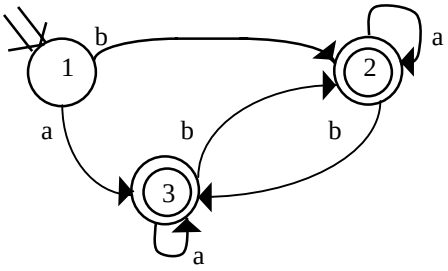
The simple case:



Two states have identical sets of transitions out.

Getting Rid of Redundant States

The harder case:



The outcomes are the same, even though the states aren't.

Finding an Algorithm for Minimization

Capture the notion of equivalence classes of strings with respect to a language.

Capture the (weaker) notion of equivalence classes of strings with respect to a language and a particular FSA.

Prove that we can always find a deterministic FSA with a number of states equal to the number of equivalence classes of strings.

Describe an algorithm for finding that deterministic FSA.

Defining Equivalence for Strings

We want to capture the notion that two strings are equivalent with respect to a language L if, no matter what is tacked on to them on the right, either they will both be in L or neither will. Why is this the right notion? Because it corresponds naturally to what the states of a recognizing FSM have to remember.

Example:

(1)	a	b	b	a	b
(2)	b	a	b	a	b

Suppose L = {w ∈ {a,b}* : |w| is even}. Are (1) and (2) equivalent?

Suppose L = {w ∈ {a,b}* : every a is immediately followed by b}. Are (1) and (2) equivalent?

Defining Equivalence for Strings

If two strings are equivalent with respect to L, we write $x \approx_L y$. Formally, $x \approx_L y$ if, $\forall z \in \Sigma^*$,

$xz \in L$ iff $yz \in L$.

Notice that $\approx_L$ is an equivalence relation.

Example:

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

$L = \{w \in \Sigma^* :$ every a is immediately followed by b $\}$

ε	aa	bbb
a	bb	baa
b	aba	
	aab	

The equivalence classes of $\approx_L$:

$|\approx_L|$ is the number of equivalence classes of $\approx_L$.

Another Example of $\approx_L$

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

$L = \{w \in \Sigma^* : |w|$ is even$\}$

ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\approx_L$:

Yet Another Example of $\approx_L$

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

$L = aab^*a$

ε	ba	aabb
a	bb	aabaa
b	aaa	aabbba
aa	aba	aabbaa
ab	aab	
	bab	

The equivalence classes of $\approx_L$:

An Example of $\approx_L$ Where All Elements of L Are Not in the Same Equivalence Class

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

$L = \{w \in \{a, b\}^* :$ no two adjacent characters are the same$\}$

ε	bb	aabaa
a	aba	aabbba
b	aab	aabbaa
aa	baa	
	aabb	

The equivalence classes of $\approx_L$:

Is $|\approx_L|$ Always Finite?

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

$L = a^n b^n$

ε	aa	aaaa
a	aba	aaaaa
b	aaa	

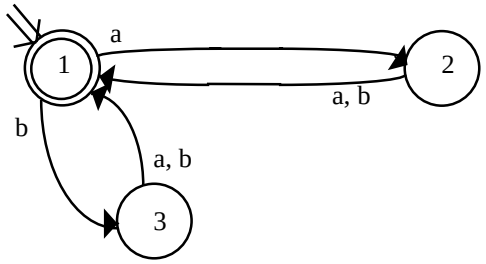
The equivalence classes of $\approx_L$:

Bringing FSMs into the Picture

$\approx_L$ is an ideal relation.

What if we now consider what happens to strings when they are being processed by a real FSM?

$\Sigma = \{a, b\}$ $L = \{w \in \Sigma^* : |w| \text{ is even}\}$



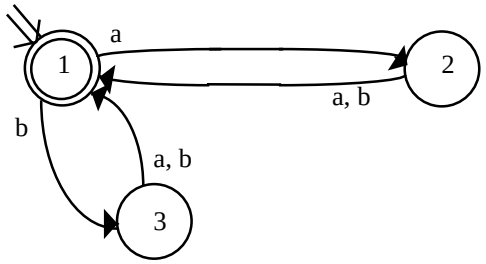
Define $\sim_M$ to relate pairs of strings that drive M from s to the same state.

Formally, if M is a deterministic FSM, then $x \sim_M y$ if there is some state q in M such that $(s, x) \vdash^*_M (q, \varepsilon)$ and $(s, y) \vdash^*_M (q, \varepsilon)$.

Notice that M is an equivalence relation.

An Example of ~M

$\Sigma = \{a, b\}$ $L = \{w \in \Sigma^* : |w| \text{ is even}\}$

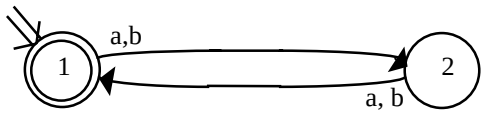


ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\sim_M$: $|\sim_M| =$

Another Example of ~M

$\Sigma = \{a, b\}$ $L = \{w \in \Sigma^* : |w| \text{ is even}\}$



ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\sim_M$: $|\sim_M| =$

The Relationship Between $\approx_L$ and $\sim_M$

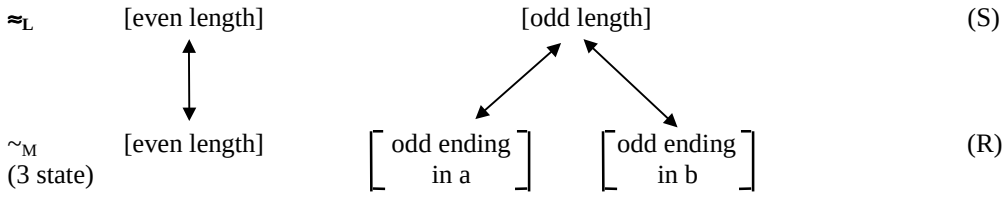
$\approx_L$: [ε, aa, bb, aabb, bbaa] |w| is even
[a, b, aba, aab, bbb, baa, aabaa] |w| is odd

$\sim_M$, 3 state machine:
q1: [ε, aa, bb, aabb, bbaa] |w| is even
q2: [a, aba, baa, aabaa] $(ab \cup ba \cup aa \cup bb)^*a$
q3: [b, aab, bbb] $(ab \cup ba \cup aa \cup bb)^*b$

$\sim_M$, 2 state machine:
q1: [ε, aa, bb, aabb, bbaa] |w| is even
q2: [a, b, aba, aab, bbb, baa, aabaa] |w| is odd

$\sim_M$ is a refinement of $\approx_L$.

The Refinement



An equivalence relation R is a refinement of another one S iff
$xRy \rightarrow xSy$
In other words, R makes all the same distinctions S does, plus possibly more.

$|R| \geq |S|$

$\sim_M$ is a Refinement of $\approx_L$.

Theorem: For any deterministic finite automaton M and any strings x, y $\in \Sigma^*$, if x $\sim_M$ y, then x $\approx_L$ y.

Proof: If x $\sim_M$ y, then x and y drive m to the same state q. From q, any continuation string w will drive M to some state r. Thus xw and yw both drive M to r. Either r is a final state, in which case they both accept, or it is not, in which case they both reject. But this is exactly the definition of $\approx_L$.

Corollary: $|\sim_M| \geq |\approx_L|$.

Going the Other Way

When is this true?

If x $\approx_{L(M)}$ y then x $\sim_M$ y.

Finding the Minimal FSM for L

What's the smallest number of states we can get away with in a machine to accept L?

Example: $L = \{w \in \Sigma^* : |w| \text{ is even}\}$

The equivalence classes of $\approx_L$:

Minimal number of states for M(L) =

This follows directly from the theorem that says that, for any machine M that accepts L, $|\sim_M|$ must be at least as large as $|\approx_L|$.

Can we always find a machine with this minimal number of states?

The Myhill-Nerode Theorem

Theorem: Let L be a regular language. Then there is a deterministic FSA that accepts L and that has precisely $|\approx_L|$ states.
Proof: (by construction)
M = K states, corresponding to the equivalence classes of $\approx_L$.
s = [ε], the equivalence class of ε under $\approx_L$.
F = {[x] : x $\in$ L}
δ([x], a) = [xa]
For this construction to prove the theorem, we must show:

1. K is finite.
2. δ is well defined, i.e., δ([x], a) = [xa] is independent of x.
3. L = L(M)

The Proof

(1) K is finite.
Since L is regular, there must exist a machine M, with $|\sim_M|$ finite. We know that

$$|\sim_M| \geq |\approx_L|$$

Thus $|\approx_L|$ is finite.
(2) δ is well defined.
This is assured by the definition of $\approx_L$, which groups together precisely those strings that have the same fate with respect to L.

The Proof, Continued

(3) L = L(M)
Suppose we knew that $([x], y) \vdash_M^* ([xy], \varepsilon)$.
Now let [x] be [ε] and let s be a string in Σ^*.
Then

$$([\varepsilon], s) \vdash_M^* ([s], \varepsilon)$$

M will accept s if $[s] \in F$.
By the definition of F, $[s] \in F$ iff all strings in [s] are in L.
So M accepts precisely the strings in L.

The Proof, Continued

Lemma: $([x], y) \vdash_M^* ([xy], \varepsilon)$
By induction on |y|:
Trivial if |y| = 0.
Suppose true for |y| = n.
Show true for |y| = n+1

Let y = y'a, for some character a. Then,
|y'| = n

$([x], y'a) \vdash_M^* ([xy'], a)$	(induction hypothesis)
$([xy',] a) \vdash_M^* ([xy'a], \varepsilon)$	(definition of δ)
$([x], y'a) \vdash_M^* ([xy'a], \varepsilon)$	(trans. of $\vdash_M^*$)
$([x], y) \vdash_M^* ([xy], \varepsilon)$	(definition of y)

Another Version of the Myhill-Nerode Theorem

Theorem: A language is regular iff $|\approx_L|$ is finite.
Example:
Consider: $L = a^n b^n$
a, aa, aaa, aaaa, aaaaa …
Equivalence classes:
Proof:
Regular $\rightarrow |\approx_L|$ *is finite*: If L is regular, then there exists an accepting machine M with a finite number of states N. We know that $N \geq |\approx_L|$. Thus $|\approx_L|$ is finite.

$|\approx_L|$ *is finite* $\rightarrow$ *regular*: If $|\approx_L|$ is finite, then the standard DFSA M_L accepts L. Since L is accepted by a FSA, it is regular.

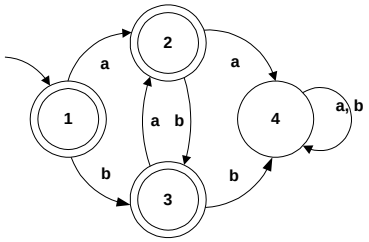
Constructing the Minimal DFA from $\approx_L$

Σ = {a, b}
L = {w ∈ {a, b}* : no two adjacent characters are the same}

The equivalence classes of $\approx_L$:

1: [ε]	ε
2: [a, ba, aba, baba, ababa, ...]	(b∪ε)(ab)*a
3: [b, ab, bab, abab, ...]	(a∪ε)(ba)*b
4: [bb, aa, bba, bbb, ...]	the rest

- Equivalence classes become states
- Start state is [ε]
- Final states are all equivalence classes in L
- δ([x], a) = [xa]



Using Myhill-Nerode to Prove that L is not Regular

L = {a^n : n is prime}

Consider: ε
a
aa
aaa
aaaa

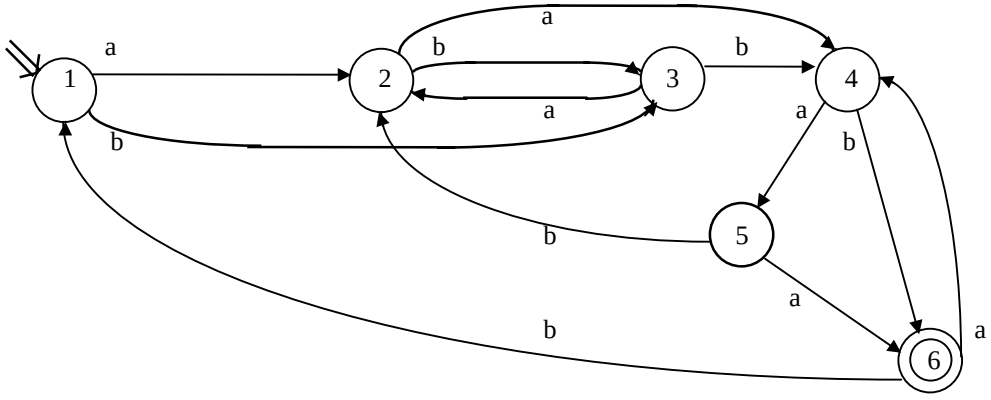
Equivalence classes:

So Where Do We Stand?

1. We know that for any regular language L there exists a minimal accepting machine M_L.
2. We know that |K| of M_L equals $|\approx_L|$.
3. We know how to construct M_L from $\approx_L$.

But is this good enough?

Consider:



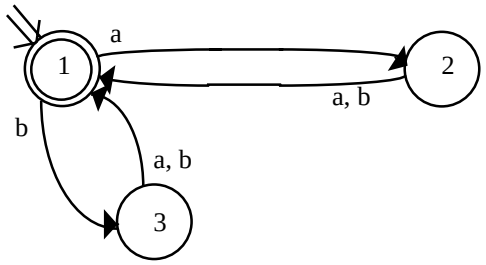
Constructing a Minimal FSA Without Knowing $\approx_L$

We want to take as input any DFSA M' that accepts L, and output a minimal, equivalent DFSA M.

What we need is a definition for "equivalent", i.e., mergeable states.

Define $q \equiv p$ iff for all strings $w \in \Sigma^*$, either w drives M to an accepting state from both q and p or it drives M to a rejecting state from both q and p.

Example:
$\Sigma = \{a, b\}$ $L = \{w \in \Sigma^* : |w| \text{ is even}\}$



Constructing $\equiv$ as the Limit of a Sequence of Approximating Equivalence Relations $\equiv_n$

(Where n is the length of the input strings that have been considered so far)

We'll consider input strings, starting with ε, and increasing in length by 1 at each iteration. We'll start by way overgrouping states. Then we'll split them apart as it becomes apparent (with longer and longer strings) that their behavior is not identical.

Initially, $\equiv_0$ has only two equivalence classes: [F] and [K - F], since on input ε, there are only two possible outcomes, accept or reject.

Next consider strings of length 1, i.e., each element of Σ. Split any equivalence classes of $\equiv_0$ that don't behave identically on all inputs. Note that in all cases, $\equiv_n$ is a refinement of $\equiv_{n-1}$.

Continue, until no splitting occurs, computing $\equiv_n$ from $\equiv_{n-1}$.

Constructing $\equiv$, Continued

More precisely, for any two states p and $q \in K$ and any $n \geq 1$, $q \equiv_n p$ iff:

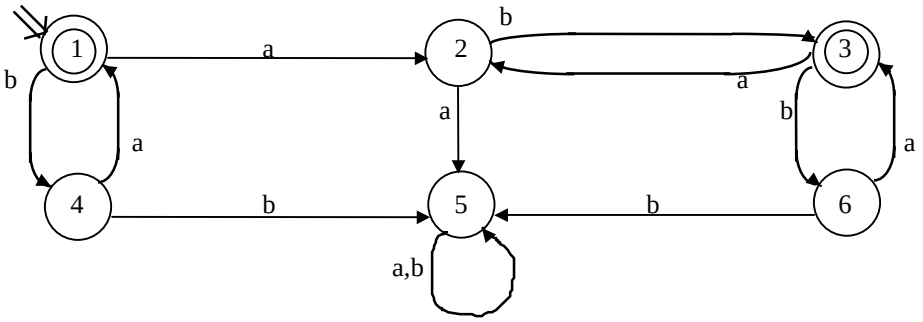
1. $q \equiv_{n-1} p$, AND
2. for all $a \in \Sigma$, $\delta(p, a) \equiv_{n-1} \delta(q, a)$

The Construction Algorithm

The equivalence classes of $\equiv_0$ are F and K-F.
Repeat for n = 1, 2, 3 …
    For each equivalence class C of $\equiv_{n-1}$ do
        For each pair of elements p and q in C do
            For each a in Σ do
                See if $\delta(p, a) \equiv_{n-1} \delta(q, a)$
            If there are any differences in the behavior of p and q, then split them and create a new equivalence class.
Until $\equiv_n = \equiv_{n-1}$. $\equiv$ is this answer. Then use these equivalence classes to coalesce states.

An Example

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



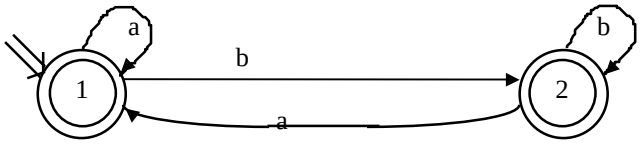
$\equiv_0 =$

$\equiv_1 =$

$\equiv_2 =$

Another Example

(a*b*)*



$\equiv_0 =$

$\equiv_1 =$

Minimal machine:

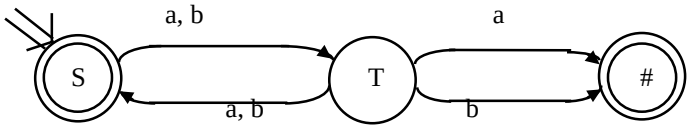
Another Example

Example: $L=\{w \in \{a, b\}^* : |w| \text{ is even}\}$

$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$

$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$



Convert to deterministic:
S = {s}
δ =

Another Example, Continued

Minimize:

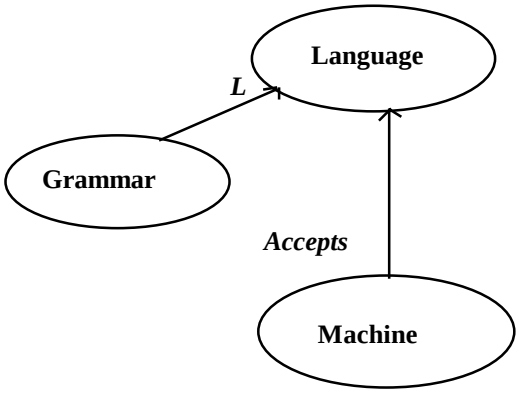
$\equiv_0$ =

$\equiv_1$ =

Minimal machine:

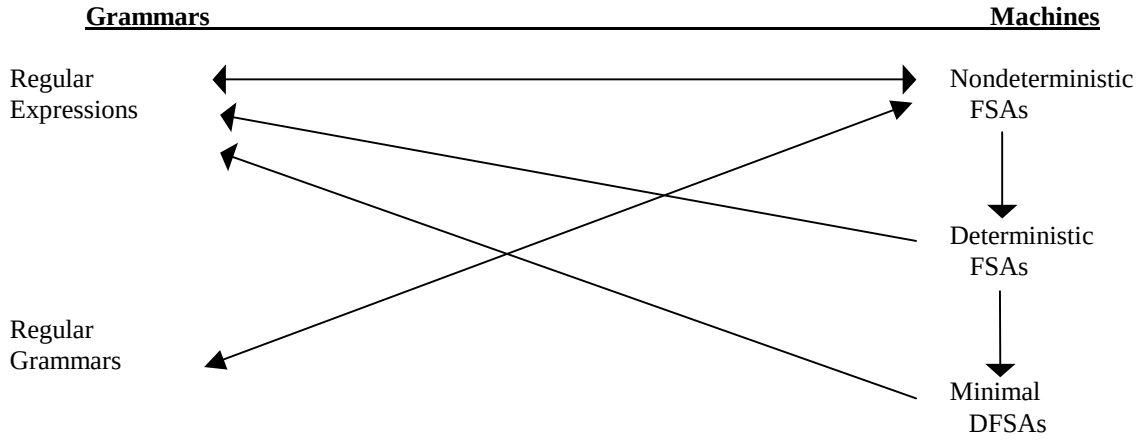
Summary of Regular Languages and Finite State Machines

Grammars, Languages, and Machines



Regular Grammars, Languages, and Machines

Most interesting languages are infinite. So we can't write them down. But we can write down finite grammars and finite machine specifications, and we can define algorithms for mapping between and among them.



What Does "Finite State" Really Mean?

There are two kinds of finite state problems:

- Those in which:
  - Some history matters.
  - Only a finite amount of history matters. In particular, it's often the case that we don't care what order things occurred in.

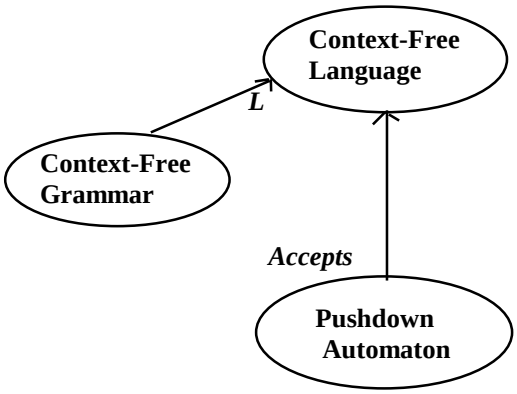
  Examples:
    - Parity
    - Money in a vending machine
    - Seat belt buzzer
- Those that are characterized by patterns.

  Examples:
    - Switching circuits:
      - Telephone
      - Railroad
    - Traffic lights
    - Lexical analysis
    - grep

Context-Free Grammars

Read K & S 3.1
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Context-Free Grammars
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Designing Context-Free Grammars.
Do Homework 11.

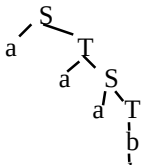
Context-Free Grammars, Languages, and Pushdown Automata



Grammars Define Languages

Think of grammars as either generators or acceptors.

Example: L = {w ∈ {a, b}* : |w| is even}

	Regular Expression	**Regular Grammar**
		$S \rightarrow \varepsilon$
	(aa ∪ ab ∪ ba ∪ bb)*	$S \rightarrow aT$
		$S \rightarrow bT$
		$T \rightarrow a$
		$T \rightarrow b$
		$T \rightarrow aS$
		$T \rightarrow bS$
Derivation (Generate)	choose aa choose ab yields	
	a a a b	*a a a b*
Parse (Accept)	use corresponding FSM	

Derivation is Not Necessarily Unique

Example: L = {w ∈ {a, b}* : there is at least one a}

Regular Expression

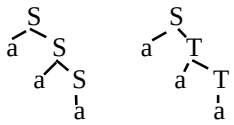
(a ∪ b)*a (a ∪ b)*

choose a from (a ∪ b)
choose a from (a ∪ b)
choose a

choose a
choose a from (a ∪ b)
choose a from (a ∪ b)

Regular Grammar

S → a
S → bS
S → aS
S → aT
T → a
T → b
T → aT
T → bT



More Powerful Grammars

Regular grammars must always produce strings one character at a time, moving left to right.

But sometimes it's more natural to describe generation more flexibly.

Example 1: L = ab*a

S → aBa
B → ε
B → bB

vs.

S → aB
B → a
B → bB

Example 2: L = $a^n b^* a^n$

S → B
S → aSa
B → ε
B → bB

Key distinction: Example 1 has no recursion on the nonregular rule.

Context-Free Grammars

Remove all restrictions on the form of the right hand sides.

S → abDeFGab

Keep requirement for single non-terminal on left hand side.

S →

but not ASB → or aSb → or ab →

Examples:

balanced parentheses

S → ε
S → SS
S → (S)

$a^n b^n$

S → a S b
S → ε

Context-Free Grammars

A context-free grammar G is a quadruple (V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals (symbols that are used in the grammar but that do not appear in strings in the language) and terminals,
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite subset of $(V - \Sigma) \times V^*$,
- S (the start symbol) is an element of V - Σ.

$\mathbf{x \Rightarrow_G y}$ is a binary relation where $x, y \in V^*$ such that $x = \alpha A \beta$ and $y = \alpha\chi\beta$ for some rule $A \rightarrow \chi$ in R.

Any sequence of the form

$$w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \ldots \Rightarrow_G w_n$$

e.g., $(S) \Rightarrow (SS) \Rightarrow ((S)S)$

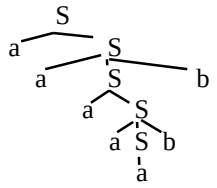
is called a **derivation in G**. Each w_i is called a **sentinel form**.

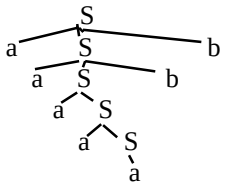
The **language generated by G** is $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$

A **language L is context free** if L = L(G) for some context-free grammar G.

Example Derivations

G = (W, Σ, R, S), where
W = {S} ∪ Σ,
Σ = {a, b},
R = { S → a,
S → aS,
S → aSb}





Another Example - Unequal a's and b's

$L = \{a^n b^m : n \neq m\}$

G = (W, Σ, R, S), where
W = {a, b, S, A, B},
Σ = {a, b},
R =

S → A /* more a's than b's
S → B /* more b's than a's
A → a
A → aA
A → aAb
B → b
B → Bb
B → aBb

English

S → NP VP
NP → the NP1 | NP1
NP1 → ADJ NP1 | N
ADJ → big | youngest | oldest
N → boy | boys
VP →V | V NP
V → run | runs

the boys run
big boys run
the youngest boy runs

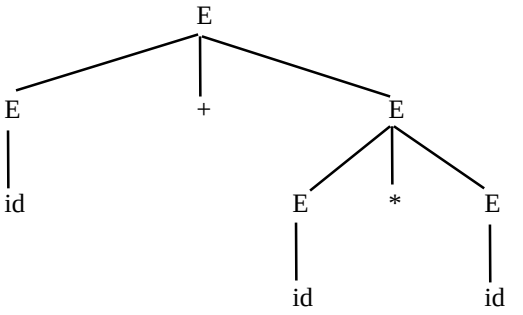
the youngest oldest boy runs
the boy run

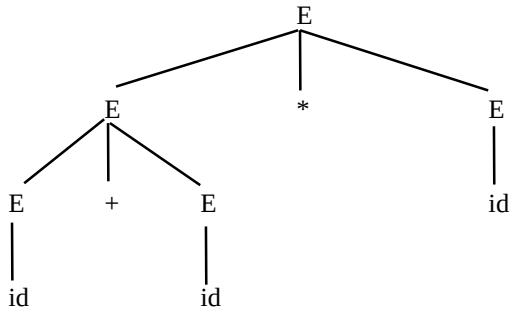
Who did you say Bill saw coming out of the hotel?

Arithmetic Expressions

The Language of Simple Arithmetic Expressions

G = (V, Σ, R, E), where
V = {+, *, id, T, F, E},
Σ = {+, *, id},
R = { E → id
E → E + E
E → E * E }





id + (id * id)

(id + id) * id

Arithmetic Expressions -- A Better Way

The Language of Simple Arithmetic Expressions

G = (V, Σ, R, E), where
V = {+, *, (,), id, T, F, E},
Σ = {+, *, (,), id},
R = { E → E + T
E→ T
T → T * F
T → F
F → (E)
F → id }

Examples:

id + id * id

id * id * id

BNF

Backus-Naur Form (BNF) is used to define the syntax of programming languages using context-free grammars.

Main idea: give descriptive names to nonterminals and put them in angle brackets.

Example: arithmetic expressions:

⟨expression⟩ → ⟨expression⟩ + ⟨term⟩
⟨expression⟩ → ⟨term⟩
⟨term⟩ → ⟨term⟩ * ⟨factor⟩
⟨term⟩ → ⟨factor⟩
⟨factor⟩ → (⟨expression⟩)
⟨factor⟩ → ⟨id⟩

The Language of Boolean Logic

G = (V, Σ, R, E), where

V = {∧, ∨, ¬,⇒ , (,), id, E, E1, E2, E3, E4 },
Σ = {∧, ∨, ¬, ⇒, (,), id},
R = { E → E ⇒ E1
E → E1
E1 → E1 ∨ E2
E1 →E2
E2 → E2 ∧ E3
E2 → E3
E3 → ¬ E4
E3 → E4
E4 →(E)
E4 → id }

Boolean Logic isn't Regular

Suppose it were regular. Then there is an N as specified in the pumping theorem.

Let w be a string of length 2N + 1 + 2|id| of the form:

w = $\underbrace{(\,(\,(\,(\,(\,(}_{N}$ id)))))) ⇒ id

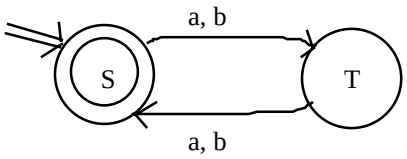
x y

$y = (^k$ for some k > 0 because |xy| ≤ N.

Then the string that is identical to w except that it has k additional ('s at the beginning would also be in the language. But it can't be because the parentheses would be mismatched. So the language is not regular.

All Regular Languages Are Context Free

(1) Every regular language can be described by a regular grammar. We know this because we can derive a regular grammar from any FSM (as well as vice versa). Regular grammars are special cases of context-free grammars.



(2) The context-free languages are precisely the languages accepted by NDPDAs. But every FSM is a PDA that doesn't bother with the stack. So every regular language can be accepted by a NDPDA and is thus context-free.

(3) Context-free languages are closed under union, concatenation, and Kleene *, and ε and each single character in Σ are clearly context free.

Parse Trees

Read K & S 3.2
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Derivations and Parse Trees.
Do Homework 12.

Parse Trees

Regular languages:

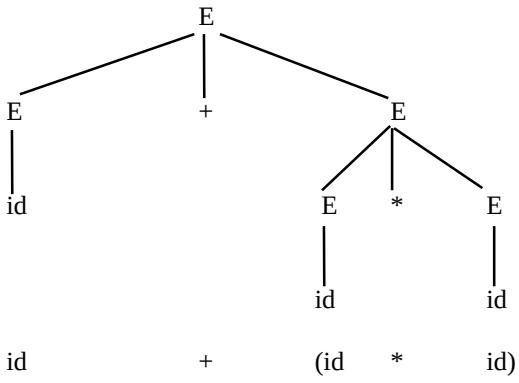
We care about recognizing patterns and taking appropriate actions.

Example: A parity checker

Structure

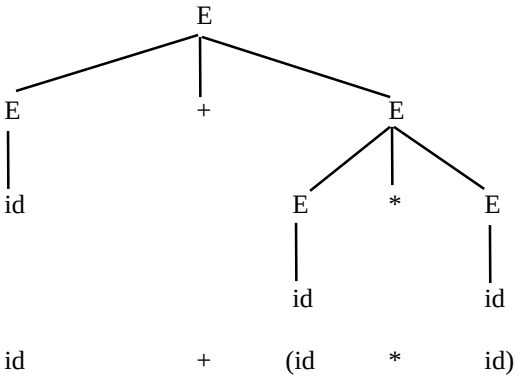
Context free languages:

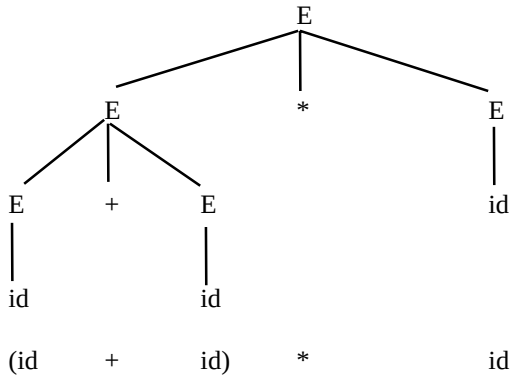
We care about structure.



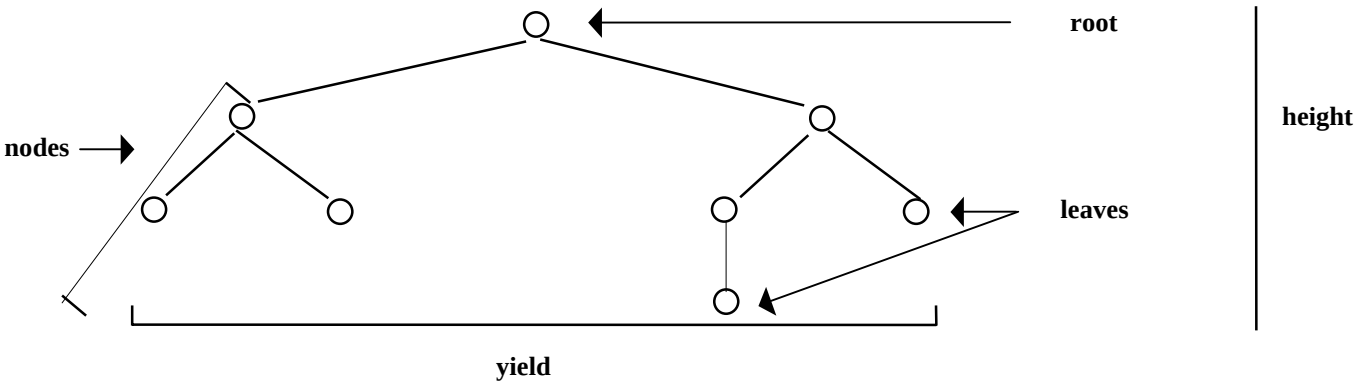
Parse Trees Capture Essential Structure

$E \rightarrow id$
$E \rightarrow E + E$
$E \rightarrow E * E$





Parse Trees are Just Trees



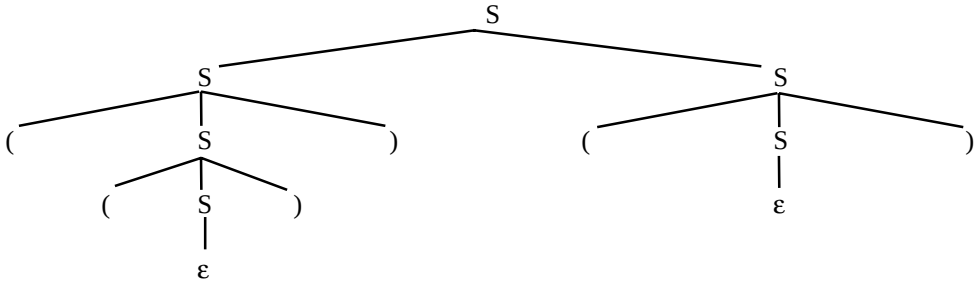
Leaves are all labeled with terminals or ε.
Other nodes are labeled with nonterminals.
A **path** is a sequence of nodes, starting at the root, ending at a leaf, and following branches in the tree.
The length of the yield of any tree T with height H and branching factor (fanout) B is ≤

Derivations

To capture structure, we must capture the path we took through the grammar. **Derivations** do that.

$S \rightarrow \varepsilon$
$S \rightarrow SS$
$S \rightarrow (S)$

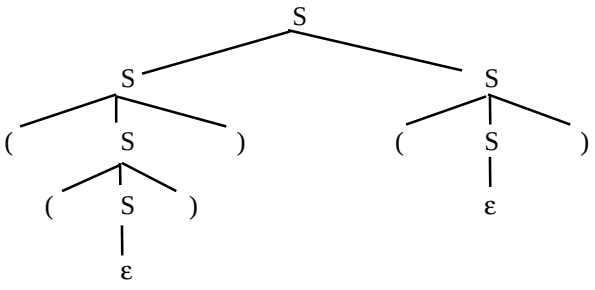
1 2 3 4 5 6
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$
1 2 3 5 4 6

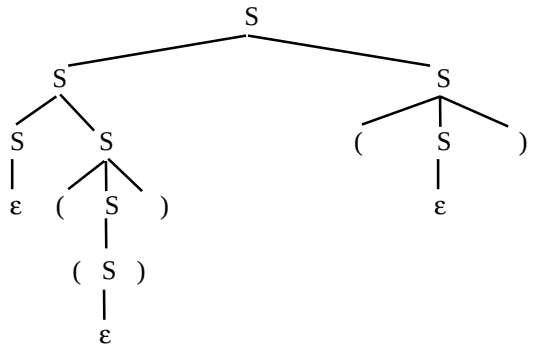


Alternative Derivations

$S \rightarrow \varepsilon$
$S \rightarrow SS$
$S \rightarrow (S)$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$
$S \Rightarrow SS \Rightarrow SSS \Rightarrow S(S)S \Rightarrow S((S))S \Rightarrow S(())S \Rightarrow S(())(S) \Rightarrow S(())() \Rightarrow (())()$





Ordering Derivations

Consider two derivations:

1 2 3 4 5 6 7
S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ (())S ⇒ (())(S) ⇒ (())()

S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ ((S))(S) ⇒ (())(S) ⇒ (())()
1 2 3 4 5 6 7

We can write these, or any, derivation as
$D_1 = x_1 \Rightarrow x_2 \Rightarrow x_3 \Rightarrow \ldots \Rightarrow x_n$
$D_2 = x_1' \Rightarrow x_2' \Rightarrow x_3' \Rightarrow \ldots \Rightarrow x_n'$

We say that D_1 precedes D_2, written $D_1 < D_2$, if:

- D_1 and D_2 are the same length > 1, and
- There is some integer k, $1 < k < n$, such that:
  - for all $i \neq k$, $x_i = x_i'$
  - $x_{k-1} = x'_{k-1} = uAvBw$: u, v, w ∈ V*, and A, B ∈ V - Σ
  - $x_k = uyvBw$, where $A \rightarrow y \in R$
  - $x_k' = uAvzw$ where $B \rightarrow z \in R$
  - $x_{k+1} = x'_{k+1} = uyvzw$

Comparing Several Derivations

Consider three derivations:

1 2 3 4 5 6 7
(1) S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ (())S ⇒(())(S) ⇒(())()

(2) S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ ((S))(S) ⇒ (())(S) ⇒(())()

(3) S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ ((S))(S) ⇒ ((S))() ⇒(())()

D1 < D2
D2 < D3
But D1 does not precede D3.
All three seem similar though. We can define similarity:

D_1 is **similar** to D_2 iff the pair (D_1, D_2) is in the reflexive, symmetric, transitive closure of <.

Note: similar is an equivalence class.

In other words, two derivations are similar if one can be transformed into another by a sequence of switchings in the order of rule applications.

Parse Trees Capture Similarity

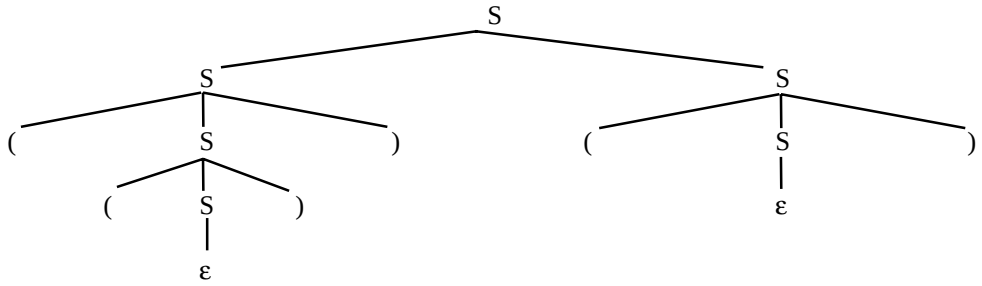
1 2 3 4 5 6 7
(1) S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ (())S ⇒(())(S) ⇒(())()

(2) S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ ((S))(S) ⇒(())(S) ⇒(())()

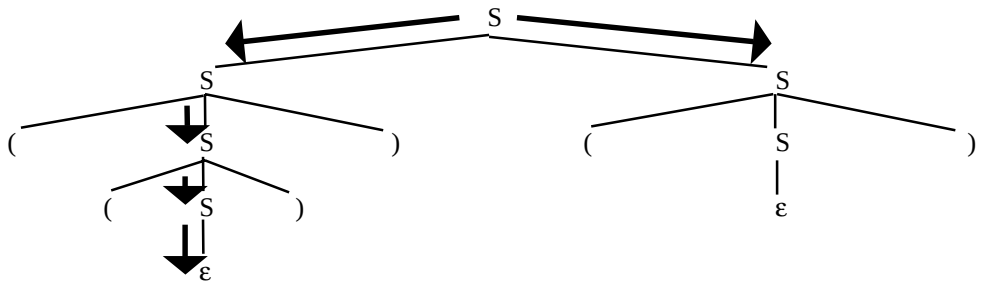
(3) S ⇒ SS ⇒ (S)S ⇒ ((S))S ⇒ ((S))(S) ⇒((S))() ⇒(())()

D1 < D2
D2 < D3

All three derivations are similar to each other. This parse tree describes this equivalence class of the similarity relation:



The Maximal Element of <



There's one derivation in this equivalence class that precedes all others in the class.

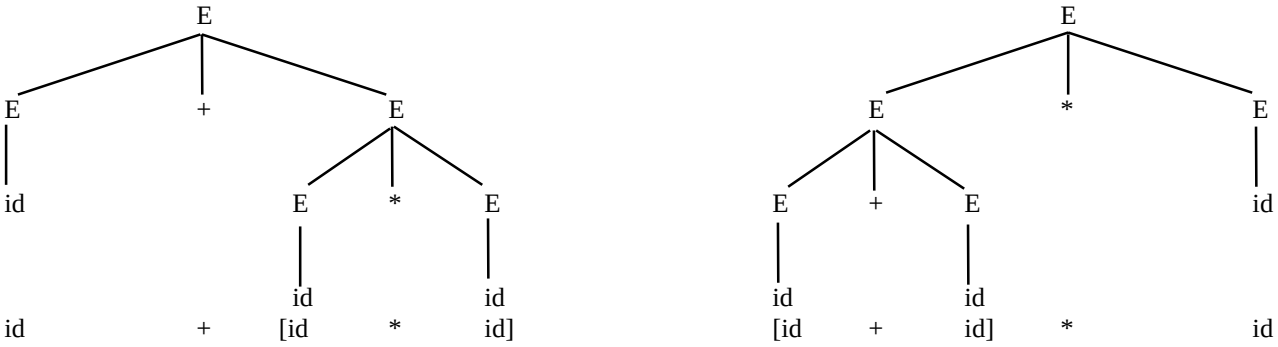
We call this the **leftmost derivation**. There is a corresponding rightmost derivation.

The leftmost (rightmost) derivation can be used to construct the parse tree and the parse tree can be used to construct the leftmost (rightmost) derivation.

Another Example

$E \rightarrow id$
$E \rightarrow E + E$
$E \rightarrow E * E$

(1) $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$
(2) $E \Rightarrow E*E \Rightarrow E*id \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$



Ambiguity

A grammar G for a language L is **ambiguous** if there exist strings in L for which G can generate more than one parse tree (note that we don't care about the number of derivations).

The following grammar for arithmetic expressions is ambiguous:

$E \rightarrow id$
$E \rightarrow E + E$
$E \rightarrow E * E$

Often, when this happens, we can find a different, unambiguous grammar to describe L.

Resolving Ambiguity in the Grammar

G = (V, Σ, R, E), where

V = {+, *, (,), id, T, F, E},

Σ = {+, *, (,), id},

R = { $E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$ }

Parse : id + id * id

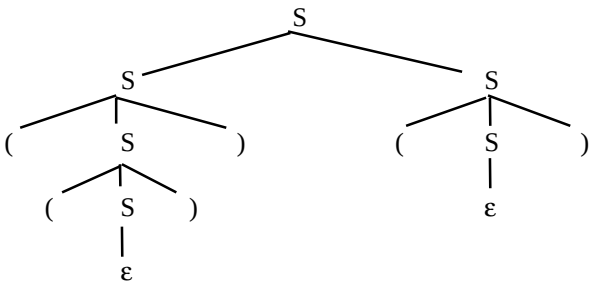
Another Example

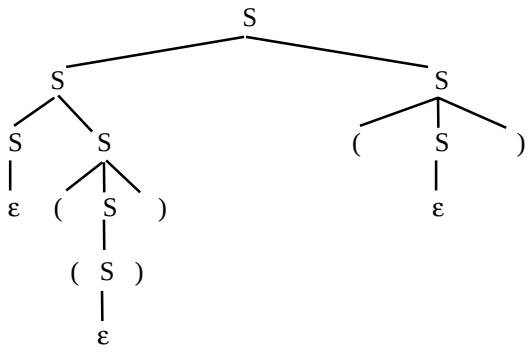
The following grammar for the language of matched parentheses is ambiguous:

$S \rightarrow \varepsilon$

$S \rightarrow SS$

$S \rightarrow (S)$





Resolving the Ambiguity with a Different Grammar

One problem is the ε production.

A different grammar for the language of balanced parentheses:

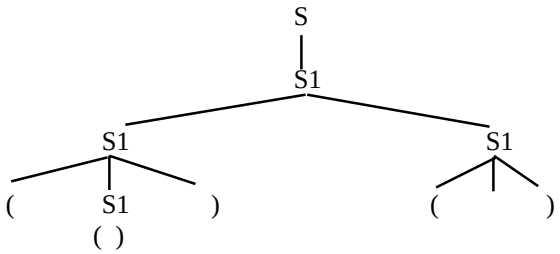
$S \rightarrow \varepsilon$

$S \rightarrow S_1$

$S_1 \rightarrow S_1 S_1$

$S_1 \rightarrow (S_1)$

$S_1 \rightarrow ()$



A General Technique for Eliminating ε

If G is any context-free grammar for a language L and $\varepsilon \notin L$ then we can construct an alternative grammar G' for L by:

1. Find the set N of nullable variables:
   A variable V is **nullable** if either:
   there is a rule
   (1) $V \rightarrow \varepsilon$
   or there is a rule
   (2) $V \rightarrow PQR\ldots$ such that P, Q, R, … are all nullable
   So begin with N containing all the variables that satisfy (1). Evaluate all other variables with respect to (2). Continue until no new variables can be added to N.
2. For every rule of the form
   $P \rightarrow \alpha Q \beta$ for some Q in N, add a rule
   $P \rightarrow \alpha\beta$
3. Delete all rules of the form
   $V \rightarrow \varepsilon$

Sometimes Eliminating Ambiguity Isn't Possible

S → NP VP
NP → the NP1 | NP1 | NP2
NP1 → ADJ NP1 | N
NP2 → NP1 PP
ADJ → big | youngest | oldest
N → boy | boys | ball | bat | autograph
VP →V | V NP
VP → VP PP
V → hit| hits
PP → with NP

The boys hit the ball with the bat.

The boys hit the ball with the autograph.

Why It's Not Possible

- We could write an unambiguous grammar to describe L but it wouldn't always get the parses we want. Any grammar that is capable of getting all the parses will be ambiguous because the facts required to choose a derivation cannot be captured in the context-free framework.
  Example: Our simple English grammar
  [[The boys] [hit [the ball] [with [the bat]]]]
  [[The boys] [hit [the ball] [with [the autograph]]]]
- There is no grammar that describes L that is not ambiguous.
  Example: $L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$

$S \rightarrow S_1 \mid S_2$
$S_1 \rightarrow S_1 c \mid A$
$A \rightarrow aAb \mid \varepsilon$
$S_2 \rightarrow aS_2 \mid B$
$B \rightarrow bBc \mid \varepsilon$

Now consider the strings $a^n b^n c^n$

They have two distinct derivations

Inherent Ambiguity of CFLs

A context free language with the property that all grammars that generate it are ambiguous is **inherently ambiguous**.

$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$ is inherently ambiguous.

Other languages that appear ambiguous given one grammar, turn out not to be inherently ambiguous because we can find an unambiguous grammar.

Examples: Arithmetic Expressions
Balanced Parentheses

Whenever we design practical languages, it is important that they not be inherently ambiguous.

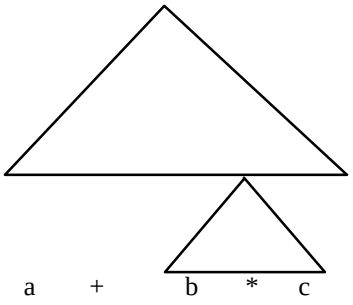
Pushdown Automata

Read K & S 3.3.
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Designing Pushdown Automata.
Do Homework 13.

Recognizing Context-Free Languages

Two notions of recognition:

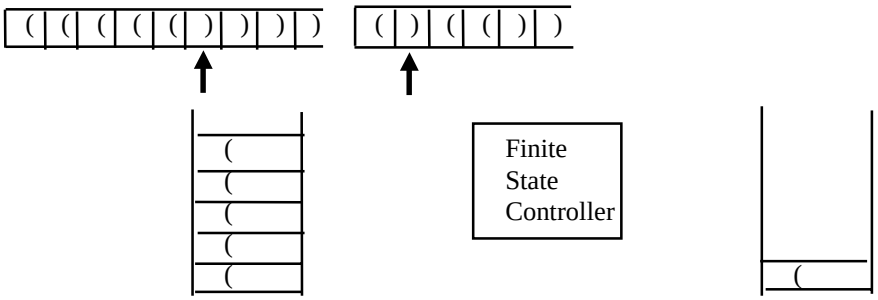
(1) Say yes or no, just like with FSMs
(2) Say yes or no, AND
if yes, describe the structure



Just Recognizing

We need a device similar to an FSM except that it needs more power.

The insight: Precisely what it needs is a stack, which gives it an unlimited amount of memory with a restricted structure.



Definition of a Pushdown Automaton

M = (K, Σ, Γ, Δ, s, F), where:

K is a finite set of states
Σ is the input alphabet
Γ is the stack alphabet
$s \in K$ is the initial state
$F \subseteq K$ is the set of final states, and
Δ is the transition relation. It is a finite subset of

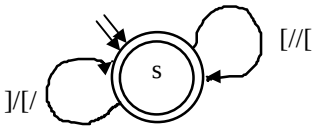
$$(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \times K \times \Gamma^*)$$

(state, input or ε, string of symbols to pop from top of stack) × (state, string of symbols to push on top of stack)

M accepts a string w iff

$(s, w, \varepsilon) \vdash_M^* (p, \varepsilon, \varepsilon)$ for some state $p \in F$

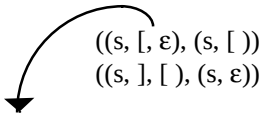
A PDA for Balanced Brackets



M = (K, Σ, Γ, Δ, s, F), where:

K = {s}	the states
Σ = {[,]}	the input alphabet
Γ = {[}	the stack alphabet
F = {s}	

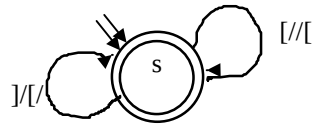
Δ contains:



Important:

This does not mean that the stack is empty.

An Example of Accepting



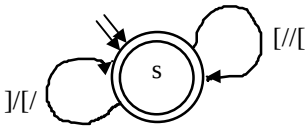
Δ contains:

[1] ((s, [, ε), (s, [ ))
[2] ((s,], [ ), (s, ε))

input = [[[ ] [ ]]]

trans	*state*	*unread input*	*stack*
	s	[[[ ] [ ]]]	ε
1	s	[[ ] [ ]]]	[
1	s	[ ] [ ]]]	[[
1	s	] [ ]]]	[[[
2	s	[ ]]]	[[
1	s	]]]	[[[
2	s	]]	[[
2	s	]	[
2	s	ε	ε

An Example of Rejecting



Δ contains:

[1] ((s, [, ε), (s, [ ))
[2] ((s,], [ ), (s, ε))

input = [[ ]]]

trans	*state*	*unread input*	*stack*
	s	[[ ]]]	ε
1	s	[ ]]]	[
1	s	]]]	[[
2	s	]]	[
2	s	]	ε
none!	s	]	ε

We're in s, a final state, but we cannot accept because the input string is not empty. So we reject.

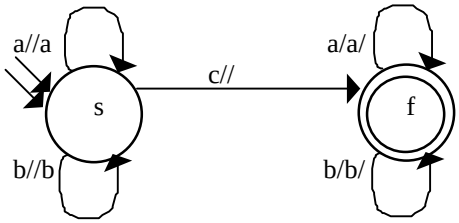
A PDA for $a^n b^n$

First we notice:

- We'll use the stack to count the a's.
- This time, all strings in L have two regions. So we need two states so that a's can't follow b's. Note the similarity to the regular language a*b*.

A PDA for wcw^R

A PDA to accept strings of the form wcw^R:



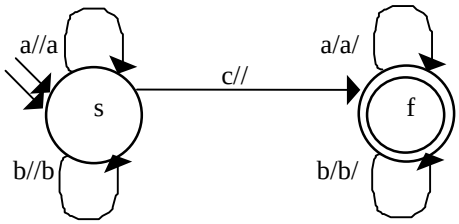
M = (K, Σ, Γ, Δ, s, F), where:

K = {s, f}	the states
Σ = {a, b, c}	the input alphabet
Γ = {a, b}	the stack alphabet
F = {f}	the final states

Δ contains:

((s, a, ε), (s, a))
((s, b, ε), (s, b))
((s, c, ε), (f, ε))
((f, a, a), (f, ε))
((f, b, b), (f, ε))

An Example of Accepting



Δ contains:

[1] ((s, a, ε), (s, a))
[2] ((s, b, ε), (s, b))
[3] ((s, c, ε), (f, ε))
[4] ((f, a, a), (f, ε))
[5] ((f, b, b), (f, ε))

input = b a c a b

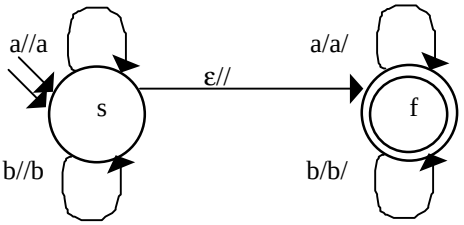
trans	*state*	*unread input*	*stack*
	s	b a c a b	ε
2	s	a c a b	b
1	s	c a b	ab
3	f	a b	ab
5	f	b	b
6	f	ε	ε

A Nondeterministic PDA

$L = ww^R$

$S \rightarrow \varepsilon$
$S \rightarrow aSa$
$S \rightarrow bSb$

A PDA to accept strings of the form ww^R:



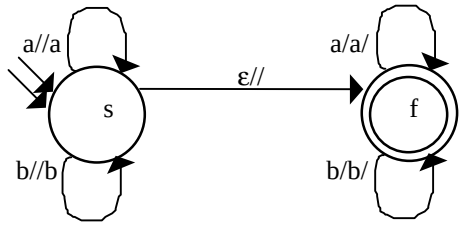
M = (K, Σ, Γ, Δ, s, F), where:

K = {s, f}	the states
Σ = {a, b, c}	the input alphabet
Γ = {a, b}	the stack alphabet
F = {f}	the final states

Δ contains:

((s, a, ε), (s, a))
((s, b, ε), (s, b))
((s, ε, ε), (f, ε))
((f, a, a), (f, ε))
((f, b, b), (f, ε))

An Example of Accepting



[1]	((s, a, ε), (s, a))	[4]	((f, a, a), (f, ε))
[2]	((s, b, ε), (s, b))	[5]	((f, b, b), (f, ε))
[3]	((s, ε, ε), (f, ε))		

input: a a b b a a

trans	***state***	***unread input***	***stack***
	s	a a b b a a	ε
1	s	a b b a a	a
3	f	a b b a a	a
4	f	b b a a	ε

none

trans	***state***	***unread input***	***stack***
	s	a a b b a a	ε
1	s	a b b a a	a
1	s	b b a a	aa
2	s	b a a	baa
3	f	b a a	baa
5	f	a a	aa
4	f	a	a
4	f	ε	ε

$L = \{a^m b^n : m \leq n\}$

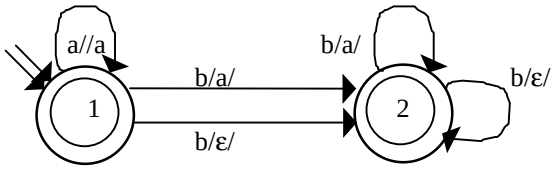
A context-free grammar for L:

$S \rightarrow \varepsilon$

$S \rightarrow Sb$ /* more b's

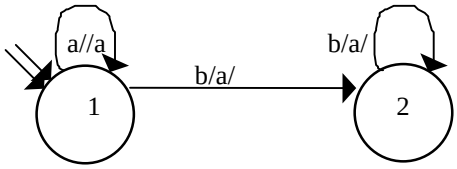
$S \rightarrow aSb$

A PDA to accept L:

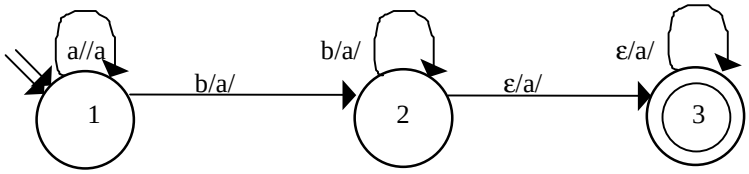


Accepting Mismatches

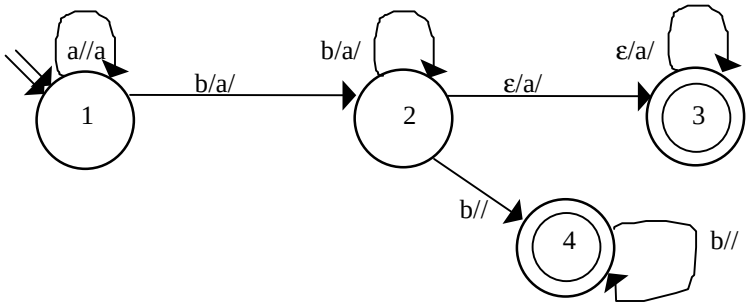
$L = \{a^m b^n \; m \neq n; \; m, n > 0\}$



- If stack and input are empty, halt and reject.
- If input is empty but stack is not (m > n) (accept):

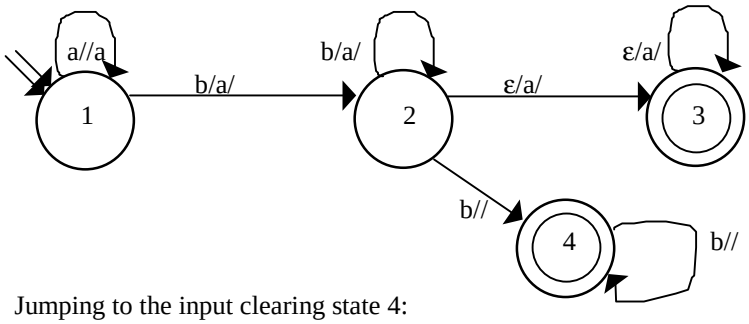


- If stack is empty but input is not (m < n) (accept):

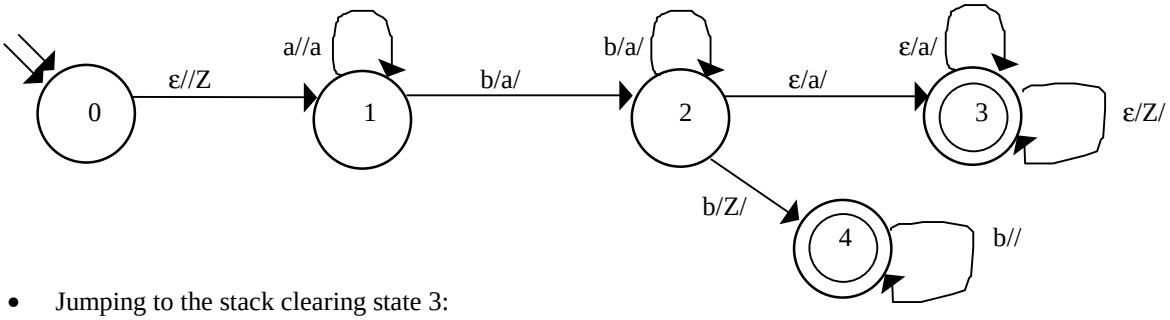


Eliminating Nondeterminism

A PDA is **deterministic** if, for each input and state, there is at most one possible transition. Determinism implies uniquely defined machine behavior.



- Jumping to the input clearing state 4:
  Need to detect bottom of stack, so push Z onto the stack before we start.



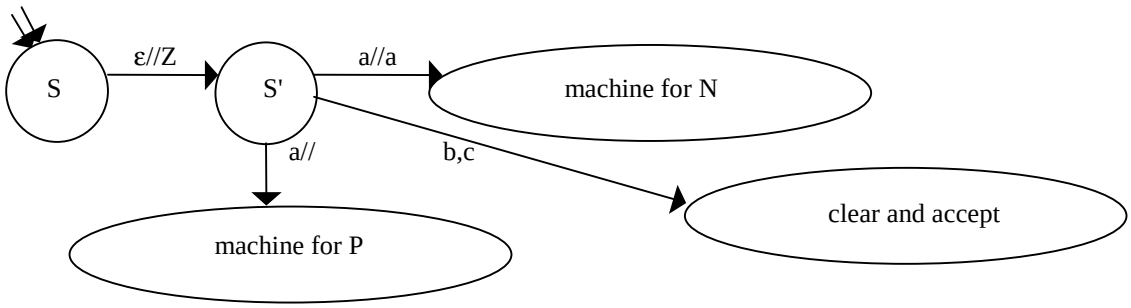
- Jumping to the stack clearing state 3:
  Need to detect end of input. To do that, we actually need to modify the definition of L to add a termination character (e.g., $)

$L = \{a^n b^m c^p : n,m,p \geq 0 \text{ and } (n \neq m \text{ or } m \neq p)\}$

$S \rightarrow NC$	/* $n \neq m$, then arbitrary c's
$S \rightarrow QP$	/* arbitrary a's, then $p \neq m$
$N \rightarrow A$	/* more a's than b's
$N \rightarrow B$	/* more b's than a's
$A \rightarrow a$	
$A \rightarrow aA$	
$A \rightarrow aAb$	
$B \rightarrow b$	
$B \rightarrow Bb$	
$B \rightarrow aBb$	

$C \rightarrow \varepsilon \mid cC$	/* add any number of c's
$P \rightarrow B'$	/* more b's than c's
$P \rightarrow C'$	/* more c's than b's
$B' \rightarrow b$	
$B' \rightarrow bB'$	
$B' \rightarrow bB'c$	
$C' \rightarrow c \mid C'c$	
$C' \rightarrow C'c$	
$C' \rightarrow bC'c$	
$Q \rightarrow \varepsilon \mid aQ$	/* prefix with any number of a's

$L = \{a^n b^m c^p : n,m,p \geq 0 \text{ and } (n \neq m \text{ or } m \neq p)\}$



Another Deterministic CFL

$L = \{a^n b^n\} \cup \{b^n a^n\}$

A CFG for L:

A NDPDA for L:

$S \rightarrow A$
$S \rightarrow B$
$A \rightarrow \varepsilon$
$A \rightarrow aAb$
$B \rightarrow \varepsilon$
$B \rightarrow bBa$

A DPDA for L:

More on PDAs

What about a PDA to accept strings of the form ww?

Every FSM is (Trivially) a PDA

Given an FSM M = (K, Σ, Δ, s, F)
and elements of Δ of the form
(p, i, q)
old state, input, new state

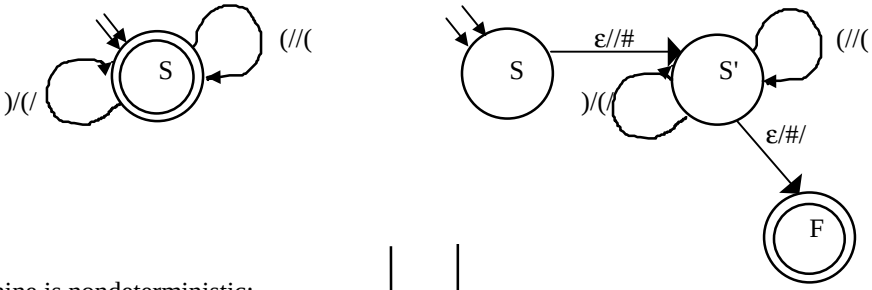
We construct a PDA M' = (K, Σ, Γ, Δ, s, F)
where Γ = ∅ /* stack alphabet
and
each transition (p, i, q) becomes
((p, i, ε), (q, ε))
old state, input, don't look at stack new state don't push on stack

In other words, we just don't use the stack.

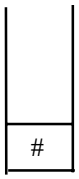
Alternative (but Equivalent) Definitions of a NDPDA

Example: Accept by final state at end of string (i.e., we don't care about the stack being empty)
We can easily convert from one of our machines to one of these:
1. Add a new state at the beginning that pushes # onto the stack.
2. Add a new final state and a transition to it that can be taken if the input string is empty and the top of the stack is #.

Converting the balanced parentheses machine:



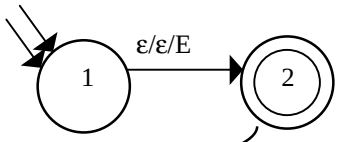
The new machine is nondeterministic:
( ) ( )
⇑
The stack will be:



What About PDA's for Interesting Languages?

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

Arithmetic Expressions



(1) (2, ε, E), (2, E+T)
(2) (2, ε, E), (2, T)
(3) (2, ε, T), (2, T*F)
(4) (2, ε, T), (2, F)
(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, id, id), (2, ε)
(8) (2, (, ( ), (2, ε)
(9) (2,),)), (2, ε)
(10) (2, +, +), (2, ε)
(11) (2, *, *), (2, ε)

Example:

a + b * c

But what we really want to do with languages like this is to extract structure.

Comparing Regular and Context-Free Languages

Regular Languages

- regular expressions
  - or -
- regular grammars
- recognize
- = DFSAs

Context-Free Languages

- context-free grammars
- parse
- = NDPDAs

Pushdown Automata and Context-Free Grammars

Read K & S 3.4.
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Context-Free Languages and PDAs.
Do Homework 14.

PDAs and Context-Free Grammars

Theorem: The class of languages accepted by PDAs is exactly the class of context-free languages.

Recall: context-free languages are languages that can be defined with context-free grammars.

Restate theorem: Can describe with context-free grammar ⇔ Can accept by PDA

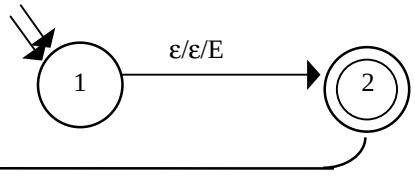
Going One Way

Lemma: Each context-free language is accepted by some PDA.
Proof (by construction by "top-down parse" conversion algorithm):

The idea: Let the stack do the work.

Example: Arithmetic expressions

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$



(1) (2, ε, E), (2, E+T)
(2) (2, ε, E), (2, T)
(3) (2, ε, T), (2, T*F)
(4) (2, ε, T), (2, F)
(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, id, id), (2, ε)
(8) (2, (, ( ), (2, ε)
(9) (2,),)), (2, ε)
(10) (2, +, +), (2, ε)
(11) (2, *, *), (2, ε)

The Top-down Parse Conversion Algorithm

Given G = (V, Σ, R, S)
Construct M such that L(M) = L(G)

M = ({p, q}, Σ, V, Δ, p, {q}), where Δ contains:

(1) ((p, ε, ε), (q, S))
push the start symbol on the stack

(2) ((q, ε, A), (q, x)) for each rule $A \rightarrow x$ in R
replace left hand side with right hand side

(3) ((q, a, a), (q, ε)) for each $a \in \Sigma$
read an input character and pop it from the stack

The resulting machine can execute a leftmost derivation of an input string in a top-down fashion.

Example of the Algorithm

$L = \{a^n b^* a^n\}$

(1) $S \rightarrow \varepsilon$
(2) $S \rightarrow B$
(3) $S \rightarrow aSa$
(4) $B \rightarrow \varepsilon$
(5) $B \rightarrow bB$

input = a a b b a a

0	(p, ε, ε), (q, S)
1	(q, ε, S), (q, ε)
2	(q, ε, S), (q, B)
3	(q, ε, S), (q, aSa)
4	(q, ε, B), (q, ε)
5	(q, ε, B), (q, bB)
6	(q, a, a), (q, ε)
7	(q, b, b), (q, ε)

trans	*state*	*unread input*	*stack*
	p	a a b b a a	ε
0	q	a a b b a a	S
3	q	a a b b a a	aSa
6	q	a b b a a	Sa
3	q	a b b a a	aSaa
6	q	b b a a	Saa
2	q	b b a a	Baa
5	q	b b a a	bBaa
7	q	b a a	Baa
5	q	b a a	bBaa
7	q	a a	Baa
4	q	a a	aa
6	q	a	a
6	q	ε	ε

Another Example

$L = \{a^n b^m c^p d^q : m + n = p + q\}$

(1) $S \rightarrow aSd$
(2) $S \rightarrow T$
(3) $S \rightarrow U$
(4) $T \rightarrow aTc$
(5) $T \rightarrow V$
(6) $U \rightarrow bUd$
(7) $U \rightarrow V$
(8) $V \rightarrow bVc$
(9) $V \rightarrow \varepsilon$

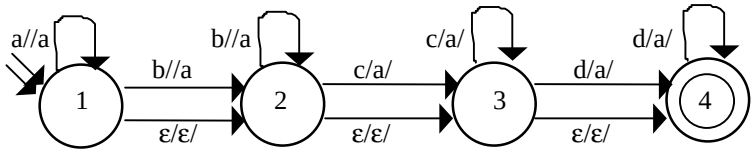
input = a a b c d d

0	(p, ε, ε), (q, S)
1	(q, ε, S), (q, aSd)
2	(q, ε, S), (q,T)
3	(q, ε, S), (q,U)
4	(q, ε, T), (q, aTc)
5	(q, ε, T), (q, V)
6	(q, ε, U), (q, bUd)
7	(q, ε, U), (q, V)
8	(q, ε, V), (q, bVc
9	(q, ε, V), (q, ε)
10	(q, a, a), (q, ε)
11	(q, b, b), (q, ε)
12	(q, c, c), (q, ε)
13	(q, d, d), (q, ε)

The Other Way—Build a PDA Directly

$L = \{a^n b^m c^p d^q : m + n = p + q\}$

(1) $S \rightarrow aSd$
(2) $S \rightarrow T$
(3) $S \rightarrow U$
(4) $T \rightarrow aTc$
(5) $T \rightarrow V$
(6) $U \rightarrow bUd$
(7) $U \rightarrow V$
(8) $V \rightarrow bVc$
(9) $V \rightarrow \varepsilon$



input = a a b c d d

Notice Nondeterminism

Machines constructed with the algorithm are often nondeterministic, even when they needn't be. This happens even with trivial languages.

Example: $L = a^n b^n$

A grammar for L is:

[1] $S \rightarrow aSb$
[2] $S \rightarrow \varepsilon$

A machine M for L is:
(0) ((p, ε, ε), (q, S))
(1) ((q, ε, S), (q, aSb))
(2) ((q, ε, S), (q, ε))
(3) ((q, a, a), (q, ε))
(4) ((q, b, b), (q, ε))

But transitions 1 and 2 make M nondeterministic.

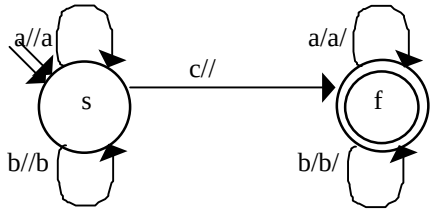
A **nondeterministic transition group** is a set of two or more transitions out of the same state that can fire on the same configuration. A **PDA is nondeterministic** if it has any nondeterministic transition groups.

A directly constructed machine for L:

Going The Other Way

Lemma: If a language is accepted by a pushdown automaton, it is a context-free language (i.e., it can be described by a context-free grammar).
Proof (by construction)

Example: $L = \{wcw^R : w \in \{a, b\}^*\}$



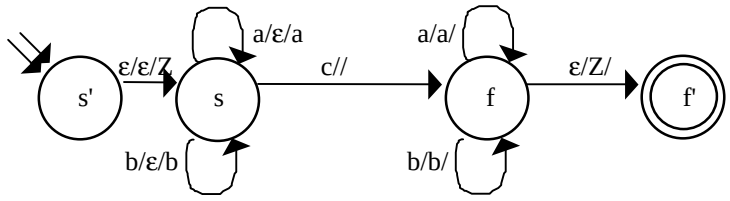
Δ contains:
((s, a, ε), (s, a))
((s, b, ε), (s, b))
((s, c, ε), (f, ε))
((f, a, a), (f, ε))
((f, b, b), (f, ε))

M = ({s, f}, {a, b, c}, {a, b}, Δ, s,{f}), where:

First Step: Make M Simple

A PDA M is simple iff:
1. there are no transitions into the start state, and
2. whenever ((q, x, β), (p, γ) is a transition of M and q is not the start state, then $\beta \in \Gamma$, and $|\gamma| \leq 2$.

Step 1: Add s' and f':

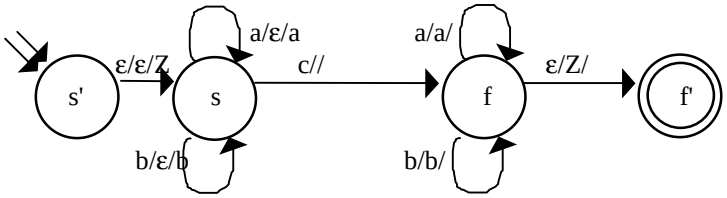


Step 2:
(1) Assure that $|\beta| \leq 1$.

(2) Assure that $|\gamma| \leq 2$.

(3) Assure that $|\beta| = 1$.

Making M Simple

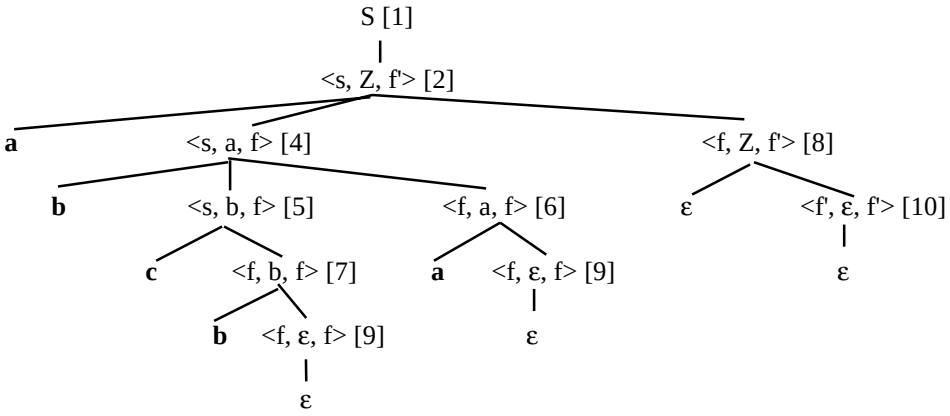


M = ({s, f, s', f'}, {a, b, c}, {a, b, Z}, Δ, s',{f'}), Δ=

	((s', ε, ε), (s, Z))
((s, a, ε), (s, a))	((s, a, Z), (s, aZ))
	((s, a, a), (s, aa))
	((s, a, b), (s, ab))
((s, b, ε), (s, b))	((s, b, Z), (s, bZ))
	((s, b, a), (s, ba))
	((s, b, b), (s, bb))
((s, c, ε), (f, ε))	((s, c, Z), (f, Z))
	((s, c, a), (f, a))
	((s, c, b), (f, b))
((f, a, a), (f, ε))	((f, a, a), (f, ε))
((f, b, b), (f, ε))	((f, b, b), (f, ε))
	((f, ε, Z), (f', ε))

Second Step - Creating the Productions

The basic idea -- simulate a leftmost derivation of M on any input string.
Example: abcba



If the nonterminal $<s_1, X, s_2> \Rightarrow^* w$, then the PDA starts in state s_1 with (at least) X on the stack and after consuming w and popping the X off the stack, it ends up in state s_2.

Start with the rule:

$S \rightarrow <s, Z, f'>$ where s is the start state, f' is the (introduced) final state and Z is the stack bottom symbol.

Transitions $((s_1, a, X), (s_2, YX))$ become a set of rules:

$<s_1, X, q> \rightarrow a <s_2, Y, r> <r, X, q>$ for $a \in \Sigma \cup \{\varepsilon\}, \forall q,r \in K$

Transitions $((s_1, a, X), (s_2, Y))$ becomes a set of rules:

$<s_1, X, q> \rightarrow a <s_2, Y, q>$ for $a \in \Sigma \cup \{\varepsilon\}, \forall q \in K$

Transitions $((s_1, a, X), (s_2, \varepsilon))$ become a rule:

$<s_1, X, s_2> \rightarrow a$ for $a \in \Sigma \cup \{\varepsilon\}$

Creating Productions from Transitions

	$S \rightarrow \langle s, Z, f' \rangle$	[1]
((s', ε, ε), (s, Z))		
((s, a, Z), (s, aZ))	$\langle s, Z, f' \rangle \rightarrow a \langle s, a, f \rangle \langle f, Z, f' \rangle$	[2]
	$\langle s, Z, s \rangle \rightarrow a \langle s, a, f \rangle \langle f, Z, s \rangle$	[x]
	$\langle s, Z, f \rangle \rightarrow a \langle s, a, s \rangle \langle s, Z, f \rangle$	[x]
	$\langle s, Z, s \rangle \rightarrow a \langle s, a, s \rangle \langle s, Z, f \rangle$	[x]
	$\langle s, Z, s' \rangle \rightarrow a \langle s, a, f \rangle \langle f, Z, s' \rangle$	[x]
((s, a, a), (s, aa))	$\langle s, a, f \rangle \rightarrow a \langle s, a, f \rangle \langle f, a, f \rangle$	[3]
((s, a, b), (s, ab))	…	
((s, b, Z), (s, bZ))	…	
((s, b, a), (s, ba))	$\langle s, a, f \rangle \rightarrow b \langle s, b, f \rangle \langle f, a, f \rangle$	[4]
((s, b, b), (s, bb))	…	
((s, c, Z), (f, Z))	…	
((s, c, a), (f, a))	$\langle s, a, f \rangle \rightarrow c \langle f, a, f \rangle$	
((s, c, b), (f, b))	$\langle s, b, f \rangle \rightarrow c \langle f, b, f \rangle$	[5]
((f, a, a), (f, ε))	$\langle f, a, f \rangle \rightarrow a \langle f, ε, f \rangle$	[6]
((f, b, b), (f, ε))	$\langle f, b, f \rangle \rightarrow b \langle f, ε, f \rangle$	[7]
((f, ε, Z), (f', ε))	$\langle f, Z, f' \rangle \rightarrow ε \langle f', ε, f' \rangle$	[8]
	$\langle f, ε, f \rangle \rightarrow ε$	[9]
	$\langle f' ε, f' \rangle \rightarrow ε$	[10]

Comparing Regular and Context-Free Languages

Regular Languages

- regular exprs.
  - or
- regular grammars
- recognize
- = DFSAs

Context-Free Languages

- context-free grammars
- parse
- = NDPDAs

Grammars and Normal Forms

Read K & S 3.7.

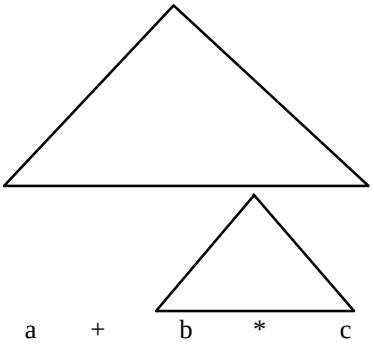
Recognizing Context-Free Languages

Two notions of recognition:

(1) Say yes or no, just like with FSMs

(2) Say yes or no, AND

if yes, describe the structure



Now it's time to worry about extracting structure (and **doing so efficiently**).

Optimizing Context-Free Languages

For regular languages:
Computation = operation of FSMs. So,
Optimization = Operations on FSMs:
Conversion to deterministic FSMs
Minimization of FSMs

For context-free languages:
Computation = operation of parsers. So,
Optimization = **Operations on languages**
Operations on grammars
Parser design

Before We Start: Operations on Grammars

There are lots of ways to transform grammars so that they are more useful for a particular purpose.
the basic idea:

1. Apply transformation 1 to G to get of undesirable property 1. Show that the language generated by G is unchanged.
2. Apply transformation 2 to G to get rid of undesirable property 2. Show that the language generated by G is unchanged AND that undesirable property 1 has not been reintroduced.
3. Continue until the grammar is in the desired form.

Examples:

- Getting rid of ε rules (nullable rules)
- Getting rid of sets of rules with a common initial terminal, e.g.,
  - $A \rightarrow aB$, $A \rightarrow aC$ become $A \rightarrow aD$, $D \rightarrow B \mid C$
- Conversion to normal forms

Normal Forms

If you want to design algorithms, it is often useful to have a limited number of input forms that you have to deal with.

Normal forms are designed to do just that. Various ones have been developed for various purposes.

Examples:

- Clause form for logical expressions to be used in resolution theorem proving
- Disjunctive normal form for database queries so that they can be entered in a query by example grid.
- Various normal forms for grammars to support specific parsing techniques.

Clause Form for Logical Expressions

$\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow [hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]$

becomes

$\neg Roman(x) \vee \neg know(x, Marcus) \vee hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)$

Disjunctive Normal Form for Queries

(category = fruit or category = vegetable)
and
(supplier = A or supplier = B)

becomes

(category = fruit and supplier = A) or
(category = fruit and supplier = B) or
(category = vegetable and supplier = A) or
(category = vegetable and supplier = B)

Category	Supplier	Price
fruit	A	
fruit	B	
vegetable	A	
vegetable	B	

Normal Forms for Grammars

Two of the most common are:

- **Chomsky Normal Form**, in which all rules are of one of the following two forms:
  - $X \rightarrow a$, where $a \in \Sigma$, or
  - $X \rightarrow BC$, where B and C are nonterminals in G

- **Greibach Normal Form**, in which all rules are of the following form:
  - $X \rightarrow a\beta$, where $a \in \Sigma$ and β is a (possibly empty) string of nonterminals

If L is a context-free language that does not contain ε, then if G is a grammar for L, G can be rewritten into both of these normal forms.

What Are Normal Forms Good For?

Examples:

- **Chomsky Normal Form**:
  - $X \rightarrow a$, where $a \in \Sigma$, or
  - $X \rightarrow BC$, where B and C are nonterminals in G

♦ The branching factor is precisely 2. Tree building algorithms can take advantage of that.

- **Greibach Normal Form**
  - $X \rightarrow a\beta$, where $a \in \Sigma$ and β is a (possibly empty) string of nonterminals

♦Precisely one nonterminal is generated for each rule application. This means that we can put a bound on the number of rule applications in any successful derivation.

Conversion to Chomsky Normal Form

Let G be a grammar for the context-free language L where $\varepsilon \notin L$.
We construct G', an equivalent grammar in Chomsky Normal Form by:

0. Initially, let G' = G.
1. Remove from G' all ε productions:
   1.1. If there is a rule $A \rightarrow \alpha B \beta$ and B is nullable, add the rule $A \rightarrow \alpha\beta$ and delete the rule $B \rightarrow \varepsilon$.
   Example:
   $S \rightarrow aA$
   $A \rightarrow B \mid CD$
   $B \rightarrow \varepsilon$
   $B \rightarrow a$
   $C \rightarrow BD$
   $D \rightarrow b$
   $D \rightarrow \varepsilon$

Conversion to Chomsky Normal Form

2. Remove from G' all unit productions (rules of the form $A \rightarrow B$, where B is a nonterminal):
   2.1. Remove from G' all unit productions of the form $A \rightarrow A$.
   2.2. For all nonterminals A, find all nonterminals B such that $A \Rightarrow^* B$, $A \neq B$.
   2.3. Create G" and add to it all rules in G' that are not unit productions.
   2.4. For all A and B satisfying 3.2, add to G"
   $A \rightarrow y1 \mid y2 \mid \ldots$ where $B \rightarrow y1 \mid y2 \mid$ is in G".
   2.5. Set G' to G".

   Example:
   $A \rightarrow a$
   $A \rightarrow B$
   $A \rightarrow EF$
   $B \rightarrow A$
   $B \rightarrow CD$
   $B \rightarrow C$
   $C \rightarrow ab$

At this point, all rules whose right hand sides have length 1 are in Chomsky Normal Form.

Conversion to Chomsky Normal Form

3. Remove from G' all productions P whose right hand sides have length greater than 1 and include a terminal (e.g., $A \rightarrow aB$ or $A \rightarrow BaC$):
   - 3.1. Create a new nonterminal T_a for each terminal a in Σ.
   - 3.2. Modify each production P by substituting T_a for each terminal a.
   - 3.3. Add to G', for each T_a, the rule $T_a \rightarrow a$

Example:

$A \rightarrow aB$
$A \rightarrow BaC$
$A \rightarrow BbC$

$T_a \rightarrow a$
$T_b \rightarrow b$

Conversion to Chomsky Normal Form

4. Remove from G' all productions P whose right hand sides have length greater than 2 (e.g., $A \rightarrow BCDE$)
   - 4.1. For each P of the form $A \rightarrow N_1N_2N_3N_4\ldots N_n$, $n > 2$, create new nonterminals M_2, M_3, … M_{n-1}.
   - 4.2. Replace P with the rule $A \rightarrow N_1M_2$.
   - 4.3. Add the rules $M_2 \rightarrow N_2M_3$, $M_3 \rightarrow N_3M_4$, … $M_{n-1} \rightarrow N_{n-1}N_n$

Example:

$A \rightarrow BCDE$ $(n = 4)$

$A \rightarrow BM_2$
$M_2 \rightarrow C\ M_3$
$M_3 \rightarrow DE$

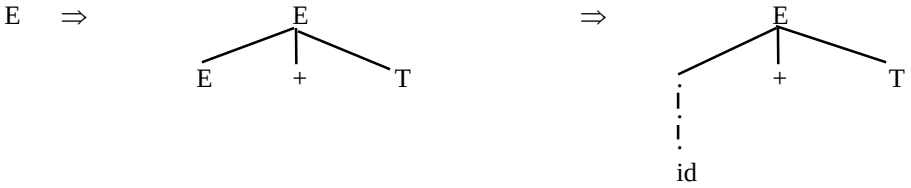
Top Down Parsing

Read K & S 3.8.
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Parsing, Sections 1 and 2.
Do Homework 15.

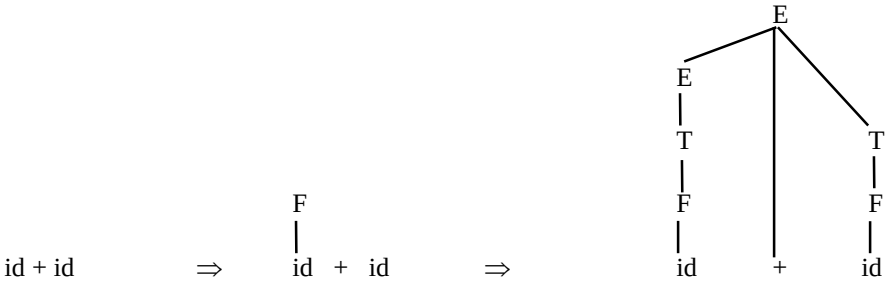
Parsing

Two basic approaches:

Top Down



Bottom Up



A Simple Parsing Example

A simple top-down parser for arithmetic expressions, given the grammar

[1] $E \rightarrow E + T$
[2] $E \rightarrow T$
[3] $T \rightarrow T * F$
[4] $T \rightarrow F$
[5] $F \rightarrow (E)$
[6] $F \rightarrow id$
[7] $F \rightarrow id(E)$

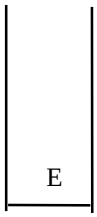
A PDA that does a top down parse:

(0) (1, ε, ε), (2, E)
(1) (2, ε, E), (2, E+T)
(2) (2, ε, E), (2, T)
(3) (2, ε, T), (2, T*F)
(4) (2, ε, T), (2, F)
(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, ε, F), (2, id(E))
(8) (2, id, id), (2, ε)
(9) (2, (, ( ), (2, ε)
(10) (2,),)), (2, ε)
(11) (2, +, +), (2, ε)
(12) (2, *, *), (2, ε)

How Does It Work?

Example: id + id * id(id)

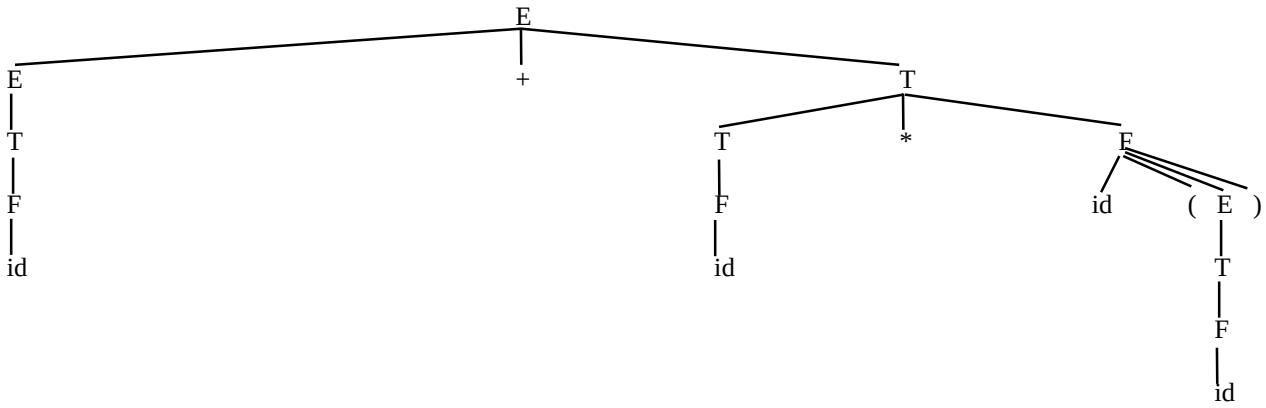
Stack:



What Does It Produce?

The leftmost derivation of the string. Why?

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow$
$id + T * F \Rightarrow id + F * F \Rightarrow id + id * F \Rightarrow$
$id + id * id(E) \Rightarrow id + id * id(T) \Rightarrow$
$id + id * id(F) \Rightarrow id + id * id(id)$



But the Process Isn't Deterministic

(0) (1, ε, ε), (2, E)
(1) (2, ε, E), (2, E+T)
(2) (2, ε, E), (2, T)
(3) (2, ε, T), (2, T*F)
(4) (2, ε, T), (2, F)
(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, ε, F), (2, id(E))
(8) (2, id, id), (2, ε)
(9) (2, (, ( ), (2, ε)
(10) (2,),)), (2, ε)
(11) (2, +, +), (2, ε)
(12) (2, *, *), (2, ε)

nondeterministic

nondeterministic

nondeterministic

Is Nondeterminism A Problem?

Yes.

In the case of regular languages, we could cope with nondeterminism in either of two ways:

- Create an equivalent deterministic recognizer (FSM)
- Simulate the nondeterministic FSM in a number of steps that was still linear in the length of the input string.

For context-free languages, however,

- The best straightforward general algorithm for recognizing a string is $O(n^3)$ and the best (very complicated) algorithm is based on a reduction to matrix multiplication, which may get close to $O(n^2)$.

We'd really like to find a deterministic parsing algorithm that could run in time proportional to the length of the input string.

Is It Possible to Eliminate Nondeterminism?

In this case: Yes

In general: No

Some definitions:

- A PDA **M is deterministic** if it has no two transitions such that for some (state, input, stack sequence) the two transitions could both be taken.

- A language **L is deterministic context-free** if L$ = L(M) for some deterministic PDA M.

Theorem: The class of deterministic context-free languages is a ***proper*** subset of the class of context-free languages.

Proof: Later.

Adding a Terminator to the Language

We define the class of deterministic context-free languages with respect to a terminator ($) because we want that class to be as large as possible.

Theorem: Every deterministic CFL (as just defined) is a context-free language.

Proof:

Without the terminator ($), many seemingly deterministic cfls aren't. Example:

$a^* \cup \{a^n b^n : n > 0\}$

Possible Solutions to the Nondeterminism Problem

1) **Modify the language**
   - Add a terminator $
2) **Change the parsing algorithm**
3) **Modify the grammar**

Modifying the Parsing Algorithm

What if we add the ability to look one character ahead in the input string?

Example: id + id * id(id)
⇑
$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow$
$id + T * F \Rightarrow id + F * F \Rightarrow id + id * F$

Considering transitions:

(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, ε, F), (2, id(E))

If we add to the state an indication of what character is next, we have:

(5) (2, **(**, ε, F), (2, (E))
(6) (2, **id**, ε, F), (2, id)
(7) (2, **id**, ε, F), (2, id(E))

Modifying the Language

So we've solved part of the problem. But what do we do when we come to the end of the input? What will be the state indicator then?

The solution is to modify the language. Instead of building a machine to accept L, we will build a machine to accept L$.

Using Lookahead

		(0) (1, ε, ε), (2, E))
[1]	$E \rightarrow E + T$	(1) (2, ε, E), (2, E+T)
[2]	$E \rightarrow T$	(2) (2, ε, E), (2, T)
[3]	$T \rightarrow T * F$	(3) (2, ε, T), (2, T*F)
[4]	$T \rightarrow F$	(4) (2, ε, T), (2, F)
[5]	$F \rightarrow (E)$	(5) (2, **(**, ε, F), (2, (E))
[6]	$F \rightarrow id$	(6) (2, **id**, ε, F), (2, id)
[7]	$F \rightarrow id(E)$	(7) (2, **id**, ε, F),(2, id(E))
		(8) (2, id, id), (2, ε)
		(9) (2, (, ( ), (2, ε)
		(10) (2,),)), (2, ε)
		(11) (2, +, +), (2, ε)
		(12) (2, *, *), (2, ε)

For now, we'll ignore the issue of when we read the lookahead character and the fact that we only care about it if the top symbol on the stack is F.

Possible Solutions to the Nondeterminism Problem

1) **Modify the language**
   - Add a terminator $
2) **Change the parsing algorithm**
   - Add one character look ahead
3) **Modify the grammar**

Modifying the Grammar

Getting rid of identical first symbols:

[6]	$F \rightarrow id$	(6) (2, id, ε, F),(2, id)
[7]	$F \rightarrow id(E)$	(7) (2, id, ε, F),(2, id(E))

Replace with:

[6']	$F \rightarrow id\ A$	(6') (2, id, ε, F), (2, id A)
[7']	$A \rightarrow \varepsilon$	(7') (2, ¬(, ε, A), (2, ε)
[8']	$A \rightarrow (E)$	(8') (2, (, ε, A), (2, (E))

The general rule for **left factoring**:

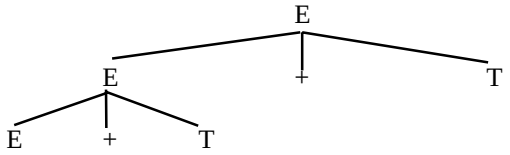
Whenever $A \rightarrow \alpha\beta_1$
$A \rightarrow \alpha\beta_2 \ldots$
$A \rightarrow \alpha\beta_n$
are rules with $\alpha \neq \varepsilon$ and $n \geq 2$, then replace them by the rules:
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1$
$A' \rightarrow \beta_2 \ldots$
$A' \rightarrow \beta_n$

Modifying the Grammar

Getting rid of left recursion:

[1]	$E \rightarrow E + T$	(1) (2, ε, E), (2, E+T)
[2]	$E \rightarrow T$	(2) (2, ε, E), (2, T)

The problem:



Replace with:

[1]	$E \rightarrow T\ E'$	(1) (2, ε, E), (2, T E')
[2]	$E' \rightarrow + T\ E'$	(2) (2, ε, E'), (2, + T E')
[3]	$E' \rightarrow \varepsilon$	(3) (2, ε, E'), (2, ε)

Getting Rid of Left Recursion

The general rule for eliminating **left recursion**:

If G contains the following rules:	Replace them with:
$A \rightarrow A\alpha_1$	$A' \rightarrow \alpha_1 A'$
$A \rightarrow A\alpha_2$ …	$A' \rightarrow \alpha_2 A'$ …
$A \rightarrow A\alpha_3$	$A' \rightarrow \alpha_3 A'$
$A \rightarrow A\alpha_n$	$A' \rightarrow \alpha_n A'$
	$A' \rightarrow \varepsilon$
$A \rightarrow \beta_1$ (where β's do not start with $A\alpha$)	$A \rightarrow \beta_1 A'$
$A \rightarrow \beta_2$	$A \rightarrow \beta_2 A'$
…	…
$A \rightarrow \beta_m$	$A \rightarrow \beta_m A'$

and $n > 0$, then

Possible Solutions to the Nondeterminism Problem

I. **Modify the language**
   A. Add a terminator $
II. **Change the parsing algorithm**
   A. Add one character look ahead
III. **Modify the grammar**
   A. Left factor
   B. Get rid of left recursion

LL(k) Languages

We have just offered heuristic rules for getting rid of some nondeterminism.

We know that not all context-free languages are deterministic, so there are some languages for which these rules won't work.

We define a **grammar** to be **LL(k)** if it is possible to decide what production to apply by looking ahead at most k symbols in the input string.

Specifically, a **grammar** G is **LL(1)** iff, whenever
$A \rightarrow \alpha \mid \beta$ are two rules in G:

1. For no terminal a do α and β derive strings beginning with a.
2. At most one of $\alpha \mid \beta$ can derive ε.
3. If $\beta \Rightarrow^* \varepsilon$, then α does not derive any strings beginning with a terminal in FOLLOW(A), defined to be the set of terminals that can immediately follow A in some sentential form.

We define a **language** to be **LL(k)** if there exists an LL(k) grammar for it.

Implementing an LL(1) Parser

If a language L has an LL(1) grammar, then we can build a deterministic LL(1) parser for it. Such a parser scans the input **L**eft to right and builds a **L**eftmost derivation.

The heart of an LL(1) parser is the parsing table, which tells it which production to apply at each step.
For example, here is the parsing table for our revised grammar of arithmetic expressions without function calls:

V\Σ	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Given input id + id * id, the first few moves of this parser will be:

	E	id + id * id$
E→TE'	TE'	id + id * id$
T→FT'	FT'E'	id + id * id$
F→id	idT'E'	id + id * id$
	T'E'	+ id * id$
T'→ε	E'	+ id * id$

But What If We Need a Language That Isn't LL(1)?

Example:

ST → if C then ST else ST
ST → if C then ST

We can apply left factoring to yield:

ST → if C then ST S'
S' → else ST | ε

Now we've procrastinated the decision. But the language is still ambiguous. What if the input is

if C_1 then if C_2 then ST_1 else ST_2

Which bracketing (rule) should we choose?

A common practice is to choose S' → else ST

We can force this if we create the parsing table by hand.

Possible Solutions to the Nondeterminism Problem

I. **Modify the language**
   A. Add a terminator $
II. **Change the parsing algorithm**
   A. Add one character look ahead
   B. Use a parsing table
   C. Tailor parsing table entries by hand
III. **Modify the grammar**
   A. Left factor
   B. Get rid of left recursion

The Price We Pay

Old Grammar

[1] $E \rightarrow E + T$
[2] $E \rightarrow T$

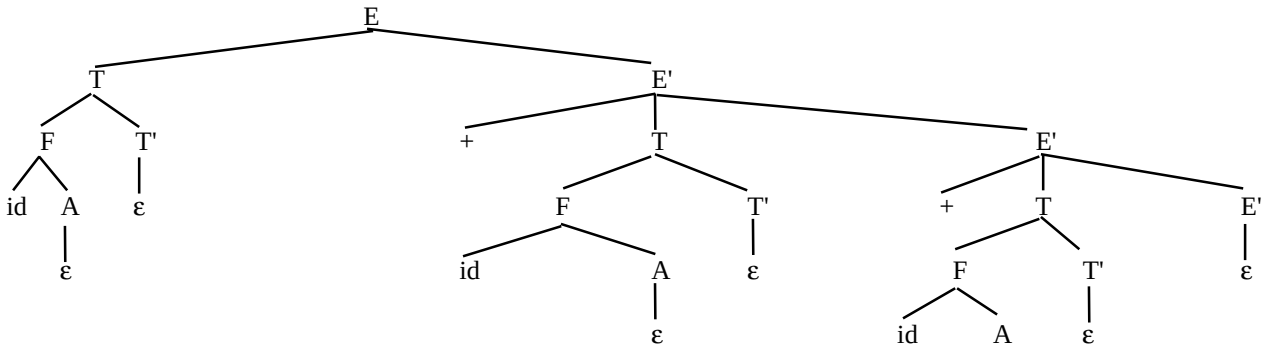
[3] $T \rightarrow T * F$
[4] $T \rightarrow F$

[5] $F \rightarrow (E)$
[6] $F \rightarrow id$
[7] $F \rightarrow id(E)$

New Grammar

$E \rightarrow TE'$
$E' \rightarrow +TE'$
$E' \rightarrow \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'$
$T' \rightarrow \varepsilon$
$F \rightarrow (E)$
$F \rightarrow idA$
$A \rightarrow \varepsilon$
$A \rightarrow (E)$

input = id + id + id



Comparing Regular and Context-Free Languages

Regular Languages

- regular exprs.
  or
- regular grammars
- = DFSAs
- recognize
- minimize FSAs

Context-Free Languages

- context-free grammars

- = NDPDAs
- parse
- find deterministic grammars
- find efficient parsers

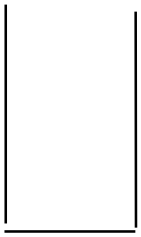
Bottom Up Parsing

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Parsing, Section 3.

Bottom Up Parsing

An Example:

[1] $E \rightarrow E + T$
[2] $E \rightarrow T$
[3] $T \rightarrow T * F$
[4] $T \rightarrow F$
[5] $F \rightarrow (E)$
[6] $F \rightarrow id$

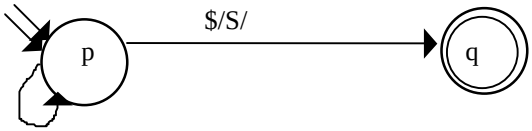


id + id * id $

Creating a Bottom Up PDA

There are two basic actions:
1. Shift an input symbol onto the stack
2. Reduce a string of stack symbols to a nonterminal

M will be:



So, to construct M from a grammar G, we need the following transitions:

(1) The shift transitions:
$((p, a, \varepsilon), (p, a))$, for each $a \in \Sigma$

(2) The reduce transitions:
$((p, \varepsilon, \alpha^R), (p, A))$, for each rule $A \rightarrow \alpha$ in G.

(3) The finish up transition (accept):
$((p, \$, S), (q, \varepsilon))$

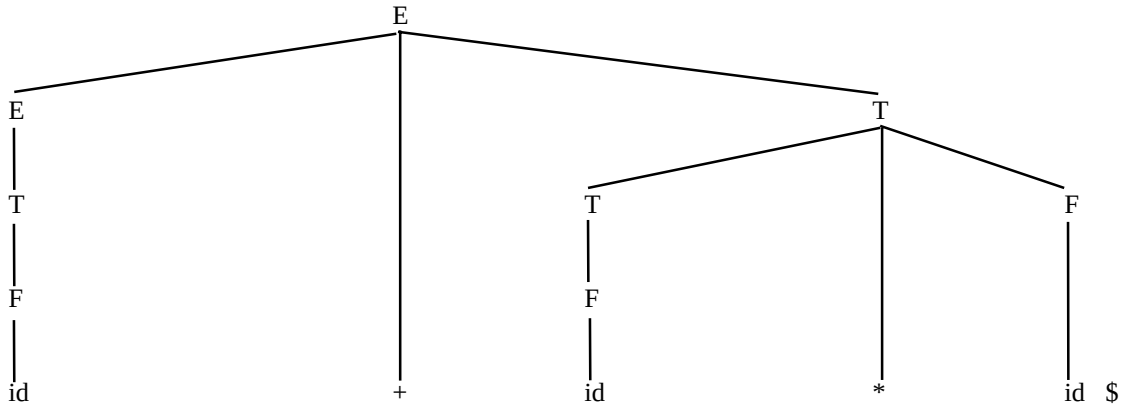
(This is the "bottom-up" CFG to PDA conversion algorithm.)

M for Expressions

0	(p, a, ε), (p, a) for each a ∈ Σ
1	(p, ε, T + E), (p, E)
2	(p, ε, T), (p, E)
3	(p, ε, F * T), (p, T)
4	(p, ε, F), (p, T)
5	(p, ε, ")"E"("), (p, F)
6	(p, ε, id), (p, F)
7	(p, $, E), (q, ε)

trans (action)	*state*	*unread input*	*stack*
	p	id + id * id$	ε
0 (shift)	p	+ id * id$	id
6 (reduce F → id)	p	+ id * id$	F
4 (reduce T → F)	p	+ id * id$	T
2 (reduce E → T)	p	+ id * id$	E
0 (shift)	p	id * id$	+E
0 (shift)	p	* id$	id+E
6 (reduce F → id)	p	* id$	F+E
4 (reduce T → F)	p	* id$	T+E (could also reduce)
0 (shift)	p	id$	*T+E
0 (shift)	p	$	id*T+E
6 (reduce F → id)	p	$	F*T+E (could also reduce T → F)
3 (reduce T → T * F)	p	$	T+E
1 (reduce E → E + T)	p	$	E
7 (accept)	q	$	ε

The Parse Tree



Producing the Rightmost Derivation

We can reconstruct the derivation that we found by reading the results of the parse bottom to top, producing:

E ⇒
E+ T ⇒
E+ T* F⇒
E+ T* id⇒
E+ F* id⇒
E+ id* id⇒
T+ id*id⇒
F+ id*id⇒
id+ id*id

This is exactly the rightmost derivation of the input string.

Possible Solutions to the Nondeterminism Problem

1) **Modify the language**
   - Add a terminator $

2) **Change the parsing algorithm**
   - *Add one character look ahead*
   - *Use a parsing table*
   - *Tailor parsing table entries by hand*
   - **Switch to a bottom-up parser**

3) **Modify the grammar**
   - *Left factor*
   - *Get rid of left recursion*

Solving the Shift vs. Reduce Ambiguity With a Precedence Relation

Let's return to the problem of deciding when to shift and when to reduce (as in our example).

We chose, correctly, to shift * onto the stack, instead of reducing T+E to E.

This corresponds to knowing that "+" has low precedence, so if there are any other operations, we need to do them first.

Solution:
1. Add a one character lookahead capability.
2. Define the precedence relation

$P \subseteq$ (V $\times$ $\{\Sigma \cup \$\}$)

top	next
stack	input
symbol	symbol

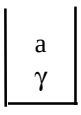
If (a,b) is in P, we reduce (without consuming the input) . Otherwise we shift (consuming the input).

How Does It Work?

We're reconstructing rightmost derivations backwards. So suppose a rightmost derivation contains

$\beta\gamma abx$
$\Uparrow$ ⟵ corresponding to a rule $A \rightarrow \gamma a$ and not some rule $X \rightarrow ab$
βAbx
$\Uparrow^*$
S

We want to undo rule A. So if the top of the stack is



and the next input character is b, we reduce now, before we put the b on the stack.

To make this happen, we put (a, b) in P. That means we'll try to reduce if a is on top of the stack and b is the next character. We will actually succeed if the next part of the stack is γ.

Example

T*F
⇑ ⟵ corresponding to a rule T→T*F
T
⇑*
E

Input: id * id * id

We want to undo rule T. So if the top of the stack is

F
*
T

and the next input character is anything legal, we reduce.

The precedence relation for expressions:

V\Σ	(	)	id	+	*	$
(						
)		•		•	•	•
id		•		•	•	•
+						
*						
E						
T		•		•		•
F		•		•	•	•

A Different Example

E+T
⇑* ⟵ corresponding to a rule E→E+T
E

We want to undo rule E if the input is E + T $
or E + T + id
but not E + T * id

The top of the stack is

T
+
E

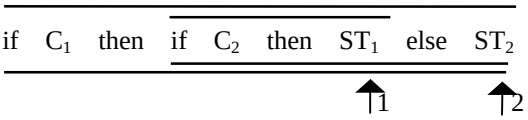
The precedence relation for expressions:

V\Σ	(	)	id	+	*	$
(						
)		•		•	•	•
id		•		•	•	•
+						
*						
E						
T		•		•		•
F		•		•	•	•

What About If Then Else?

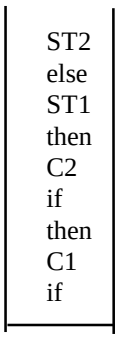
ST → if C then ST else ST
ST → if C then ST

What if the input is



Which bracketing (rule) should we choose?

We don't put (ST, else) in the precedence relation, so we will not reduce at 1. At 2, we reduce:



Resolving Reduce vs. Reduce Ambiguities

0	(p, a, ε), (p, a) for each a ∈ Σ
1	(p, ε, T + E), (p, E)
2	(p, ε, T), (p, E)
3	(p, ε, F * T), (p, T)
4	(p, ε, F), (p, T)
5	(p, ε, ")" E "("), (p, F)
6	(p, ε, id), (p, F)
7	(p, $, E), (q, ε)

trans (action)	***state***	***unread input***	***stack***
	p	id + id * id$	ε
0 (shift)	p	+ id * id$	id
6 (reduce F → id)	p	+ id * id$	F
4 (reduce T → F)	p	+ id * id$	T
2 (reduce E → T)	p	+ id * id$	E
0 (shift)	p	id * id$	+E
0 (shift)	p	* id$	id+E
6 (reduce F → id)	p	* id$	F+E
4 (reduce T → F)	p	* id$	T+E (could also reduce)
0 (shift)	p	id$	*T+E
0 (shift)	p	$	id*T+E
6 (reduce F → id)	p	$	F*T+E (**could also reduce T → F**)
3 (reduce T → T * F)	p	$	T+E
1 (reduce E → E + T)	p	$	E
7 (accept)	q	$	ε

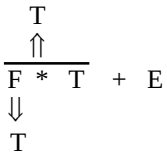
The Longest Prefix Heuristic

A simple to implement heuristic rule, when faced with competing reductions, is:

Choose the longest possible stack string to reduce.

Example:

Suppose the stack has



We call grammars that become unambiguous with the addition of a precedence relation and the longest string reduction heuristic **weak precedence grammars**.

Possible Solutions to the Nondeterminism Problem in a Bottom Up Parser

1) **Modify the language**
   - Add a terminator $

2) **Change the parsing algorithm**
   - Add one character lookahead
   - Use a precedence table
   - Add the longest first heuristic for reduction
   - **Use an LR parser**

3) **Modify the grammar**

LR Parsers

LR parsers scan each input **L**eft to right and build a **R**ightmost derivation. They operate bottom up and deterministically using a parsing table derived from a grammar for the language to be recognized.

A grammar that can be parsed by an LR parser examining up to k input symbols on each move is an **LR(k)** grammar. Practical LR parsers set k to 1.

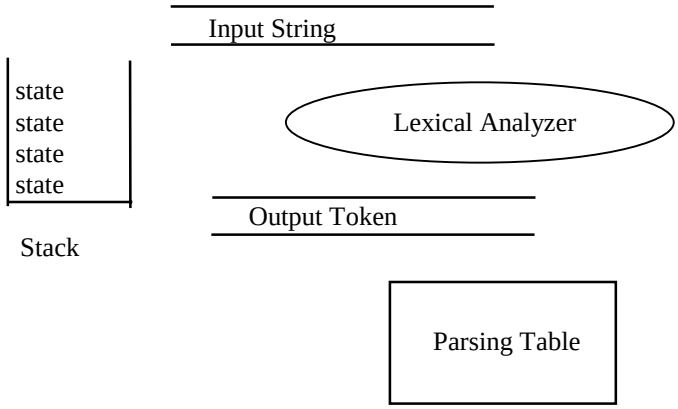
An LALR (or Look Ahead LR) parser is a specific kind of LR parser that has two desirable properties:
- The parsing table is not huge.
- Most useful languages can be parsed.

Another big reason to use an LALR parser:

There are automatic tools that will construct the required parsing table from a grammar and some optional additional information.

We will be using such a tool: **yacc**

How an LR Parser Works



In simple cases, think of the "states" on the stack as corresponding to either terminal or nonterminal characters.

In more complicated cases, the states contain more information: they encode both the top stack symbol and some facts about lower objects in the stack. This information is used to determine which action to take in situations that would otherwise be ambiguous.

The Actions the Parser Can Take

At each step of its operation, an LR parser does the following two things:

1) Based on its current state, it decides whether it needs a lookahead token. If it does, it gets one.
2) Based on its current state and the lookahead token if there is one, it chooses one of four possible actions:
   - Shift the lookahead token onto the stack and clear the lookahead token.
   - Reduce the top elements of the stack according to some rule of the grammar.
   - Detect the end of the input and accept the input string.
   - Detect an error in the input.

A Simple Example

0: S → rhyme $end ;
1: rhyme → sound place ;
2: sound → DING DONG ;
3: place → DELL

state 0 *(empty)*
$accept : _rhyme $end
DING shift 3
. error
rhyme goto 1
sound goto 2

⇐ *the rule this came from*
state 3
current position of input
if none of the others match
push state 2

state 1 *(rhyme)*
$accept : rhyme_$end
$end accept
. error

if we see EOF, accept

state 2 *(sound)*
rhyme : sound_place
DELL shift 5
. error
place goto 4

by rule 1

state 3 *(DING)*
sound : DING_DONG
DONG shift 6
. error

state 4 *(place)*
rhyme : sound place_ (1)
. reduce 1

state 5 *(DELL)*
place : DELL_ (3)
. reduce 3

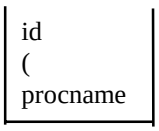
state 6 *(DONG)*
sound : DING DONG_ (2)
. reduce 2

When the States Are More than Just Stack Symbols

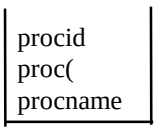
[1] <stmt> → procname (<paramlist>)
[2] <stmt> → <exp> := <exp>
[3] <paramlist> → <paramlist>, <param> | <param>
[4] <param> → id
[5] <exp> → arrayname (<subscriptlist>)
[6] <subscriptlist> → <subscriptlist>, <sub> | <sub>
[7] <sub> → id

Example:

procname (id)



Should we reduce id by rule 4 or rule 7?



The parsing table can get complicated as we incorporate more stack history into the states.

The Language Interpretation Problem:

Input: -(17 * 83.56) + 72 / 12

Output: -1414.52

The Language Interpretation Problem:

Input: -(17 * 83.56) + 72 / 12

Compute the answer

Output: -1414.52

The Language Interpretation Problem:

Input: -(17 * 83.56) + 72 / 12

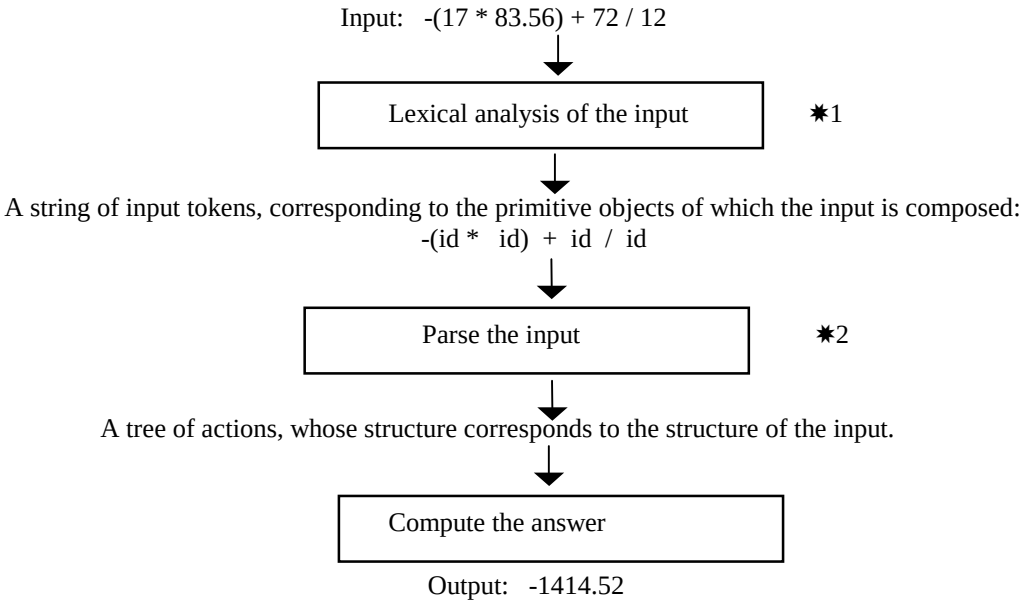
Parse the input ✷2

A tree of actions, whose structure corresponds to the structure of the input.

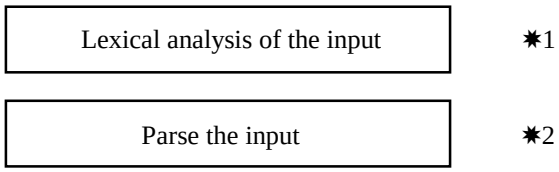
Compute the answer

Output: -1414.52

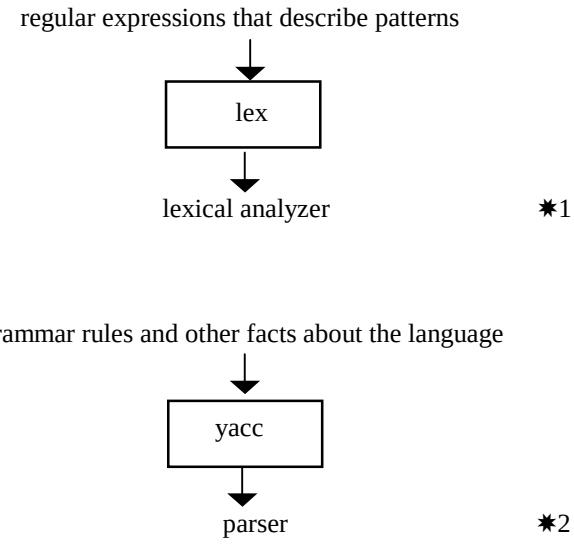
The Language Interpretation Problem:



yacc and lex



Where do the procedures to do these things come from?



lex

The input to lex:

```
definitions
%%
rules
%%
user routines
```

All strings that are not matched by any rule are simply copied to the output.

Rules:

```
[ \t]+;                                          get rid of blanks and tabs

[A-Za-z][A-Za-z0-9]*    return(ID);              find identifiers

[0-9]+          {       sscanf(yytext, "%d", &yylval);
                        return (INTEGER);  }     return INTEGER and put the value in yylval
```

How Does lex Deal with Ambiguity in Rules?

lex invokes two disambiguating rules:

1. The longest match is prefered.
2. Among rules that matched the same number of characters, the rule given first is preferred.

Example:

```
integer     action 1
[a-z]+      action 2
```

```
input:          integers        take action 2
                integer         take action 1
```

yacc
(**Y**et **A**nother **C**ompiler **C**ompiler)

The input to yacc:

```
declarations
%%
rules
%%
#include "lex.yy.c"
any other programs
```

This structure means that lex.yy.c will be compiled as part of y.tab.c, so it will have access to the same token names.

Declarations:

```
%token name1  name2  …
```

Rules:

```
V       : a   b   c
V       : a   b   c             {action}
V       : a   b   c             {$$ = $2}       returns the value of b
```

Example

Input to yacc:

```
%token DING DONG DELL
%%
rhyme :  sound place ;
sound :  DING DONG ;
place :  DELL
%%
#include "lex.yy.c"
```

```
state 0 (empty)
        $accept : _rhyme $end
        DING shift 3
        . error
        rhyme goto 1
        sound goto 2
state 1 (rhyme)
        $accept : rhyme_$end
        $end accept
        . error
state 2 (sound)
        rhyme : sound_place
        DELL shift 5
        . error
        place goto 4
state 3 (DING)
        sound : DING_DONG
        DONG shift 6
        . error
state 4 (place)
        rhyme : sound place_ (1)
        . reduce 1
state 5 (DELL)
        place : DELL_ (3)
        . reduce 3
state 6 (DONG)
        sound : DING DONG_ (2)
        . reduce 2
```

How Does yacc Deal with Ambiguity in Grammars?

The parser table that yacc creates represents some decision about what to do if there is ambiguity in the input grammar rules. How does yacc make those decisions? By default, yacc invokes two disambiguating rules:

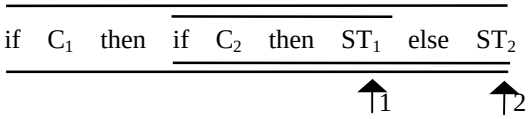
1. In the case of a shift/reduce conflict, shift.
2. In the case of a reduce/reduce conflict, reduce by the earlier grammar rule.

yacc tells you when it has had to invoke these rules.

Shift/Reduce Conflicts - If Then Else

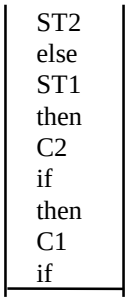
$ST \rightarrow$ if C then ST else ST
$ST \rightarrow$ if C then ST

What if the input is



Which bracketing (rule) should we choose?

yacc will choose to shift rather than reduce.

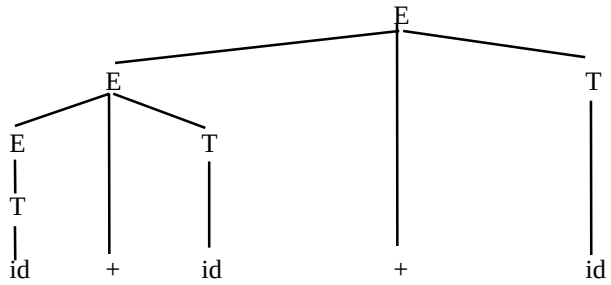


Shift/Reduce Conflicts - Left Associativity

We know that we can force left associativity by writing it into our grammars.

Example:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow id$



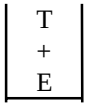
What does the shift rather than reduce heuristic if we instead write:

$E \rightarrow E + E$
$E \rightarrow id$

id + id + id

Shift/Reduce Conflicts - Operator Precedence

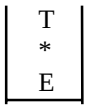
Recall the problem: input: id + id * id



Should we reduce or shift on * ?

The "always shift" rule solves this problem.

But what about: id * id + id



Should we reduce or shift on + ?

This time, if we shift, we'll fail.

One solution was the precedence table, derived from an unambiguous grammar, which can be encoded into the parsing table of an LR parser, since it tells us what to do for each top-of-stack, input character combination.

Operator Precedence

We know that we can write an unambiguous grammar for arithmetic expressions that gets the precedence right. But it turns out that we can build a faster parser if we instead write:

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

And, in addition, we specify operator precedence. In yacc, we specify associativity (since we might not always want left) and precedence using statements in the declaration section of our grammar:

```
%left '+'  '-'
%left '*'  '/'
```

Operators on the first line have lower precedence than operators on the second line, and so forth.

Reduce/Reduce Conflicts

Recall:

2. In the case of a reduce/reduce conflict, reduce by the earlier grammar rule.

This can easily be used to simulate the longest prefix heuristic, "Choose the longest possible stack string to reduce."

[1] $E \rightarrow E + T$
[2] $E \rightarrow T$
[3] $T \rightarrow T * F$
[4] $T \rightarrow F$
[5] $F \rightarrow (E)$
[6] $F \rightarrow id$

Generating an Executable System

Step 1: Create the input to lex and the input to yacc.

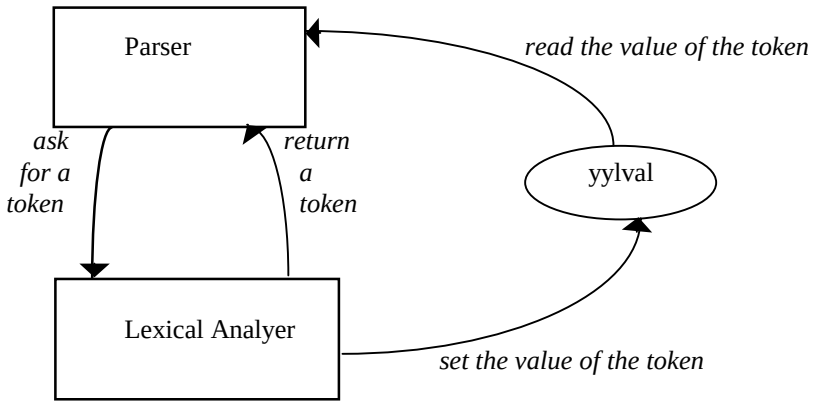
Step 2:

$ lex ourlex.l	creates lex.yy.c
$ yacc ouryacc.y	creates y.tab.c
$ cc -o ourprog y.tab.c -ly -ll	actually compiles y.tab.c and lex.yy.c, which is included. -ly links the yacc library, which includes main and yyerror. -ll links the lex library

Step 3: Run the program

$ ourprog

Runtime Communication Between lex and yacc-Generated Modules



Summary

Efficient parsers for languages with the complexity of a typical programming language or command line interface:

- Make use of special purpose constructs, like precedence, that are very important in the target languages.
- May need complex transition functions to capture all the relevant history in the stack.
- Use heuristic rules, like shift instead of reduce, that have been shown to work most of the time.
- Would be very difficult to construct by hand (as a result of all of the above).
- Can easily be built using a tool like yacc.

Languages That Are and Are Not Context-Free

Read K & S 3.5, 3.6, 3.7.
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Closure Properties of Context-Free Languages
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: The Context-Free Pumping Lemma.
Do Homework 16.

Deciding Whether a Language is Context-Free

Theorem: There exist languages that are not context-free.

Proof:
(1) There are a countably infinite number of context-free languages. This true because every description of a context-free language is of finite length, so there are a countably infinite number of such descriptions.

(2) There are an uncountable number of languages.

Thus there are more languages than there are context-free languages.

So there must exist some languages that are not context-free.

Example: $\{a^n b^n c^n\}$

Showing that a Language is Context-Free

Techniques for showing that a language L is context-free:

1. Exhibit a context-free grammar for L.
2. Exhibit a PDA for L.
3. Use the closure properties of context-free languages.

Unfortunately, these are weaker than they are for regular languages.

The Context-Free Languages are Closed Under Union

Let $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and
$G_2 = (V_2, \Sigma_2, R_2, S_2)$

Assume that G_1 and G_2 have disjoint sets of nonterminals, not including S.

Let $L = L(G_1) \cup L(G_2)$

We can show that L is context-free by exhibiting a CFG for it:

The Context-Free Languages are Closed Under Concatenation

Let $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and
$G_2 = (V_2, \Sigma_2, R_2, S_2)$

Assume that G_1 and G_2 have disjoint sets of nonterminals, not including S.

Let $L = L(G_1)\, L(G_2)$

We can show that L is context-free by exhibiting a CFG for it:

The Context-Free Languages are Closed Under Kleene Star

Let $G_1 = (V_1, \Sigma_1, R_1, S_1)$

Assume that G_1 does not have the nonterminal S.

Let $L = L(G_1)^*$

We can show that L is context-free by exhibiting a CFG for it:

What About Intersection and Complement?

We know that they share a fate, since

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

But what fate?

We proved closure for regular languages two different ways. Can we use either of them here:

1. Given a deterministic automaton for L, construct an automaton for its complement. Argue that, if closed under complement and union, must be closed under intersection.
2. Given automata for L_1 and L_2, construct a new automaton for $L_1 \cap L_2$ by simulating the parallel operation of the two original machines, using states that are the Cartesian product of the sets of states of the two original machines.

More on this later.

The Intersection of a Context-Free Language and a Regular Language is Context-Free

$L = L(M_1)$, a PDA $= (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, F_1)$
$R = L(M_2)$, a deterministic FSA $= (K_2, \Sigma, \delta, s_2, F_2)$

We construct a new PDA, M_3, that accepts $L \cap R$ by simulating the parallel execution of M_1 and M_2.

$M = (K_1 \times K_2, \Sigma, \Gamma_1, \Delta, (s_1, s_2), F_1 \times F_2)$

Insert into Δ:

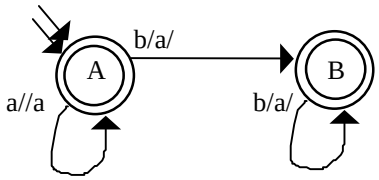
For each rule $((q_1, a, \beta), (p_1, \gamma))$ in Δ_1,
and each rule (q_2, a, p_2) in δ,
$(((q_1, q_2), a, \beta), ((p_1, p_2), \gamma))$

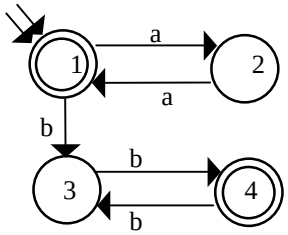
For each rule $((q_1, \varepsilon, \beta), (p_1, \gamma)$ in Δ_1,
and each state q_2 in K_2,
$(((q_1, q_2), \varepsilon, \beta), ((p_1, q_2), \gamma))$

This works because: we can get away with only one stack.

Example

L = $a^n b^n$ ∩ (aa)*(bb)*





((A, a, ε), (A, a))
((A, b, a), (B, ε))
((B, b, a), (B, ε))

(1, a, 2)
(1, b, 3)
(2, a, 1)
(3, b, 4)
(4, b, 3)

A PDA for L:

Don't Try to Use Closure Backwards

One Closure Theorem:

If L_1 and L_2 are context free, then so is

$$L_3 = L_1 \cup L_2.$$

But what if L_3 and L_1 are context free? What can we say about L_2?

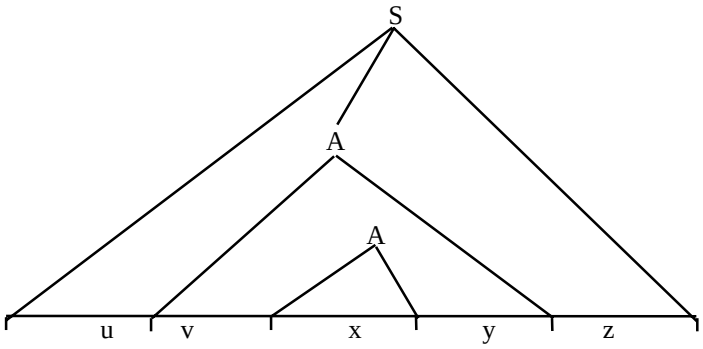
$$L_3 = L_1 \cup L_2.$$

Example:

$$a^n b^n c^* = a^n b^n c^* \cup a^n b^n c^n$$

The Context-Free Pumping Lemma

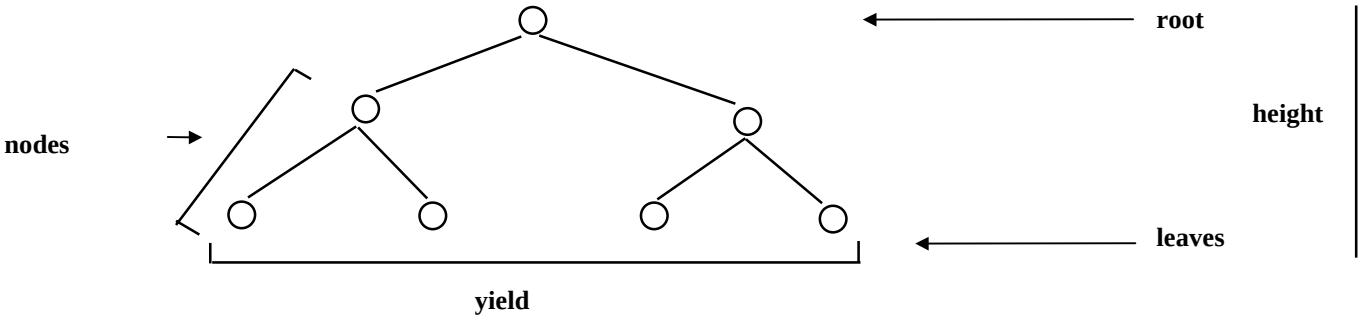
This time we use parse trees, not automata as the basis for our argument.



If L is a context-free language, and if w is a string in L where |w| > K, for some value of K, then w can be rewritten as uvxyz, where |vy| > 0 and |vxy| ≤ M, for some value of M.

uxz, uvxyz, uvvxyyz, uvvvxyyyz, etc. (i.e., $uv^n xy^n z$, for $n \geq 0$) are all in L.

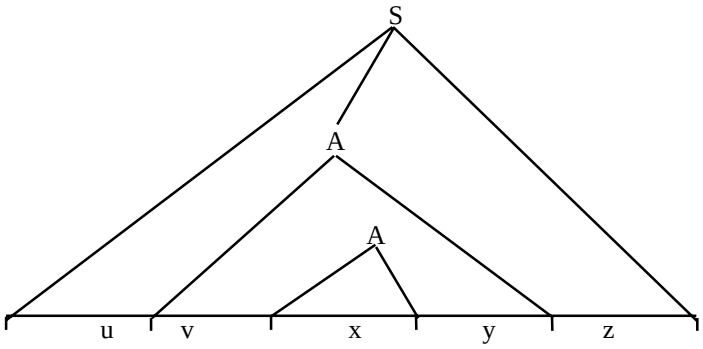
Some Tree Basics



Theorem: The length of the yield of any tree T with height H and branching factor (**fanout**) B is $\leq B^H$.

Proof: By induction on H. If H is 1, then just a single rule applies. By definition of fanout, the longest yield is B.
Assume true for H = n.
Consider a tree with H = n + 1. It consists of a root, and some number of subtrees, each of which is of height ≤ n (so induction hypothesis holds) and yield $\leq B^n$. The number of subtrees ≤ B. So the yield must be $\leq B(B^n)$ or B^{n+1}.

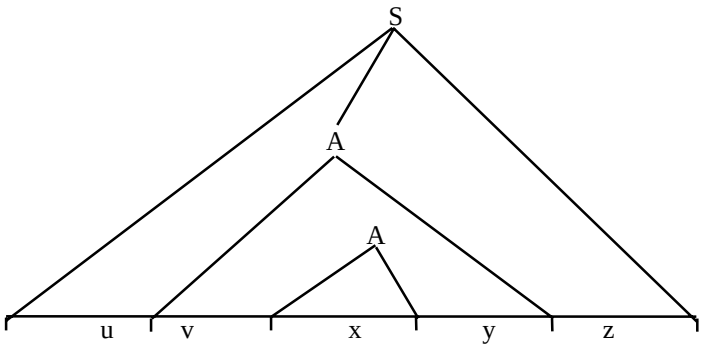
What Is K?



Let T be the number of nonterminals in G.
If there is a tree of height > T, then some nonterminal occurs more than once on some path. If it does, we can pump its yield.
Since a tree of height = T can produce only strings of length $\leq B^T$, any string of length $> B^T$ must have a repeated nonterminal and thus be pumpable.

So $K = B^T$, where T is the number of nonterminals in G and B is the branching factor (fanout).

What is M?



Assume that we are considering the bottom most two occurrences of some nonterminal. Then the yield of the upper one is at most B^{T+1} (since only one nonterminal repeats).

So $M = B^{T+1}$.

The Context-Free Pumping Lemma

Theorem: Let G = (V, Σ, R, S) be a context-free grammar with T nonterminal symbols and fanout B. Then any string $w \in L(G)$ where $|w| > K$ (B^T) can be rewritten as w = uvxyz in such a way that:

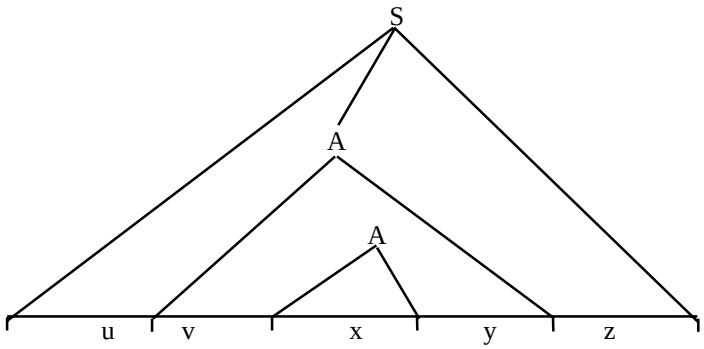
- $|vy| > 0$,
- $|vxy| \leq M$ (B^{T+1}), (making this the "strong" form),
- for every $n \geq 0$, uv^nxy^nz is in L(G).

Proof:
Let w be such a string and let T be the parse tree with root labeled S and with yield w that has the smallest number of leaves among all parse trees with the same root and yield. T has a path of length at least T+1, with a bottommost repeated nonterminal, which we'll call A. Clearly v and y can be repeated any number of times (including 0). If |vy| = 0, then there would be a tree with root S and yield w with fewer leaves than T. Finally, $|vxy| \leq B^{T+1}$.

An Example of Pumping

$L = \{a^nb^nc^n : n \geq 0\}$

Choose $w = a^ib^ic^i$ where $i > \lceil K/3 \rceil$ (making |w| > K)



Unfortunately, we don't know where v and y fall. But there are two possibilities:

1. If vy contains all three symbols, then at least one of v or y must contain two of them. But then uvvxyyz contains at least one out of order symbol.
2. If vy contains only one or two of the symbols, then uvvxyyz must contain unequal numbers of the symbols.

Using the Strong Pumping Lemma for Context Free Languages

If L is context free, then
  There exist K and M (with $M \geq K$) such that
    For all strings w, where |w| > K,
      (Since true for all such w, it must be true for any paricular one, so you pick w)
      (Hint: describe w in terms of K or M)

      there exist u, v, x, y, z such that w = uvxyz and |vy| > 0, and
                                                |vxy| ≤ M, and
                                                for all $n \geq 0$, uv^nxy^nz is in L.

We need to **pick w**, then show that there are no values for uvxyz that satisfy all the above criteria. To do that, we just need to focus on possible values for v and y, the pumpable parts. So we **show that all possible picks for v and y violate at least one of the criteria**.

Write out a single string, w (in terms of K or M) **Divide w into regions**.

For each possibility for v and y (described in terms of the regions defined above), find some value n such that uv^nxy^nz is not in L. Almost always, the easiest values are 0 (pumping out) or 2 (pumping in). Your value for n may differ for different cases.

v	y	n	why the resulting string is not in L
[1]			
[2]			
[3]			
[4]			
[5]			
[6]			
[7]			
[8]			
[9]			
[10]			

Convince the reader that there are no other cases.

Q. E. D.

A Pumping Lemma Proof in Full Detail

Proof that $L = \{a^n b^n c^n : n \geq 0\}$ is not context free.

Suppose L is context free. The context free pumping lemma applies to L. Let M be the number from the pumping lemma. Choose w = $a^M b^M c^M$. Now $w \in L$ and $|w| > M \geq K$. From the pumping lemma, for all strings w, where $|w| > K$, there exist u, v, x, y, z such that w = uvxyz and $|vy| > 0$, and $|vxy| \leq M$, and for all $n \geq 0$, $uv^n xy^n z$ is in L. There are two main cases:

1. Either v or y contains two or more different types of symbols ("a", "b" or "c"). In this case, uv^2xy^2z is not of the form a*b*c* and hence $uv^2xy^2z \notin L$.
2. Neither v nor y contains two or more different types of symbols. In this case, vy may contain at most two types of symbols. The string uv^0xy^0z will decrease the count of one or two types of symbols, but not the third, so $uv^0xy^0z \notin L$

Cases 1 and 2 cover all the possibilities. Therefore, regardless of how w is partitioned, there is some $uv^n xy^n z$ that is not in L. Contradiction. Therefore L is not context free.

Note: the underlined parts of the above proof is "boilerplate" that can be reused. A complete proof should have this text or something equivalent.

Context-Free Languages Over a Single-Letter Alphabet

Theorem: Any context-free language over a single-letter alphabet is regular.
Examples:

L = $\{a^n b^n\}$
L′ = $\{a^n a^n\}$
= $\{a^{2n}\}$
= $\{w \in \{a\}^* : |w| \text{ is even}\}$

L = $\{ww^R : w \in \{a, b\}^*\}$
L′ = $\{ww^R : w \in \{a\}^*\}$
= $\{ww: w \in \{a\}^*\}$
= $\{w \in \{a\}^* : |w| \text{ is even}\}$

L = $\{a^n b^m : n, m \geq 0 \text{ and } n \neq m\}$
L′ = $\{a^n a^m : n, m \geq 0 \text{ and } n \neq m\}$
=

Proof: See Parikh's Theorem

Another Language That Is Not Context Free

$L = \{a^n : n \geq 1 \text{ is prime}\}$

Two ways to prove that L is not context free:

1. Use the pumping lemma:

Choose a string $w = a^n$ such that n is prime and $n > K$.

w = aaaaaaaaaaaaaaaaaaaaaaa

u v x y z

Let $vy = a^p$ and $uxz = a^r$. Then $r + kp$ must be prime for all values of k. This can't be true, as we argued to show that L was not regular.

2. $|\Sigma_L| = 1$. So if L were context free, it would also be regular. But we know that it is not. So it is not context free either.

Using Pumping and Closure

$L = \{w \in \{a, b, c\}^* : w \text{ has an equal number of a's, b's, and c's}\}$

L is not context free.

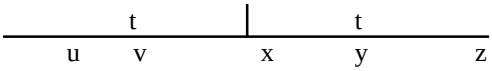
Try pumping: Let $w = a^K b^K c^K$

Now what?

Using Intersection with a Regular Language to Make Pumping Tractable

$L = \{tt : t \in \{a, b\}^*\}$

Let's try pumping: $|w| > K$



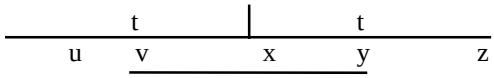
What if u is ε,
v is w,
x is ε,
y is w, and
z is ε

Then all pumping tells us is that $t^n t^n$ is in L.

L = {tt : t ∈ {a, b}* }

What if we let |w| > M, i.e. choose to pump the string $a^M b a^M b$:

Now v and y can't be t, since |vxy| ≤ M:

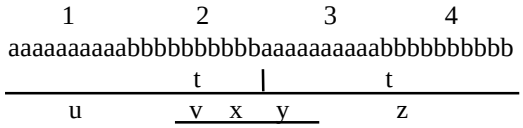


Suppose |v| = |y|. Now we have to show that repeating them makes the two copies of t different. But we can't.

L = {tt : t ∈ {a, b}* }

But let's consider L' = L ∩ a*b*a*b*

This time, we let |w| > 2M, and the number of both a's and b's in w >M:



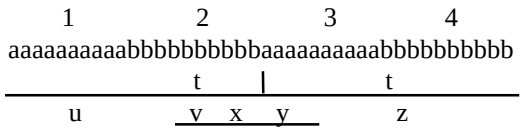
Now we use pumping to show that L' is not context free.

First, notice that if either v or y contains both a's and b's, then we immediately violate the rules for L' when we pump.

So now we know that v and y must each fall completely in one of the four marked regions.

L' = {tt : t ∈ {a, b}* } ∩ a*b*a*b*

|w| > 2M, and the number of both a's and b's in w >M:



Consider the combinations of (v, y):

(1,1)

(2,2)

(3,3)

(4,4)

(1,2)

(2,3)

(3,4)

(1,3)

(2,4)

(1,4)

The Context-Free Languages Are Not Closed Under Intersection

Proof: (by counterexample)

Consider $L = \{a^n b^n c^n : n \geq 0\}$

L is not context-free.

Let $L_1 = \{a^n b^n c^m : n, m \geq 0\}$ /* equal a's and b's
$L_2 = \{a^m b^n c^n : n, m \geq 0\}$ /* equal b's and c's

Both L_1 and L_2 are context-free.

But $L = L_1 \cap L_2$.

So, if the context-free languages were closed under intersection, L would have to be context-free. But it isn't.

The Context-Free Languages Are Not Closed Under Complementation

Proof: (by contradiction)

By definition:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Since the context-free languages are closed under union, if they were also closed under complementation, they would necessarily be closed under intersection. But we just showed that they are not. Thus they are not closed under complementation.

The Deterministic Context-Free Languages Are Closed Under Complement

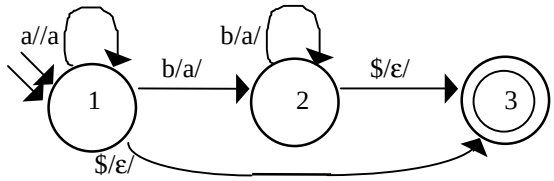
Proof:

Let L be a language such that L$ is accepted by the deterministic PDA M. We construct a deterministic PDA M' to accept (the complement of L)$, just as we did for FSMs:

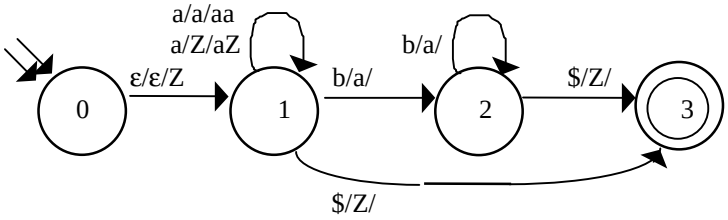
1. Initially, let M' = M.
2. M' is already deterministic.
3. Make M' simple. Why?
4. Complete M' by adding a dead state, if necessary, and adding all required transitions into it, including:
    - Transitions that are required to assure that for all input, stack combinations some transition can be followed.
    - If some state q has a transition on (ε, ε) and if it does not later lead to a state that does consume something then make a transiton on (ε, ε) to the dead state.
5. Swap final and nonfinal states.
6. Notice that M′ is still deterministic.

An Example of the Construction

$L = a^n b^n$ M accepts L$ (and is deterministic):

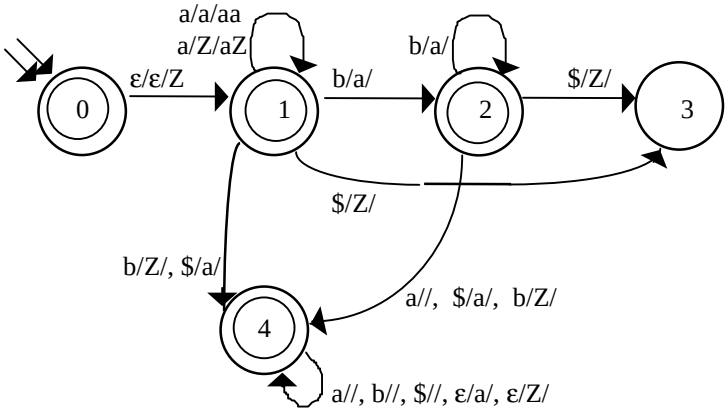


Set M = M'. Make M simple.



The Construction, Continued

Add dead state(s) and swap final and nonfinal states:



Issues: 1) Never having the machine die
2) $\neg(L\$) \neq (\neg L)\$$
3) Keeping the machine deterministic

Deterministic vs. Nondeterministic Context-Free Languages

Theorem: The class of deterministic context-free languages is a ***proper*** subset of the class of context-free languages.

Proof: Consider $L = \{a^n b^m c^p : m \neq n \text{ or } m \neq p\}$ L is context free (we have shown a grammar for it).

But L is not deterministic. If it were, then its complement L_1 would be deterministic context free, and thus certainly context free. But then

$L_2 = L_1 \cap a^*b^*c^*$ (a regular language)

would be context free. But

$L_2 = \{a^n b^n c^n : n \geq 0\}$, which we know is not context free.

Thus there exists at least one context-free language that is not deterministic context free.

Note that deterministic context-free languages are **not** closed under union, intersection, or difference.

Decision Procedures for CFLs & PDAs

Decision Procedures for CFLs

There are decision procedures for the following (G is a CFG):

- Deciding whether $w \in L(G)$.
- Deciding whether $L(G) = \emptyset$.
- Deciding whether L(G) is finite/infinite.

Such decision procedures usually involve conversions to Chomsky Normal Form or Greibach Normal Form. Why?

Theorem: For any context free grammar G, there exists a number n such that:

1. If $L(G) \neq \emptyset$, then there exists a $w \in L(G)$ such that $|w| < n$.
2. If L(G) is infinite, then there exists $w \in L(G)$ such that $n \leq |w| < 2n$.

There are **not** decision procedures for the following:

- Deciding whether $L(G) = \Sigma^*$.
- Deciding whether $L(G_1) = L(G_2)$.

If we could decide these problems, we could decide the halting problem. (More later.)

Decision Procedures for PDA's

There are decision procedures for the following (M is a PDA):

- Deciding whether $w \in L(M)$.
- Deciding whether $L(M) = \emptyset$.
- Deciding whether L(M) is finite/infinite.

Convert M to its equivalent PDA and use the corresponding CFG decision procedure. Why avoid using PDA's directly?

There are **not** decision procedures for the following:

- Deciding whether $L(M) = \Sigma^*$.
- Deciding whether $L(M_1) = L(M_2)$.

If we could decide these problems, we could decide the halting problem. (More later.)

Comparing Regular and Context-Free Languages

Regular Languages

- regular exprs.
  - or
- regular grammars
- recognize
- = DFSAs
- recognize
- minimize FSAs

- closed under:
  - concatenation
  - union
  - Kleene star
  - complement
  - intersection
- pumping lemma
- deterministic = nondeterministic

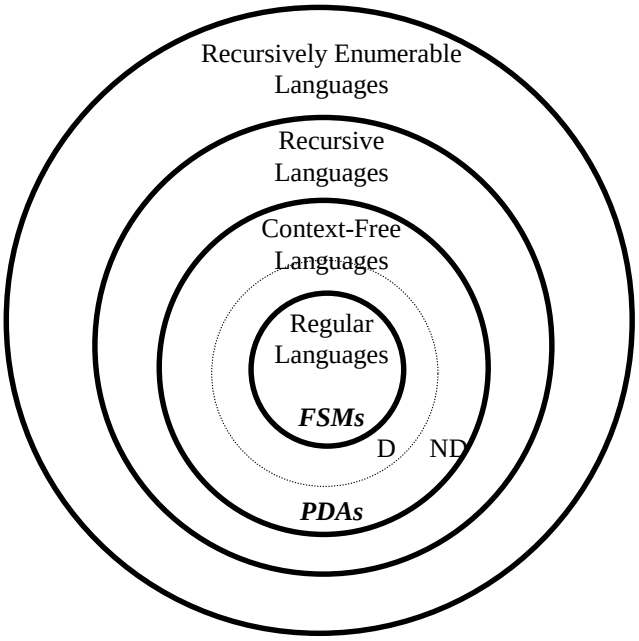
Context-Free Languages

- context-free grammars

- parse
- = NDPDAs
- parse
- find deterministic grammars
- find efficient parsers
- closed under:
  - concatenation
  - union
  - Kleene star

- intersection w/ reg. langs
- pumping lemma
- deterministic ≠ nondeterministic

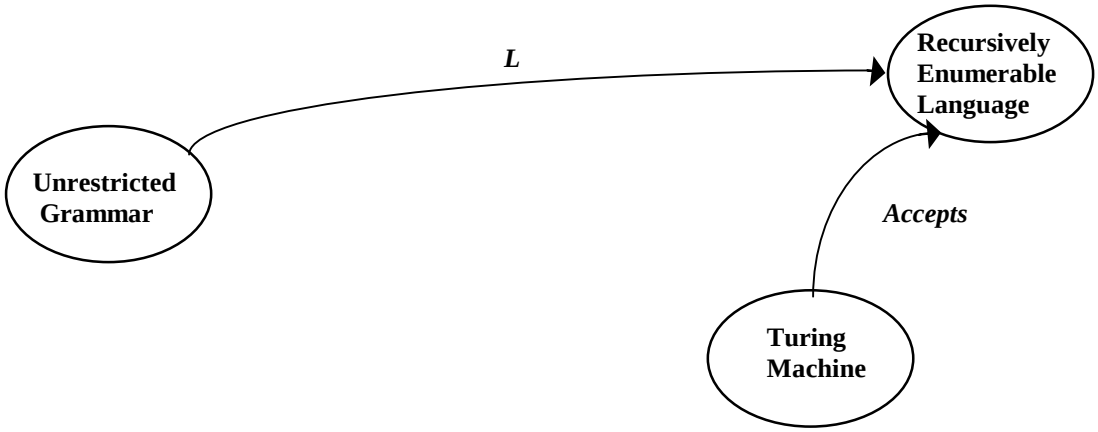
Languages and Machines



Turing Machines

Read K & S 4.1.
Do Homework 17.

Grammars, Recursively Enumerable Languages, and Turing Machines

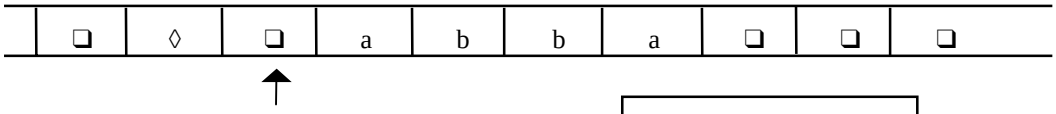


Turing Machines

Can we come up with a new kind of automaton that has two properties:

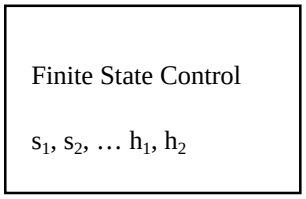
- powerful enough to describe all computable things
  unlike FSMs and PDAs
- simple enough that we can reason formally about it
  like FSMs and PDAs
  unlike real computers

Turing Machines



At each step, the machine may:

- go to a new state, and
- either
  - write on the current square, or
  - move left or right



A Formal Definition

A Turing machine is a quintuple $(K, \Sigma, \delta, s, H)$:

K is a finite set of states;
Σ is an alphabet, containing at least ❑ and ◊, but not $\rightarrow$ or $\leftarrow$;
$s \in K$ is the initial state;
$H \subseteq K$ is the set of halting states;
δ is a function from:

$(K - H)$	$\times$	Σ	to	K	$\times$	$(\Sigma \cup \{\rightarrow, \leftarrow\})$
non-halting state	$\times$	input symbol		state	$\times$	action (write or move)

such that

(a) if the input symbol is ◊, the action is $\rightarrow$, and
(b) ◊ can never be written .

Notes on the Definition

1. The input tape is infinite to the right (and full of ❑), but has a wall to the left. Some definitions allow infinite tape in both directions, but it doesn't matter.

2. δ is a function, not a relation. So this is a definition for deterministic Turing machines.

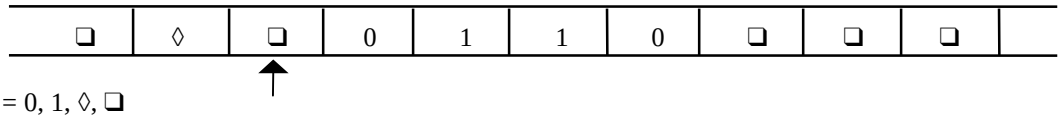
3. δ must be defined for all state, input pairs unless the state is a halt state.

4. Turing machines do not necessarily halt (unlike FSM's). Why? To halt, they must enter a halt state. Otherwise they loop.

5. Turing machines generate output so they can actually compute functions.

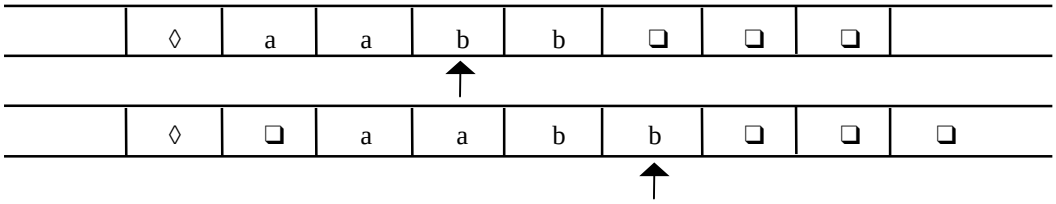
A Simple Example

A Turing Machine Odd Parity Machine:



Σ = 0, 1, ◊, ❑
s =
H =
δ =

Formalizing the Operation



(1)

(2)

A **configuration** of a Turing machine
M = (K, Σ, δ, s, H) is a member of

K	×	◊Σ*	×	(Σ*(Σ - {❑})) ∪ ε
state		input up to scanned square		input after scanned square

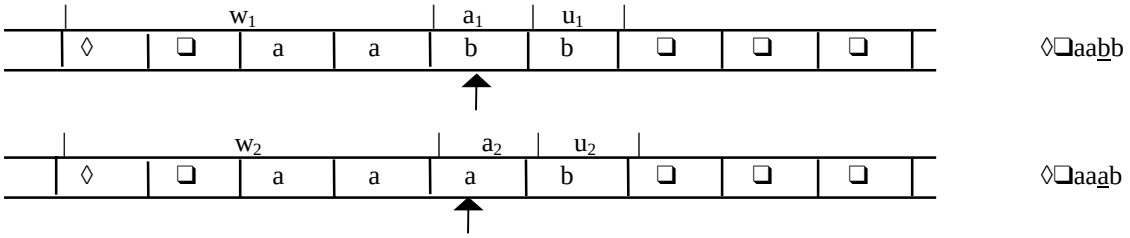
The input after the scanned square may be empty, but it may not end with a blank. We assume the entire tape to the right of the input is filled with blanks.

(1) (q, ◊aab, b) = (q, ◊aa<u>b</u>b)
(2) (h, ◊❑aabb, ε) = (h, ◊❑aab<u>b</u>) a halting configuration

Yields

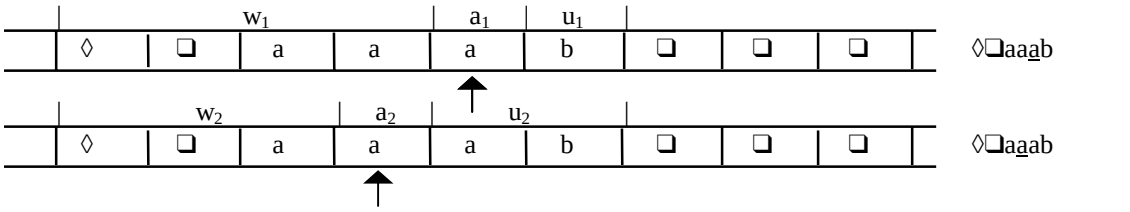
$(q_1, w_1\underline{a_1}u_1) \vdash_M (q_2, w_2\underline{a_2}u_2)$, a_1 and $a_2 \in \Sigma$, iff $\exists\ b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, $\delta(q_1, a_1) = (q_2, b)$ and either:

(1) $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$, and $a_2 = b$ (rewrite without moving the head)

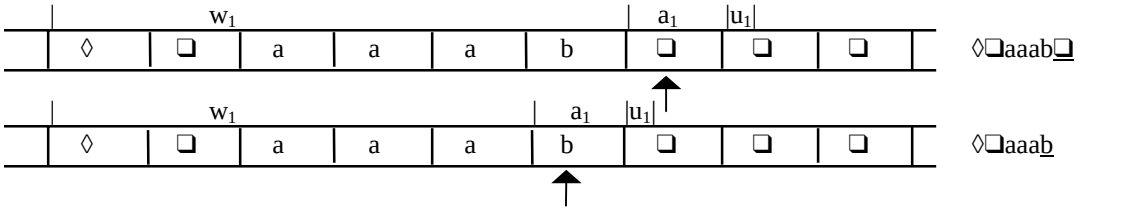


Yields, Continued

(2) $b = \leftarrow$, $w_1 = w_2a_2$, and either
(a) $u_2 = a_1u_1$, if $a_1 \neq ❑$ or $u_1 \neq \varepsilon$,



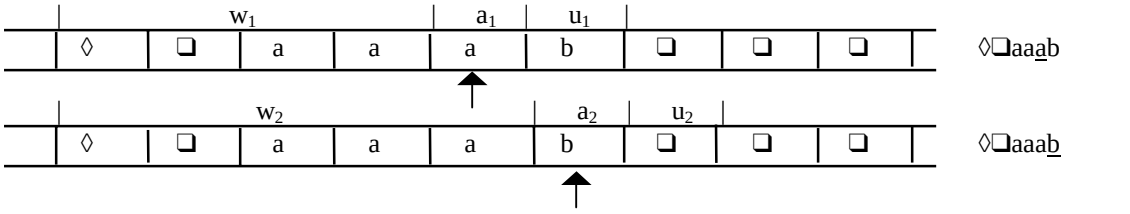
or (b) $u_2 = \varepsilon$, if $a_1 = ❑$ and $u_1 = \varepsilon$



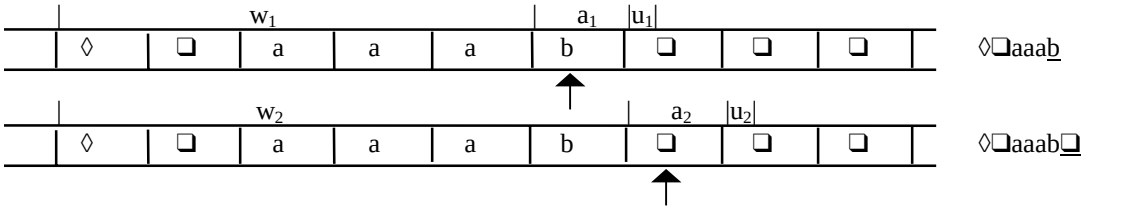
If we scan left off the first square of the blank region, then drop that square from the configuration.

Yields, Continued

(3) $b = \rightarrow$, $w_2 = w_1a_1$, and either
(a) $u_1 = a_2u_2$



or (b) $u_1 = u_2 = \varepsilon$ and $a_2 = ❑$



If we scan right onto the first square of the blank region, then a new blank appears in the configuration.

Yields, Continued

For any Turing machine M, let $|-_M^*$ be the reflexive, transitive closure of $|-_M$.

Configuration C_1 **yields** configuration C_2 if

$C_1 \ |-_M^* \ C_2$.

A **computation** by M is a sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that

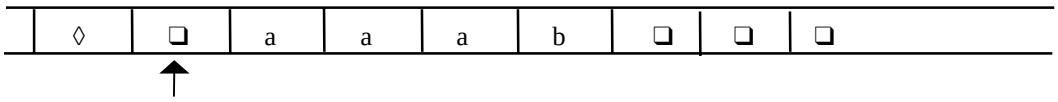
$C_0 \ |-_M \ C_1 \ |-_M \ C_2 \ |-_M \ \ldots \ |-_M \ C_n$.

We say that the computation is of **length** n or that it has n **steps**, and we write

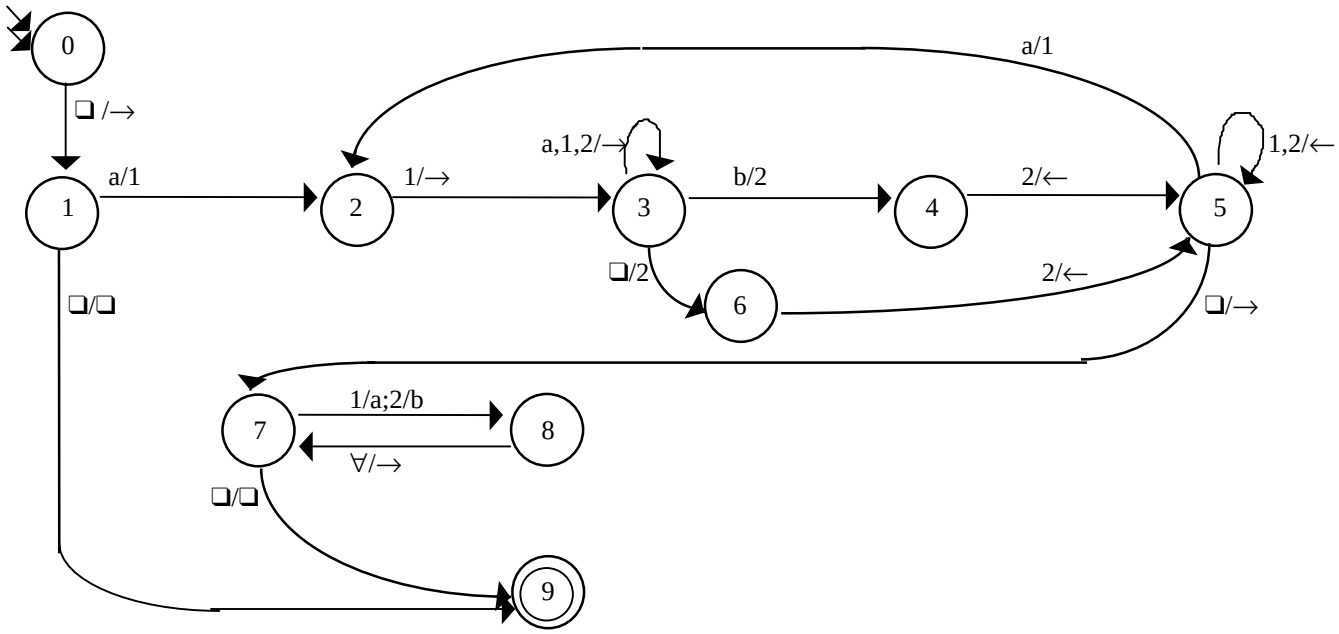
$C_0 \ |-_M^n \ C_n$

A Context-Free Example

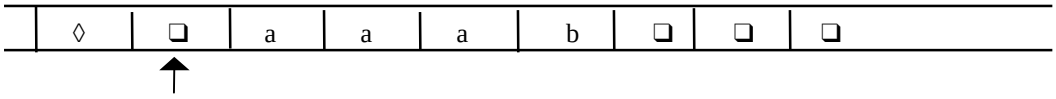
M takes a tape of a's then b's, possibly with more a's, and adds b's as required to make the number of b's equal the number of a's.



K = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Σ = a, b, ◊, ❑, 1, 2
s = 0 H = {9} δ =



An Example Computation



(0, ◊❑aaab) $|-_M$
(1, ◊❑aaab) $|-_M$
(2, ◊❑1aab) $|-_M$
(3, ◊❑1aab) $|-_M$
(3, ◊❑1aab) $|-_M$
(3, ◊❑1aab) $|-_M$
(4, ◊❑1aa2) $|-_M$
…

Notes on Programming

The machine has a strong procedural feel.

It's very common to have state pairs, in which the first writes on the tape and the second move. Some definitions allow both actions at once, and those machines will have fewer states.

There are common idioms, like scan left until you find a blank.

Even a very simple machine is a nuisance to write.

A Notation for Turing Machines

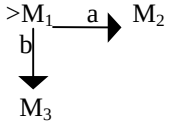
(1) Define some basic machines

- Symbol writing machines
  For each $a \in \Sigma - \{\lozenge\}$, define M_a, written just a, = $(\{s, h\}, \Sigma, \delta, s, \{h\})$,
  for each $b \in \Sigma - \{\lozenge\}$, $\delta(s, b) = (h, a)$
  $\delta(s, \lozenge) = (s, \rightarrow)$
  Example:
  a writes an a

- Head moving machines
  For each $a \in \{\leftarrow, \rightarrow\}$, define M_a, written R($\rightarrow$) and L($\leftarrow$):
  for each $b \in \Sigma - \{\lozenge\}$, $\delta(s, b) = (h, a)$
  $\delta(s, \lozenge) = (s, \rightarrow)$
  Examples:
  R moves one square to the right
  aR writes an a and then moves one square to the right.

A Notation for Turing Machines, Cont'd

(2) The rules for combining machines: as with FSMs



- Start in the start state of M_1.
- Compute until M_1 reaches a halt state.
- Examine the tape and take the appropriate transition.
- Start in the start state of the next machine, etc.
- Halt if any component reaches a halt state and has no place to go.
- If any component fails to halt, then the entire machine may fail to halt.

Shorthands

M_1 —a→ M_2 (a and b) becomes M_1 —a, b→ M_2

M_1 —all elems of Σ→ M_2 becomes M_1 → M_2 (or M_1M_2)

MM becomes M^2

M_1 —all elems of Σ except a→ M_2 becomes M_1 —x ≠ a→ M_2

and x takes on the value of the current square

M_1 —a, b→ M_2 becomes M_1 —x = a, b→ M_2

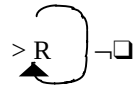
and x takes on the value of the current square

M —x ? y→ M_2

if x = y then take the transition

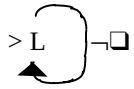
e.g., > —x ≠ ❑→ Rx if the current square is not blank, go right and copy it.

Some Useful Machines



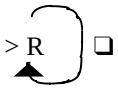
find the first blank square to the right of the current square

$R_{❑}$



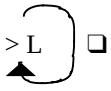
find the first blank square to the left of the current square

$L_{❑}$



find the first nonblank square to the right of the current square

$R_{¬❑}$



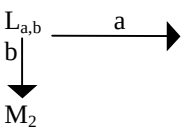
find the first nonblank square to the left of the current square

$L_{¬❑}$

More Useful Machines

L_a find the first occurrence of a to the left of the current square

$R_{a,b}$ find the first occurrence of a or b to the right of the current square



find the first occurrence of a or b to the left of the current square, then go to M_1 if the detected character is a; go to M_2 if the detected character is b

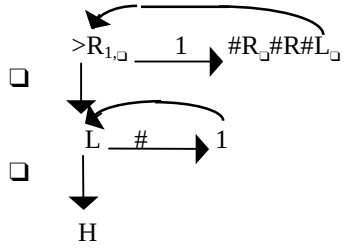
$L_{x=a,b}$ find the first occurrence of a or b to the left of the current square and set x to the value found

$L_{x=a,b}Rx$ find the first occurrence of a or b to the left of the current square, set x to the value found, move one square to the right, and write x (a or b)

An Example

Input: ◊❑w w ∈ {1}*
Output: ◊❑w^3

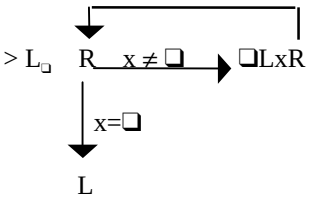
Example: ◊ ❑111❑❑❑❑❑❑❑❑❑❑❑❑



A Shifting Machine $S_{\leftarrow}$

Input: ❑❑w❑
Output: ❑w❑

Example: ❑❑abba❑❑❑❑❑❑❑❑❑❑❑❑



Computing with Turing Machines

Read K & S 4.2.
Do Homework 18.

Turing Machines as Language Recognizers

Convention: We will write the input on the tape as:

◊❑w❑ , w contains no ❑s

The initial configuration of M will then be:

(s, ◊❑w)

A recognizing Turing machine M must have two halting states: y and n
Any configuration of M whose state is:

y is an accepting configuration
n is a rejecting configuration

Let Σ_0, the input alphabet, be a subset of Σ_M-{❑,◊}
Then M **decides** a language $L \subseteq \Sigma_0^*$ iff for any string

$w \in \Sigma_0^*$ it is true that:

if $w \in L$ then M accepts w, and
if $w \notin L$ then M rejects w.

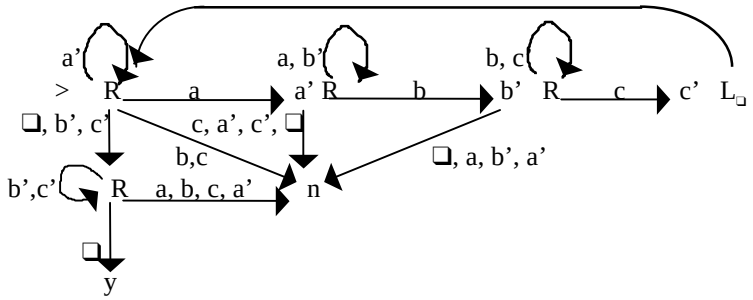
A language L is **recursive** if there is a Turing machine M that decides it.

A Recognition Example

$L = \{a^n b^n c^n : n \geq 0\}$

Example: ◊❑aabbcc❑❑❑❑❑❑❑❑❑

Example: ◊❑aaccb❑❑❑❑❑❑❑❑❑

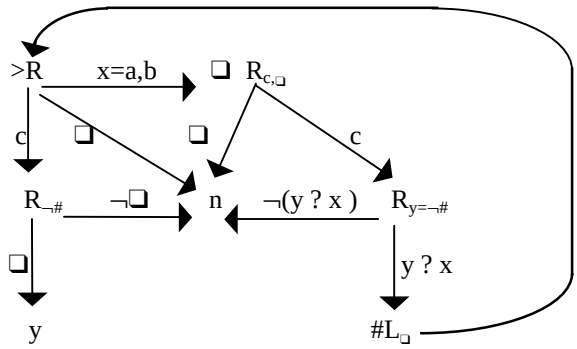


Another Recognition Example

L = {wcw : w ∈ {a, b}*}

Example: ◊❑abbcabb❑❑❑

Example: ◊❑acabb❑❑❑



Do Turing Machines Stop?

FSMs Always halt after n steps, where n is the length of the input. At that point, they either accept or reject.

PDAs Don't always halt, but there is an algorithm to convert any PDA into one that does halt.

Turing machines Can do one of three things:

(1) Halt and accept
(2) Halt and reject
(3) Not halt

And now there is no algorithm to determine whether a given machine always halts.

Computing Functions

Let $\Sigma_0 \subseteq \Sigma - \{◊, ❑\}$ and let $w \in \Sigma_0^*$

Convention: We will write the input on the tape as: ◊❑w❑

The initial configuration of M will then be: (s, ◊❑w)

Define M(w) = y iff:

- M halts if started in the input configuration,
- the tape of M when it halts is ◊❑y❑, and
- $y \in \Sigma_0^*$

Let f be any function from Σ_0^* to Σ_0^*.

We say that M **computes** f if, for all $w \in \Sigma_0^*$, M(w) = f(w)

A function f is **recursive** if there is a Turing machine M that computes it.

Example of Computing a Function

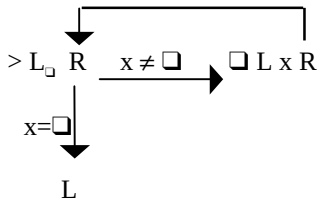
f(w) = ww

Input: ◊❑w❑❑❑❑❑❑ Output: ◊❑ww❑

Define the copy machine C: ◊❑w❑❑❑❑❑❑ → ◊❑w❑w❑

Remember the $S_{\leftarrow}$ machine:

◊❑w❑w❑ → ◊❑ww❑



Then the machine to compute f is just >C S $L_{❑\leftarrow}$

Computing Numeric Functions

We say that a Turing machine M computes a function f from N^k to N provided that

$$num(M(n_1;n_2;\ldots n_k)) = f(num(n_1), \ldots num(n_k))$$

Example: Succ(n) = n + 1

We will represent n in binary. So $n \in 0 \cup 1\{0,1\}^*$

Input: ◊❑n❑❑❑❑❑❑ Output: ◊❑n+1❑
◊❑1111❑❑❑❑ Output: ◊❑10000❑

Why Are We Working with Our Hands Tied Behind Our Backs?

Turing machines are more powerful than any of the other formalisms we have studied so far.

Turing machines are a **lot** harder to work with than all the real computers we have available.

Why bother?

The very simplicity that makes it hard to program Turing machines makes it possible to reason formally about what they can do. If we can, once, show that anything a real computer can do can be done (albeit clumsily) on a Turing machine, then we have a way to reason about what real computers can do.

Recursively Enumerable and Recursive Languages

Read K & S 4.5.

Recursively Enumerable Languages

Let Σ_0, the input alphabet to a Turing machine M, be a subset of $\Sigma_M - \{\square, \lozenge\}$

Let $L \subseteq \Sigma_0^*$.

M **semidecides** L iff

for any string $w \in \Sigma_0^*$,

$w \in L \Rightarrow$ M halts on input w

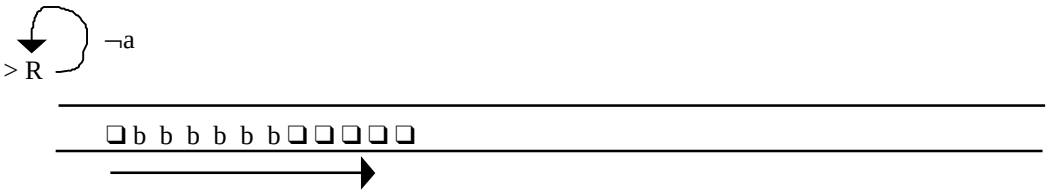
$w \notin L \Rightarrow$ M does not halt on input w

$M(w) = \uparrow$

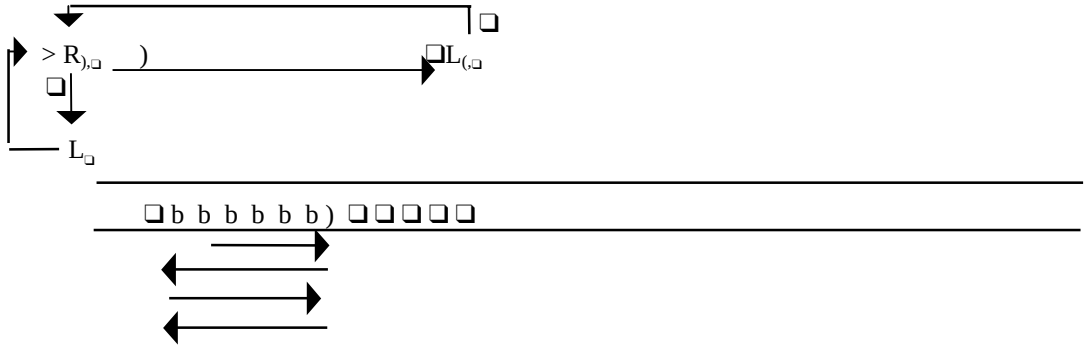
L is **recursively enumerable** iff there is a Turing machine that semidecides it.

Examples of Recursively Enumerable Languages

L = {w ∈ {a, b}* : w contains at least one a}



L = {w ∈ {a, b, (,) }* : w contains at least one set of balanced parentheses}



Recursively Enumerable Languages that Aren't Also Recursive

A Real Life Example:

L = {w ∈ {friends} : w will answer the message you've just sent out}

Theoretical Examples

L = {Turing machines that halt on a blank input tape}

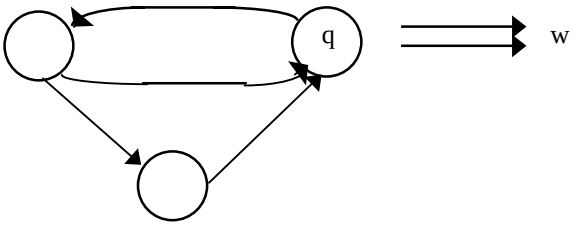
Theorems with valid proofs.

Why Are They Called Recursively Enumerable Languages?

Enumerate means list.

We say that Turing machine M **enumerates** the language L iff, for some fixed state q of M,

$L = \{w : (s, \lozenge\underline{\square}) \mid\!-_M^* (q, \lozenge\square w)\}$



A language is **Turing-enumerable** iff there is a Turing machine that enumerates it.

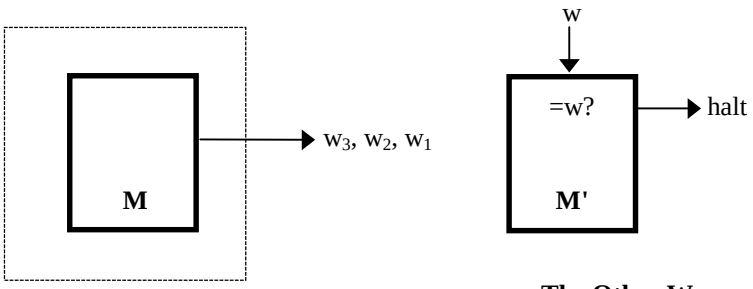
Note that q is not a halting state. It merely signals that the current contents of the tape should be viewed as a member of L.

Recursively Enumerable and Turing Enumerable

Theorem: A language is recursively enumerable iff it is Turing-enumerable.

Proof that Turing-enumerable implies RE: Let M be the Turing machine that enumerates L. We convert M to a machine M' that semidecides L:

1. Save input w.
2. Begin enumerating L. Each time an element of L is enumerated, compare it to w. If they match, accept.



The Other Way

Proof that RE implies Turing-enumerable:

If $L \subseteq \Sigma^*$ is a recursively enumerable language, then there is a Turing machine M that semidecides L.

A procedure to enumerate all elements of L:

Enumerate all $w \in \Sigma^*$ lexicographically.

e.g., ε, a, b, aa, ab, ba, bb, …

As each string w_i is enumerated:

1. Start up a copy of M with w_i as its input.
2. Execute one step of each M_i initiated so far, excluding only those that have previously halted.
3. Whenever an M_i halts, output w_i.

ε [1]					
ε [2]	a [1]				
ε [3]	a [2]	b [1]			
ε [4]	a [3]	b [2]	aa [1]		
ε [5]	a [4]	b [3]	aa [2]	ab [1]	
ε [6]	a [5]		aa [3]	ab [2]	ba [1]

Every Recursive Language is Recursively Enumerable

If L is recursive, then there is a Turing machine that decides it.

From M, we can build a new Turing machine M' that semidecides L:

1. Let n be the reject (and halt) state of M.
2. Then add to δ'
   ((n, a), (n, a)) for all a ∈ Σ

What about the other way around?
Not true. There are recursively enumerable languages that are not recursive.

The Recursive Languages Are Closed Under Complement

Proof: (by construction) If L is recursive, then there is a Turing machine M that decides L.

We construct a machine M' to decide $\overline{L}$ by taking M and swapping the roles of the two halting states y and n.

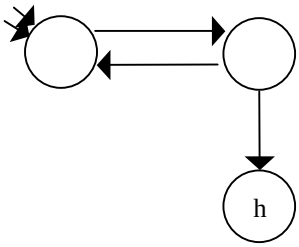
M: M':

This works because, by definition, M is

- deterministic
- complete

Are the Recursively Enumerable Languages Closed Under Complement?

M: M':



Lemma: There exists at least one language L that is recursively enumerable but not recursive.

Proof that M' doesn't exist: Suppose that the RE languages were closed under complement. Then if L is RE, $\overline{L}$ would be RE. If that were true, then $\overline{L}$ would also be recursive because we could construct M to decide it:

1. Let T_1 be the Turing machine that semidecides L.
2. Let T_2 be the Turing machine that semidecides $\overline{L}$.
3. Given a string w, fire up both T_1 and T_2 on w. Since any string in Σ* must be in either L or $\overline{L}$, one of the two machines will eventually halt. If it's T_1, accept; if it's T_2, reject.

But we know that there is at least one RE language that is not recursive. Contradiction.

Recursive and RE Languages

Theorem: A language is recursive iff both it and its complement are recursively enumerable.

Proof:

- L recursive implies L and ¬L are RE: Clearly L is RE. And, since the recursive languages are closed under complement, ¬L is recursive and thus also RE.
- L and ¬L are RE implies L recursive: Suppose L is semidecided by M1 and ¬L is semidecided by M2. We construct M to decide L by using two tapes and simultaneously executing M1 and M2. One (but not both) must eventually halt. If it's M1, we accept; if it's M2 we reject.

Lexicographic Enumeration

We say that M **lexicographically enumerates** L if M enumerates the elements of L in lexicographic order. A language L is **lexicographically Turing-enumerable** iff there is a Turing machine that lexicographically enumerates it.

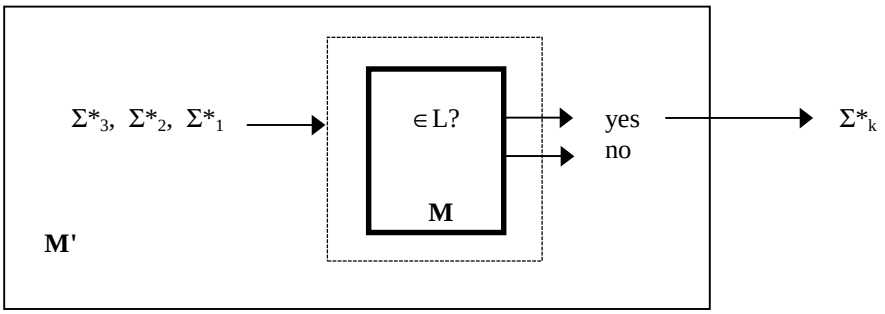
Example: $L = \{a^n b^n c^n\}$

Lexicographic enumeration:

Proof

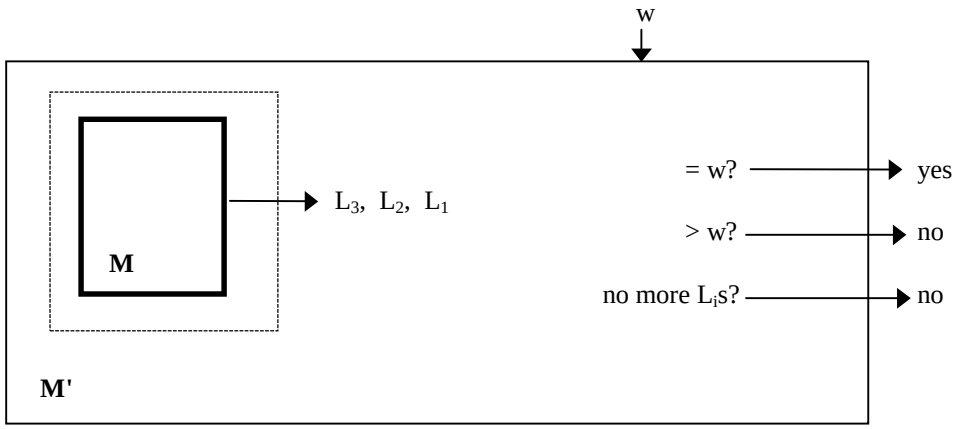
Theorem: A language is recursive iff it is lexicographically Turing-enumerable.

Proof that recursive implies lexicographically Turing enumerable: Let M be a Turing machine that decides L. Then M' lexicographically generates the strings in Σ* and tests each using M. It outputs those that are accepted by M. Thus M' lexicographically enumerates L.



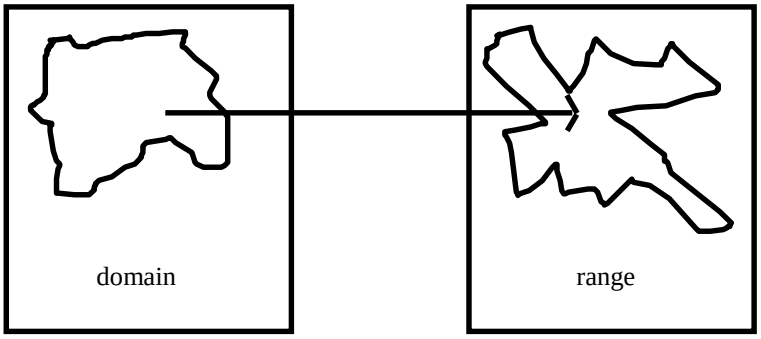
Proof, Continued

Proof that lexicographically Turing enumerable implies recursive: Let M be a Turing machine that lexicographically enumerates L. Then, on input w, M' starts up M and waits until either M generates w (so M' accepts), M generates a string that comes after w (so M' rejects), or M halts (so M' rejects). Thus M' decides L.



Partially Recursive Functions

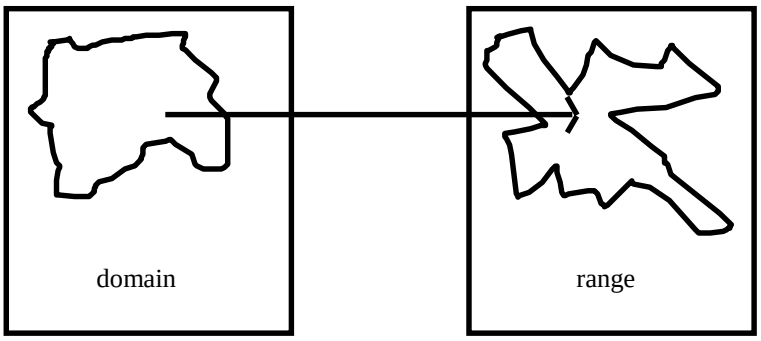
	Languages	Functions
Tm always halts	**recursive**	**recursive**
Tm halts if yes	**recursively enumerable**	**?**



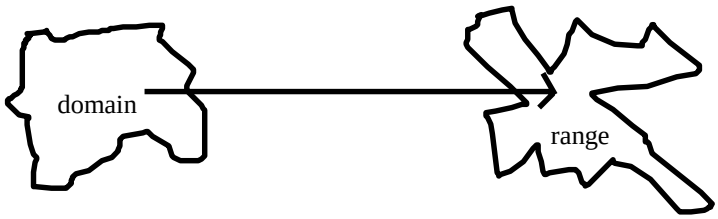
Suppose we have a function that is not defined for all elements of its domain.

Example: f: N $\rightarrow$ N, f(n) = n/2

Partially Recursive Functions



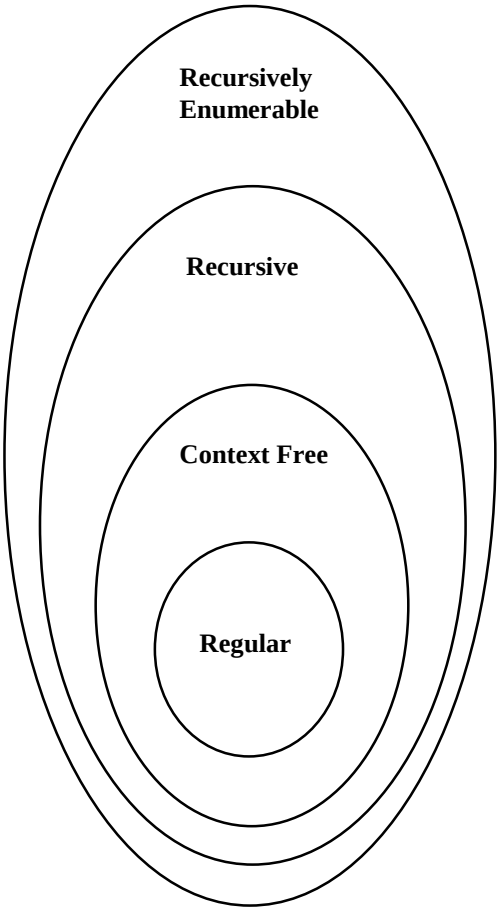
One solution: Redefine the domain to be exactly those elements for which f is defined:



But what if we don't know? What if the domain is not a recursive set (but it is recursively enumerable)? Then we want to define the domain as some larger, recursive set and say that the function is partially recursive. There exists a Turing machine that halts if given an element of the domain but does not halt otherwise.

Language
Summary

IN		OUT
Semidecidable Enumerable Unrestricted grammar	**Recursively Enumerable**	
Decision procedure Lexicicographically enumerable Complement is recursively enumer.	**Recursive**	Diagonalization Reduction
CF grammar PDA Closure	**Context Free**	Pumping Closure
Regular expression FSM Closure	**Regular**	Pumping Closure



Turing Machine Extensions

Read K & S 4.3.1, 4.4.
Do Homework 19.

Turing Machine Definitions

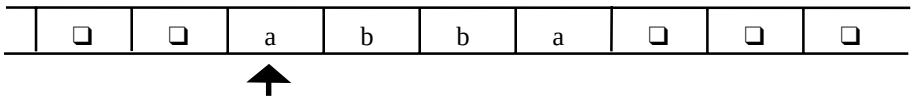
An alternative definition of a Turing machine:
(K, Σ, Γ, δ, s, H):

Γ is a finite set of allowable tape symbols. One of these is ❑.

Σ is a subset of Γ not including ❑, the input symbols.

δ is a function from:

$K \times \Gamma$ to $K \times (\Gamma - \{❑\}) \times \{\leftarrow, \rightarrow\}$
state, tape symbol, L or R



Example transition: ((s, a), (s, b, →))

Do these Differences Matter?

Remember the goal:

Define a device that is:
- powerful enough to describe all computable things,
- simple enough that we can reason formally about it

Both definitions are simple enough to work with, although details may make specific arguments easier or harder.

But, do they differ in their power?

Answer: No.

Consider the differences:
- One way or two way infinite tape: we're about to show that we can simulate two way infinite with ours.
- Rewrite and move at the same time: just affects (linearly) the number of moves it takes to solve a problem.

Turing Machine Extensions

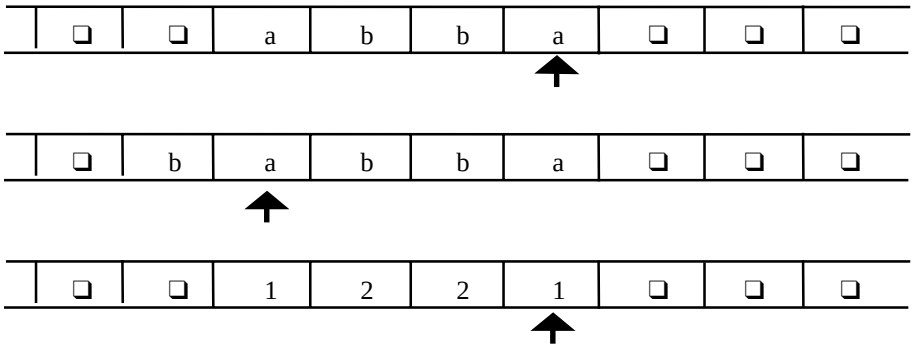
In fact, there are lots of extensions we can make to our basic Turing machine model. They may make it easier to write Turing machine programs, but none of them increase the power of the Turing machine because:

We can show that every extended machine has an equivalent basic machine.

We can also place a bound on any change in the complexity of a solution when we go from an extended machine to a basic machine.

Some possible extensions:
- Multiple tapes
- Two-way infinite tape
- Multiple read heads
- Two dimensional "sheet" instead of a tape
- Random access machine
- Nondeterministic machine

Multiple Tapes



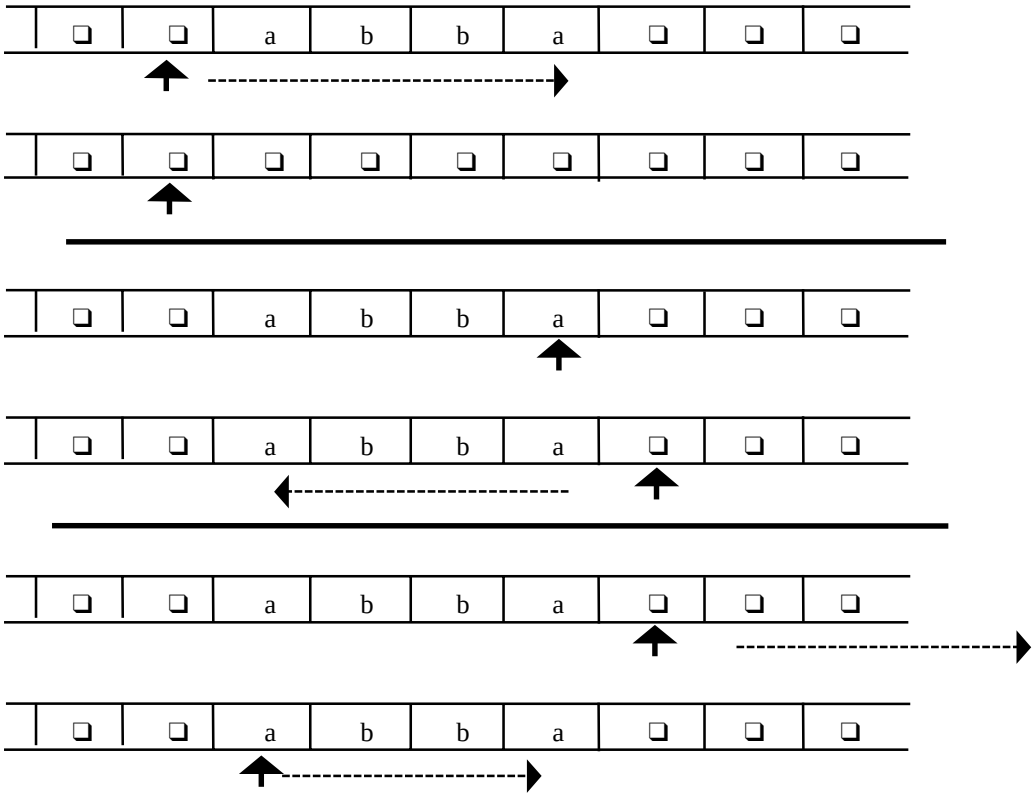
The transition function for a k-tape Turing machine:

$((K\text{-}H) , \Sigma_1$ to $(K, \Sigma_{1'} \cup \{\leftarrow, \rightarrow\}$
$, \Sigma_2$ $, \Sigma_{2'} \cup \{\leftarrow, \rightarrow\}$
$, .$ $, .$
$, .$ $, .$
$, \Sigma_k)$ $, \Sigma_{k'} \cup \{\leftarrow, \rightarrow\})$

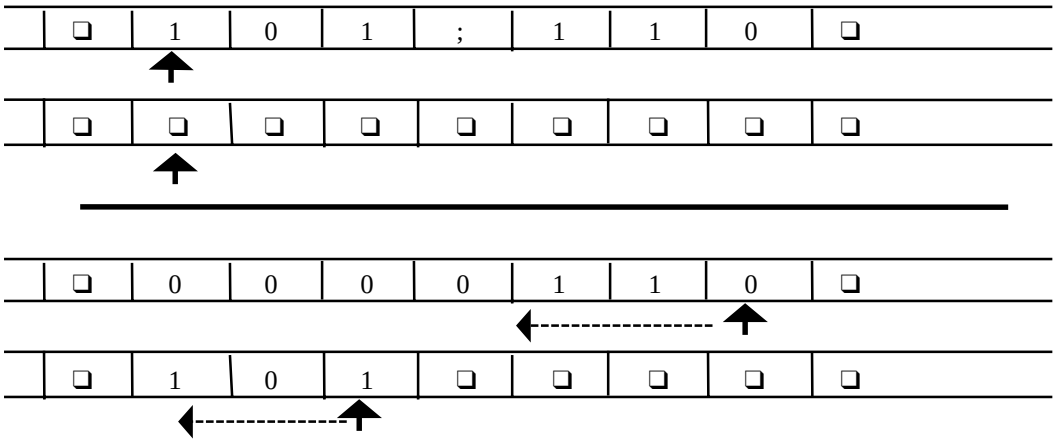
Input: input as before on tape 1, others blank
Output: output as before on tape 1, others ignored

An Example of a Two Tape Machine

Copying a string



Another Two Tape Example - Addition



Adding Tapes Adds No Power

Theorem: Let M be a k-tape Turing machine for some $k \geq 1$. Then there is a standard Turing machine M' where $\Sigma \subseteq \Sigma'$, and such that:

- For any input string x, M on input x halts with output y on the first tape iff M' on input x halts at the same halting state and with the same output on its tape.
- If, on input x, M halts after t steps, then M' halts after a number of steps which is $O(t \cdot (|x| + t))$.

Proof: By construction

◊	◊	❑	a	b	a	❑	❑	❑	❑
	0	0	1	0	0	0	0		
	◊	a	b	b	a	b	a		
	0	1	0	0	0	0	0		

Alphabet (Σ') of M' = $\Sigma \cup (\Sigma \times \{0, 1\})^k$

e.g., ◊, (◊, 0, ◊, 0), (❑, 0, a, 1)

The Operation of M'

◊	◊	❑	a	b	a	❑	❑	❑	❑
	0	0	1	0	0	0	0		
	◊	a	b	b	a	b	a		
	0	1	0	0	0	0	0		

1. Set up the multitrack tape:
   1) Shift input one square to right, then set up each square appropriately.
2. Simulate the computation of M until (if) M would halt: (start each step to the right of the divided tape)
   1) Scan left and store in the state the k-tuple of characters under the read heads. Move back right.
   2) Scan left and update each track as required by the transitions of M. Move back right.
      i) If necessary, subdivide a new square into tracks.
3. When M would halt, reformat the tape to throw away all but track 1, position the head correctly, then go to M's halt state.

How Many Steps Does M' Take?

Let: x be the input string, and
t be the number of steps it takes M to execute.

Step 1 (initialization) $O(|x|)$

Step 2 (computation)

Number of passes = t

Work at each pass: $2.1 = 2 \cdot$ (length of tape)
$= 2 \cdot (|x| + 2 + t)$
$2.2 = 2 \cdot (|x| + 2 + t)$

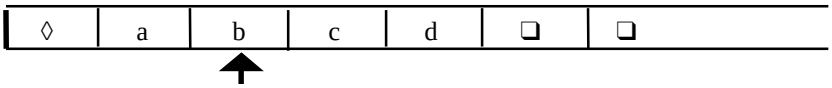
Total = $O(t \cdot (|x| + t))$

Step 3 (clean up) O(length of tape)

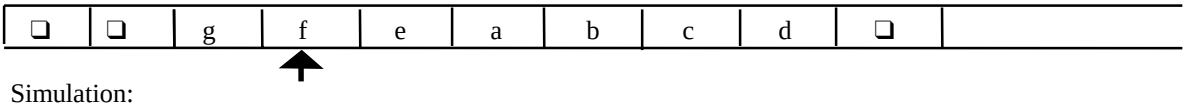
Total = $O(t \cdot (|x| + t))$

Two-Way Infinite Tape

Our current definition:

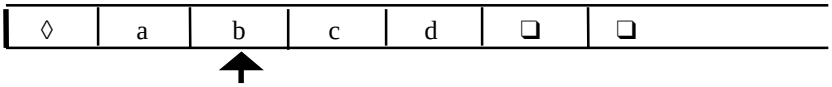


Proposed definition:



Simulation:

Track 1



Track 2

Simulating a PDA

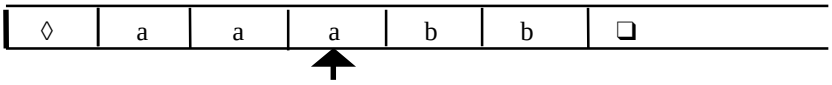
The components of a PDA:

- Finite state controller
- Input tape
- Stack

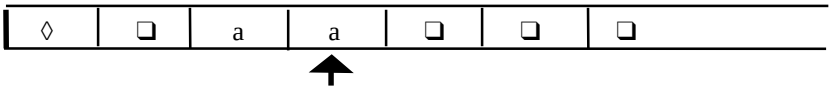
The simulation:

- Finite state controller:
- Input tape:
- Stack:

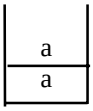
Track 1 (Input)



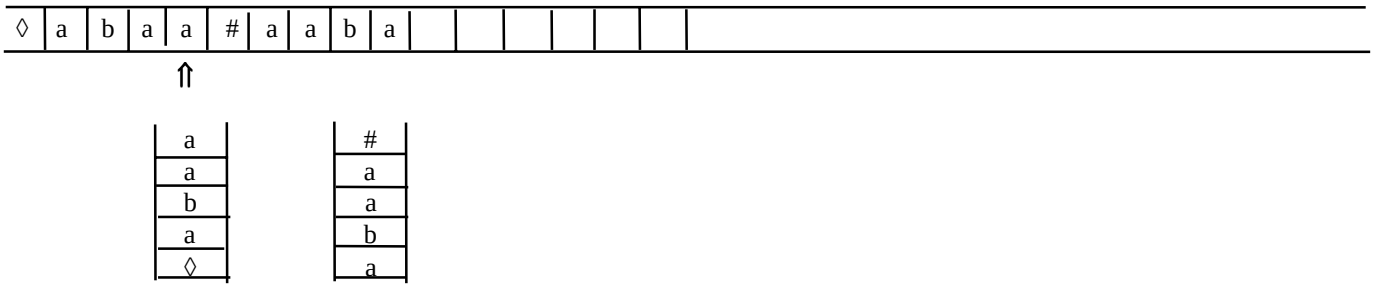
Track 2



Corresponding to



Simulating a Turing Machine with a PDA with Two Stacks



Random Access Turing Machines

A random access Turing machine has:

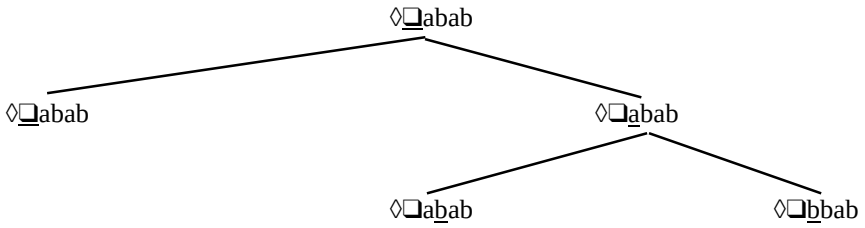
- a fixed number of registers
- a finite length program, composed of instructions with operators such as read, write, load, store, add, sub, jump
- a tape
- a program counter

Theorem: Standard Turing machines and random access Turing machines compute the same things. Furthermore, the number of steps it takes a standard machine is bounded by a polynomial in the number of steps it takes a random access machine.

Nondeterministic Turing Machines

A **nondeterministic** Turing machine is a quintuple (K, Σ, Δ, s, H)
where K, Σ, s, and H are as for standard Turing machines, and Δ is a ***subset*** of

$$((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$$



What does it mean for a nondeterministic Turing machine to compute something?

- Semidecides - at least one halts.
- Decides - ?
- Computes - ?

Nondeterministic Semideciding

Let M = (K, Σ, Δ, s, H) be a nondeterministic Turing machine. We say that M **accepts** an input

$w \in (\Sigma - \{◊, ❑\})^*$ iff

(s, ◊❑w) yields a least one accepting configuration.

We say that M **semidecides** a language

$L \subseteq (\Sigma - \{◊, ❑\})^*$ iff

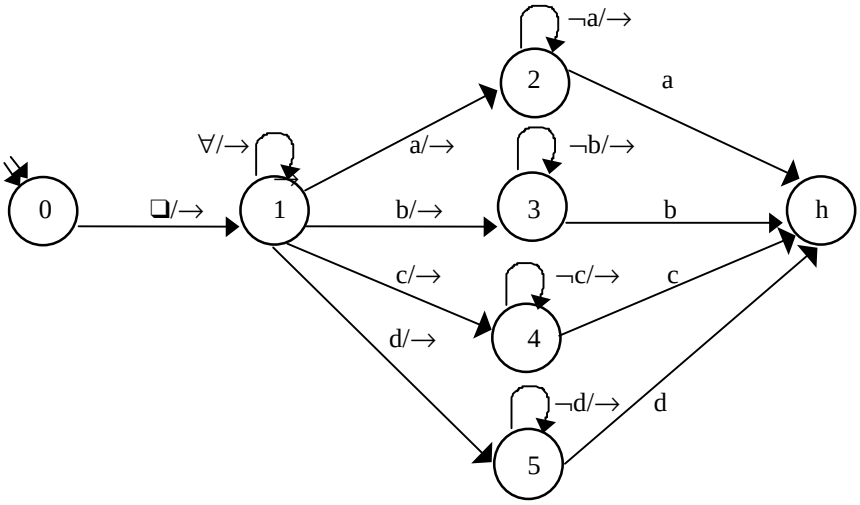
for all $w \in (\Sigma - \{◊, ❑\})^*$:

$w \in L$ iff

(s, ◊❑w) yields a least one halting configuration.

An Example

L = {w ∈ {a, b, c, d}* : there are two of at least one letter}



Nondeterministic Deciding and Computing

M **decides** a language L if, for all $w \in (\Sigma - \{\lozenge, \square\})^*$:
1. all of M's computations on w halt, and
2. $w \in L$ iff ***at least one*** of M's computations accepts.

M **computes** a function f if, for all $w \in (\Sigma - \{\lozenge, \square\})^*$:
1. all of M's computations halt, and
2. ***all*** of M's computations result in f(w)

Note that all of M's computations halt iff:

There is a natural number N, depending on M and w, such that there is no configuration C satisfying
$(s, \lozenge\underline{\square}w) \mid\!-_M^N C$.

An Example of Nondeterministic Deciding

$L = \{w \in \{0, 1\}^* : w \text{ is the binary encoding of a composite number}\}$

M decides L by doing the following on input w:

1. Nondeterministically choose two binary numbers $1 < p, q$, where $|p|$ and $|q| \leq |w|$, and write them on the tape, after w, separated by ;.

   $\lozenge\underline{\square}110011;111;1111\square\square$

2. Multiply p and q and put the answer, A, on the tape, in place of p and q.

   $\lozenge\underline{\square}110011;1011111\square\square$

3. Compare A and w. If equal, go to y. Else go to n.

Equivalence of Deterministic and Nondeterministic Turing Machines

Theorem: If a nondeterministic Turing machine M semidecides or decides a language, or computes a function, then there is a standard Turing machine M' semideciding or deciding the same language or computing the same function.

Note that while nondeterminism doesn't change the computational power of a Turing Machine, it can exponentially increase its speed!

Proof: (by construction)
For semideciding: We build M', which runs through all possible computations of M. If one of them halts, M' halts

Recall the way we did this for FSMs: simulate being in a combination of states.

Will this work here?

What about: Try path 1. If it accepts, accept. Else
Try path 2. If it accepts, accept. Else
•
•

The Construction

At any point in the operation of a nondeterministic machine M, the maximum number of branches is

$r = \underset{\text{states}}{|K|} \cdot \underset{\text{actions}}{(|\Sigma| + 2)}$

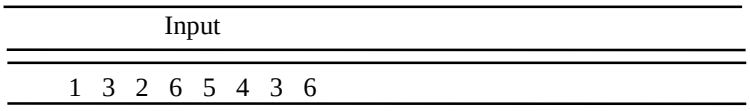
So imagine a table:

	1	2	3		r
(q1,σ1)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)
(q1,σ2)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)
(q1,σn)					
(q2,σ1)					
(q|K|,σn)					

Note that if, in some configuration, there are not r different legal things to do, then some of the entries on that row will repeat.

The Construction, Continued

M_d: (suppose r = 6)



M_d chooses its 1st move from column 1
M_d chooses its 2nd move from column 3
M_d chooses its 3rd move from column 2

-
-

until there are no more numbers on Tape 2

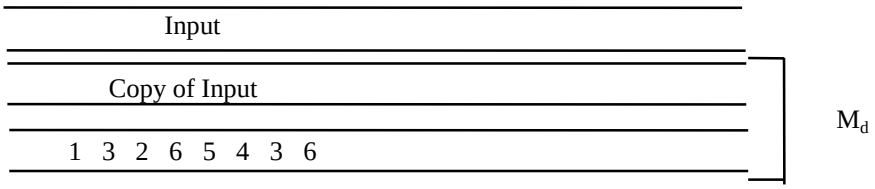
M_d either:

- discovers that M would accept, or
- comes to the end of Tape 2.

In either case, it halts.

The Construction, Continued

M' (the machine that simulates M):



Steps of M':

write ε on Tape 3
until M_d accepts do
(1) copy Input from Tape 1 to Tape 2
(2) run M_d
(3) if M_d accepts, exit
(4) otherwise, generate lexicographically next string on Tape 3.

Pass	1	2	3		7	8	9		
Tape3	ε	1	2	...	6	11	12	...	2635

Nondeterministic Algorithms

Other Turing Machine Extensions

Multiple heads (on one tape)

Emulation strategy: Use tracks to keep track of tape heads. (See book)

Multiple tapes, multiple heads

Emulation strategy: Use tracks to keep track of tapes and tape heads.

Two-dimensional semi-infinite "tape"

Emulation strategy: Use diagonal enumeration of two-dimensional grid. Use second tape to help you keep track of where the tape head is. (See book)

Two-dimensional infinite "tape" (really a sheet)

Emulation strategy: Use modified diagonal enumeration as with the semi-infinite case.

What About Turing Machine Restrictions?

Can we make Turing machines even more limited and still get all the power?

Example:

We allow a tape alphabet of arbitrary size. What happens if we limit it to:

- One character?
- Two characters?
- Three characters?

Problem Encoding, TM Encoding, and the Universal TM

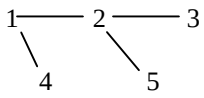
Read K & S 5.1 & 5.2.

Encoding a Problem as a Language

A Turing Machines deciding a language is analogous to the TM solving a **decision problem**.

Problem: Is the number n prime?
Instance of the problem: Is the number 9 prime?
Encoding of the problem, ⟨n⟩: n as a binary number. Example: 1001

Problem: Is an undirected graph G connected?
Instance of the problem: Is the following graph connected?



Encoding of the problem, ⟨G⟩:

1) |V| as a binary number
2) A list of edges represented by pairs of binary numbers being the vertex numbers that the edge connects
3) All such binary numbers are separated by "/".

Example: 101/1/10/10/11/1/100/10/101

Problem View vs. Language View

Problem View: It is *unsolvable* whether a Turing Machine halts on a given input. This is called the **Halting Problem**.

Language View: Let H = {⟨M, w⟩ : TM M halts on input string w}
H is recursively enumerable but not recursive.

The Universal Turing Machine

Problem: All our machines so far are hardwired.

Question: Does it make sense to talk about a programmable Turing machine that accepts as input
program input string
executes the program, and outputs
output string

Yes, it's called the Universal Turing Machine.

Notice that the Universal Turing machine semidecides H = {⟨M, w⟩ : TM M halts on input string w} = L(U).

To define the Universal Turing Machine U we need to do two things:
1. Define an encoding operation for Turing machines.
2. Describe the operation of U given an input tape containing two inputs:
   - encoded Turing machine M,
   - encoded input string to be given to M.

Encoding a Turing Machine M

We need to describe M = (K, Σ, δ, s, H) as a string. To do this we must:
1. Encode δ
2. Specify s.
3. Specify H (and y and n, if applicable)

1. To encode δ, we need to:
    1. Encode the states
    2. Encode the tape alphabet
    3. Specify the transitions

1.1 Encode the states as
qs : $s \in \{0, 1\}^+$ and
|s| = i and
i is the smallest integer such that $2^i \geq |K|$

Example: 9 states i = 4
s = q0000,
remaining states: q0001, q0010, q0011,
q0100, q0101, q0110, q0111, q1000

Encoding a Turing Machine M, Continued

1.2 Encode the tape alphabet as
as : $s \in \{0, 1\}^+$ and
|s| = j and
j is the smallest integer such that $2^j \geq |\Sigma| + 2$ (the + 2 allows for ← and →)
Example: Σ = {◊, ❑, a, b} j = 3
❑ = a000
◊ = a001
← = a010
→ = a011
a = a100
b = a101

Encoding a Turing Machine M, Continued

1.3 Specify transitions as (state, input, state, output)
Example: (q00,a000,q11,a000)
2. Specify s as $q0^i$
3. Specify H:
- States with no transitions out are in H.
- If M decides a language, then H = {y, n}, and we will adopt the convention that y is the lexicographically smaller of the two states.
    y = q010 n = q011

Encoding Input Strings

We encode input strings to a machine M using the same character encoding we use for M.
For example, suppose that we are using the following encoding for symbols in M:

symbol	representation
❑	a000
◊	a001
←	a010
→	a011
a	a100

Then we would represent the string s = ◊aa❑a as "s" = ⟨s⟩ = a001a100a100a000a100

An Encoding Example

Consider M = ({s, q, h}, {❑, ◊,a}, δ, s, {h}), where δ =

state	symbol	δ
s	a	(q, ❑)
s	❑	(h, ❑)
s	◊	(s, →)
q	a	(s, a)
q	❑	(s, →)
q	◊	(q, →)

state/symbol	representation
s	q00
q	q01
h	q11
❑	a000
◊	a001
←	a010
→	a011
a	a100

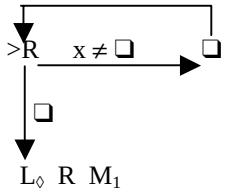
The representation of M, denoted, "M", ⟨M⟩, or sometimes ρ(M) =
(q00,a100,q01,a000), (q00,a000,q11,a000), (q00,a001,q00,a011),
(q01,a100,q00,a100), (q01,a000,q00,a011), (q01,a001,q01,a011)

Another Win of Encoding

One big win of defining a way to encode any Turing machine M:

- It will make sense to talk about operations on programs (Turing machines). In other words, we can talk about some Turing machine T that takes another Turing machine (say M_1) as input and transforms it into a different machine (say M_2) that performs some different, but possibly related task.

Example of a transforming TM T:
Input: a machine M_1 that reads its input tape and performs some operation P on it.
Output: a machine M_2 that performs P on an empty input tape:



The Universal Turing Machine

The specification for U:

U("M" "w") = "M(w)"

	"M ---------- M"				"w---------w"			
	1	0	0	0	0	0	0	
◊	❑	❑	❑	❑	❑	❑	❑	❑	❑
	❑	❑	❑	❑	❑	❑	❑	
	❑	❑	❑	❑	❑	❑	❑□	
	❑	❑	❑	❑	❑	❑	❑	

	"◊	❑"	"w---------w"			❑	❑	
	1	0	0	0	0	0	0	
◊	"M ---------- M"					❑	❑	❑	❑	❑
	1	0	0	0	0	0	0	
	q	0	0	0	❑	❑	❑	
	1	❑	❑	❑	❑	❑	❑	

Initialization of U:

1. Copy "M" onto tape 2
2. Insert "◊❑" at the left edge of tape 1, then shift w over.
3. Look at "M", figure out what i is, and write the encoding of state s on tape 3.

The Operation of U

◊	a	0	0	1	a	0	0	❑	❑
	1	0	0	0	0	0	0		
	"M ----------			M"	❑	❑	❑		
	1	0	0	0	0	0	0		
	q	0	0	0	❑	❑	❑		
	1	❑	❑	❑	❑	❑	❑		

Simulate the steps of M:

1. Start with the heads:
   tape 1: the a of the character being scanned,
   tape 2: far left
   tape 3: far left

2. Simulate one step:
   1. Scan tape 2 for a quadruple that matches current state, input pair.
   2. Perform the associated action, by changing tapes 1 and 3. If necessary, extend the tape.
   3. If no quadruple found, halt. Else go back to 2.

An Example

Tape 1: a001a000a100a100a000a100

◊ ❑ a a ❑ a

Tape 2: (q00,a000,q11,a000), (q00,a001,q00,a011),
(q00,a100,q01,a000), (q01,a000,q00,a011),
(q01,a001,q01,a011), (q01,a100,q00,a100)

Tape 3: q01

Result of simulating the next step:

Tape 1: a001a000a100a100a000a100

◊ ❑ a a ❑ a

Tape 3: q00

If A Universal Machine is Such a Good Idea …

Could we define a Universal Finite State Machine?

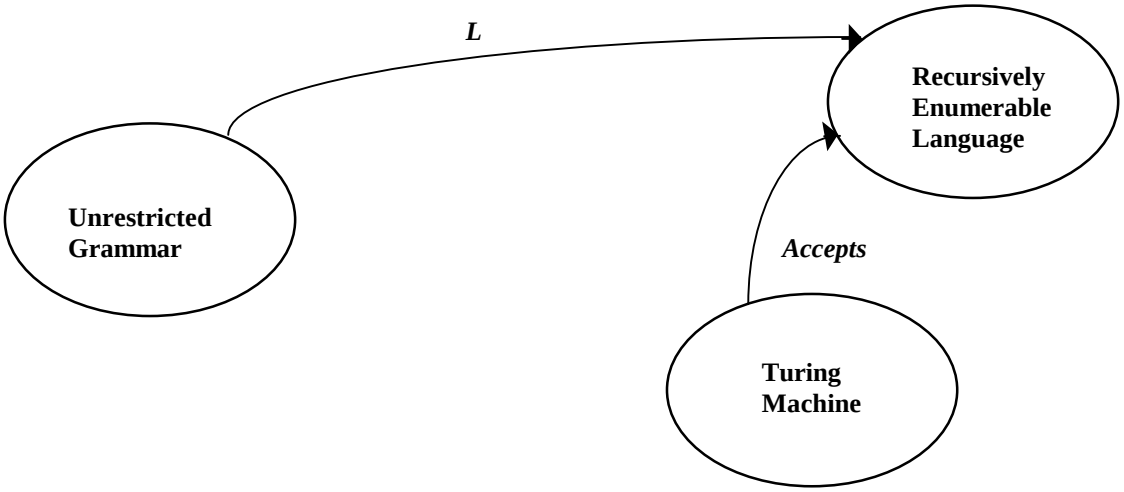
Such a FSM would accept the language

L = {"F" "w" : F is a finite state machine, and w ∈ L(F) }

Grammars and Turing Machines

Do Homework 20.

Grammars, Recursively Enumerable Languages, and Turing Machines



Unrestricted Grammars

An unrestricted, or Type 0, or phrase structure grammar G is a quadruple
(V, Σ, R, S), where

- V is an alphabet,
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite subset of
  - (V* (V-Σ) V*) × V*,
    context N *context* → result
- S (the start symbol) is an element of V - Σ.

We define derivations just as we did for context-free grammars.
The language generated by G is

$\{w \in \Sigma^* : S \Rightarrow_G^* w\}$

There is no notion of a derivation tree or rightmost/leftmost derivation for unrestricted grammars.

Unrestricted Grammars

Example: $L = a^n b^n c^n$, $n > 0$

$S \rightarrow aBSc$
$S \rightarrow aBc$
$Ba \rightarrow aB$
$Bc \rightarrow bc$
$Bb \rightarrow bb$

Another Example

$L = \{w \in \{a, b, c\}^+ :$ number of a's, b's and c's is the same$\}$

$S \rightarrow ABCS$
$S \rightarrow ABC$
$AB \rightarrow BA$
$BC \rightarrow CB$
$AC \rightarrow CA$
$BA \rightarrow AB$
$CA \rightarrow AC$
$CB \rightarrow BC$
$A \rightarrow a$
$B \rightarrow b$
$C \rightarrow c$

A Strong Procedural Feel

Unrestricted grammars have a procedural feel that is absent from restricted grammars.

Derivations often proceed in phases. We make sure that the phases work properly by using nonterminals as flags that we're in a particular phase.

It's very common to have two main phases:
- Generate the right number of the various symbols.
- Move them around to get them in the right order.

No surprise: unrestricted grammars are general computing devices.

Equivalence of Unrestricted Grammars and Turing Machines

Theorem: A language is generated by an unrestricted grammar if and only if it is recursively enumerable (i.e., it is semidecided by some Turing machine M).

Proof:
Only if (grammar → TM): by construction of a nondeterministic Turing machine.

If (TM → grammar): by construction of a grammar that mimics backward computations of M.

Proof that Grammar → Turing Machine

Given a grammar G, produce a Turing machine M that semidecides L(G).

M will be nondeterministic and will use two tapes:

◊	◊	❑	a	b	a	❑	❑	❑	❑	
	0	1	0	0	0	0	0			
	◊	a	S	T	a	b	❑			
	0	1	0	0	0	0	0			

For each nondeterministic "incarnation":
- Tape 1 holds the input.
- Tape 2 holds the current state of a proposed derivation.

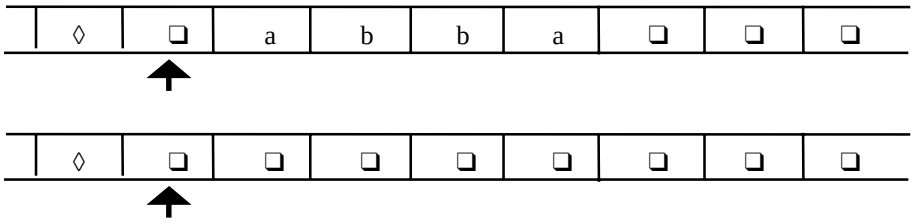
At each step, M nondeterministically chooses a rule to try to apply and a position on tape 2 to start looking for the left hand side of the rule. Or it chooses to check whether tape 2 equals tape 1. If any such machine succeeds, we accept. Otherwise, we keep looking.

Proof that Turing Machine → Grammar

Suppose that M semidecides a language L (it halts when fed strings in L and loops otherwise). Then we can build M' that halts in the configuration (h, ◊❑).

We will define G so that it simulates M' backwards.
We will represent the configuration (q, ◊u❑w) as
>uaqw<

M'
goes from



Then, if w ∈ L, we require that our grammar produce a derivation of the form
$S \Rightarrow_G$ >❑h< (produces final state of M')
$\Rightarrow_G^*$ >❑abq< (some intermediate state of M')
$\Rightarrow_G^*$ >❑sw< (the initial state of M')
$\Rightarrow_G$ w< (via a special rule to clean up >❑s)
$\Rightarrow_G$ w (via a special rule to clean up <)

The Rules of G

$S \rightarrow$ >❑h< (the halting configuration)

>❑s → ε (clean-up rules to be applied at the end)
< → ε

Rules that correspond to δ:

If δ(q, a) = (p, b) : bp → aq

If δ(q, a) = (p, →) : abp → aqb ∀b ∈ Σ
a❑p< → aq<

If δ(q, a) = (p, ←), a ≠ ❑ pa → aq

If δ(q, ❑) = (p, ←) p❑b → ❑qb ∀b ∈ Σ
p< → ❑q<

A *REALLY* Simple Example

M' = (K, {a}, δ, s, {h}), where

δ = {	((s, ❑), (q, →)),	1
	((q, a), (q, →)),	2
	((q, ❑), (t, ←)),	3
	((t, a), (p, ❑)),	4
	((t, ❑), (h, ❑)),	5
	((p, ❑), (t, ←))	6

L = a*

	S →>❑h<	(3)	t❑❑ → ❑q❑	
	>❑s → ε		t❑a → ❑qa	
	< → ε		t< → ❑q<	
		(4)	❑p → at	
(1)	❑❑q→ ❑s❑	(5)	❑h → ❑t	
	❑aq → ❑sa	(6)	t❑❑ → ❑p❑	
	❑❑q< → ❑s<		t❑a → ❑pa	
(2)	a❑q → aq❑		t< → ❑p<	
	aaq → aqa			
	a❑q< → aq<			

Working It Out

	S →>❑h<	1	(3)	t❑❑ → ❑q❑	10
	>❑s → ε	2		t❑a → ❑qa	11
	< → ε	3		t< → ❑q<	12
			(4)	❑p → at	13
(1)	❑❑q→ ❑s❑	4	(5)	❑h → ❑t	14
	❑aq → ❑sa	5	(6)	t❑❑ → ❑p❑	15
	❑❑q< → ❑s<	6		t❑a → ❑pa	16
(2)	a❑q → aq❑	7		t< → ❑p<	17
	aaq → aqa	8			
	a❑q< → aq<	9			

>❑saa<	1	S	⇒ >❑h<	1
>❑aqa<	2		⇒ >❑t<	14
>❑aaq<	2		⇒ >❑❑p<	17
>❑aa❑q<	3		⇒ >❑at<	13
>❑aat<	4		⇒ >❑a❑p<	17
>❑a❑p<	6		⇒ >❑aat<	13
>❑at<	4		⇒ >❑aa❑q<	12
>❑❑p<	6		⇒ >❑aaq<	9
>❑t<	5		⇒ >❑aqa<	8
>❑h<			⇒ >❑saa<	5
			⇒ aa<	2
			⇒ aa	3

An Alternative Proof

An alternative is to build a grammar G that simulates the forward operation of a Turing machine M. It uses alternating symbols to represent two interleaved tapes. One tape remembers the starting string, the other "working" tape simulates the run of the machine.

The first (generate) part of G:
Creates all strings over Σ^* of the form

$w = \Diamond \Diamond \square \square Q s\ a_1\ a_1\ a_2\ a_2\ a_3\ a_3 \square \square \ldots$

The second (test) part of G simulates the execution of M on a particular string w. An example of a partially derived string:

$\Diamond \Diamond \square \square$ a 1 b 2 c c b 4 Q3 a 3

Examples of rules:

b b Q 4 $\rightarrow$ b 4 Q 4 (rewrite b as 4)
b 4 Q 3 $\rightarrow$ Q 3 b 4 (move left)

The third (cleanup) part of G erases the junk if M ever reaches h.

Example rule:

h a 1 $\rightarrow$ a # h (sweep # h to the right erasing the working "tape")

Computing with Grammars

We say that **G computes f** if, for all w, $v \in \Sigma^*$,

$SwS \Rightarrow_G^* v$ iff $v = f(w)$

Example:

$S1S \Rightarrow_G^* 11$
$S11S \Rightarrow_G^* 111$ $f(x) = succ(x)$

A function f is called **grammatically computable** iff there is a grammar G that computes it.

Theorem: A function f is recursive iff it is grammatically computable.
In other words, if a Turing machine can do it, so can a grammar.

Example of Computing with a Grammar

f(x) = 2x, where x is an integer represented in unary

G = ({S, 1}, {1}, R, S), where R =

S1 $\rightarrow$ 11S
SS $\rightarrow \varepsilon$

Example:

Input: S111S

Output:

More on Functions: Why Have We Been Using Recursive as a Synonym for Computable?
Primitive Recursive Functions

Define a set of basic functions:

- $zero_k\ (n_1, n_2, \ldots n_k) = 0$
- $identity_{k,j}\ (n_1, n_2, \ldots n_k) = n_j$
- successor(n) = n + 1

Combining functions:

- Composition of g with $h_1, h_2, \ldots h_k$ is
  $g(h_1(\ ), h_2(\ ), \ldots h_k(\ ))$
- Primitive recursion of f in terms of g and h:
  $f(n_1,n_2,\ldots n_k,\ \ 0) = g(n_1,n_2,\ldots n_k)$
  $f(n_1,n_2,\ldots n_k,m+1) = h(n_1,n_2,\ldots n_k, m, f(n_1, n_2,\ldots n_k,m))$

Example: plus(n, 0) = n
plus(n, m+1) = succ(plus(n, m))

Primitive Recursive Functions and Computability

Trivially true: all primitive recursive functions are Turing computable.
What about the other way: Not all Turing computable functions are primitive recursive.

Proof:
Lexicographically enumerate the unary primitive recursive functions, $f_0, f_1, f_2, f_3, \ldots$.
Define $g(n) = f_n(n) + 1$.
G is clearly computable, but it is not on the list. Suppose it were f_m for some m. Then
$f_m(m) = f_m(m) + 1$, which is absurd.

	0	1	2	3	4
f_0					
f_1					
f_2					
f_3				27	
f_4					

Suppose g is f_3. Then g(3) = 27 + 1 = 28. Contradiction.

Functions that Aren't Primitive Recursive

Example: Ackermann's function: A(0, y) = y + 1
A(x + 1, 0) = A(x, 1)
A(x + 1, y + 1) = A(x, A(x + 1, y))

	0	**1**	**2**	**3**	**4**
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	$2^{65536}-3$ *	$2^{2^{65536}}-3$ #	$2^{2^{2^{65536}}}-3$ %

* 19,729 digits
\# 10^{5940} digits
% $10^{10^{5939}}$ digits

10^{17} seconds since big bang
10^{87} protons and neutrons
10^{-23} light seconds = width of proton or neutron

Thus writing digits at the speed of light on all protons and neutrons in the universe (all lined up) starting at the big bang would have produced 10^{127} digits.

Recursive Functions

A function is **μ-recursive** if it can be obtained from the basic functions using the operations of:

- Composition,
- Recursive definition, and
- Minimalization of minimalizable functions:

The **minimalization** of g (of k + 1 arguments) is a function f of k arguments defined as:

$f(n_1,n_2,\ldots n_k)$ = the least m such at $g(n_1,n_2,\ldots n_k,m)=1$, if such an m exists,
0 otherwise

A function g is **minimalizable** iff for every $n_1,n_2,\ldots n_k$, there is an m such that $g(n_1,n_2,\ldots n_k,m)=1$.

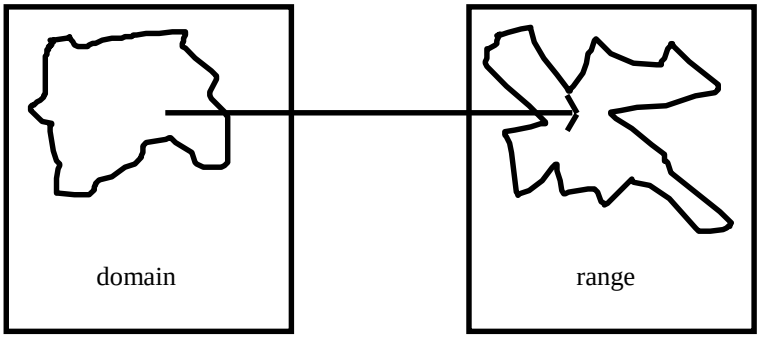
Theorem: A function is μ-recursive iff it is recursive (i.e., computable by a Turing machine).

Partial Recursive Functions

Consider the following function f:

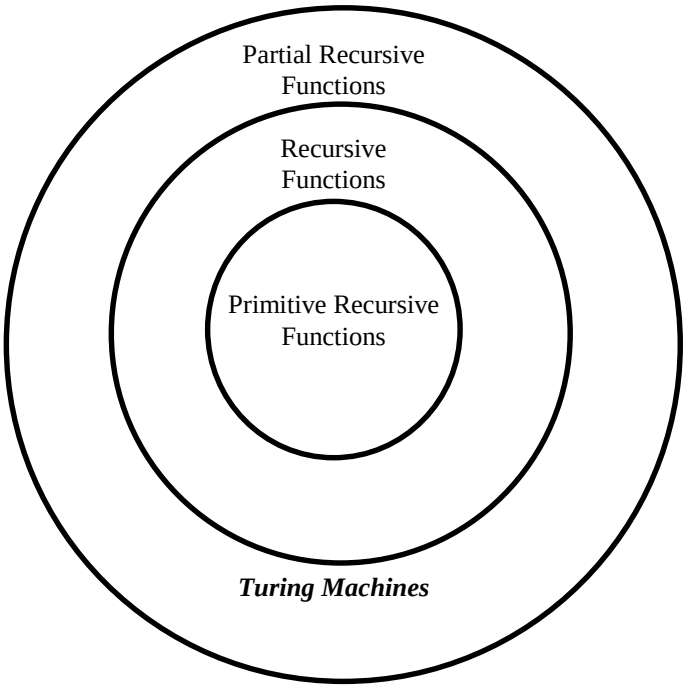
f(n) = 1 if TM(n) halts on a blank tape
0 otherwise

The domain of f is the natural numbers. Is f recursive?

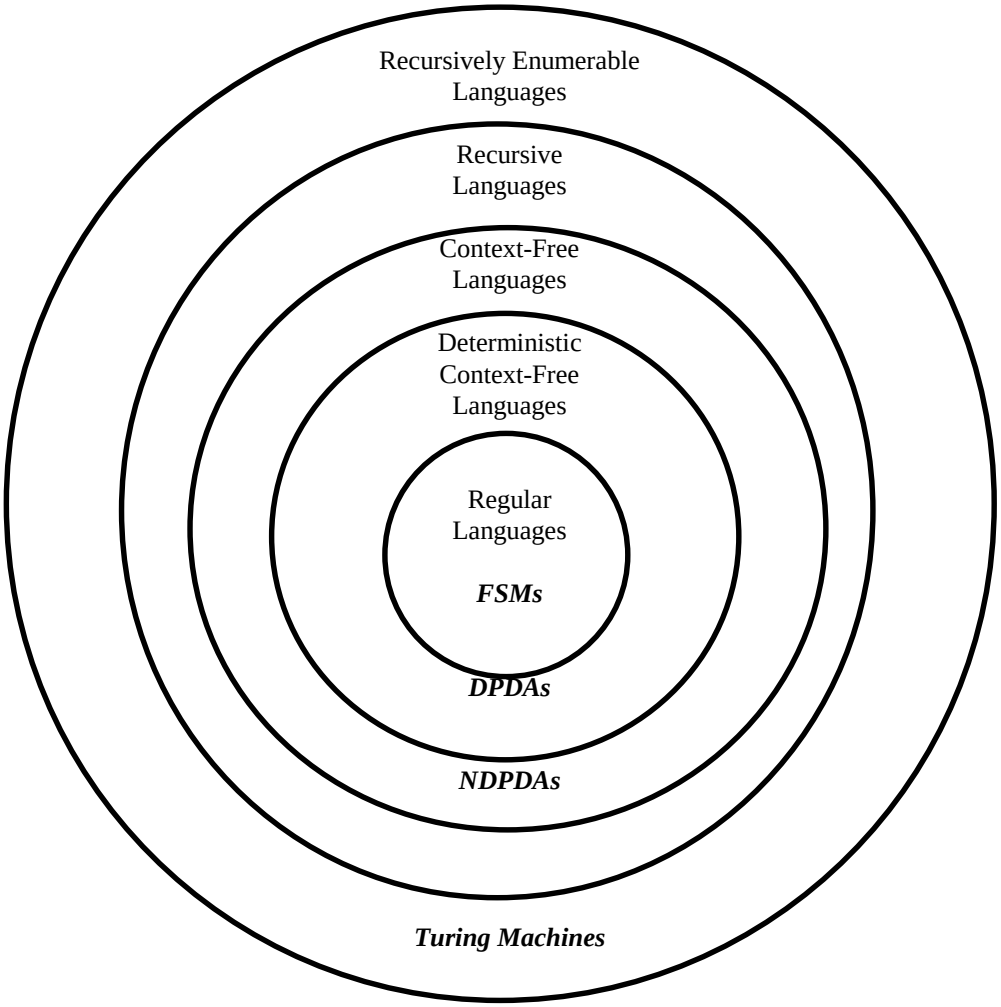


Theorem: There are uncountably many partially recursive functions (but only countably many Turing machines).

Functions and Machines



Languages and Machines



Is There Anything In Between CFGs and Unrestricted Grammars?

Answer: yes, various things have been proposed.

Context-Sensitive Grammars and Languages:

A grammar G is context sensitive if all productions are of the form

$x \rightarrow y$

and $|x| \leq |y|$

In other words, there are no length-reducing rules.

A language is context sensitive if there exists a context-sensitive grammar for it.

Examples:

$L = \{a^n b^n c^n, n > 0\}$

$L = \{w \in \{a, b, c\}^+ : \text{number of a's, b's and c's is the same}\}$

Context-Sensitive Languages are Recursive

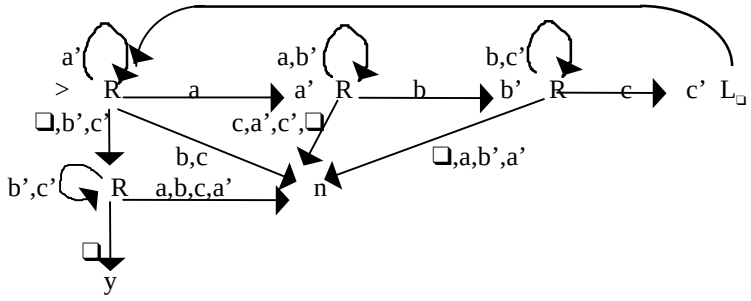
The basic idea: To decide if a string w is in L, start generating strings systematically, shortest first. If you generate w, accept. If you get to strings that are longer than w, reject.

Linear Bounded Automata

A linear bounded automaton is a nondeterministic Turing machine the length of whose tape is bounded by some fixed constant k times the length of the input.

Example: $L = \{a^n b^n c^n : n \geq 0\}$

◊❑aabbcc❑❑❑❑❑❑❑❑❑



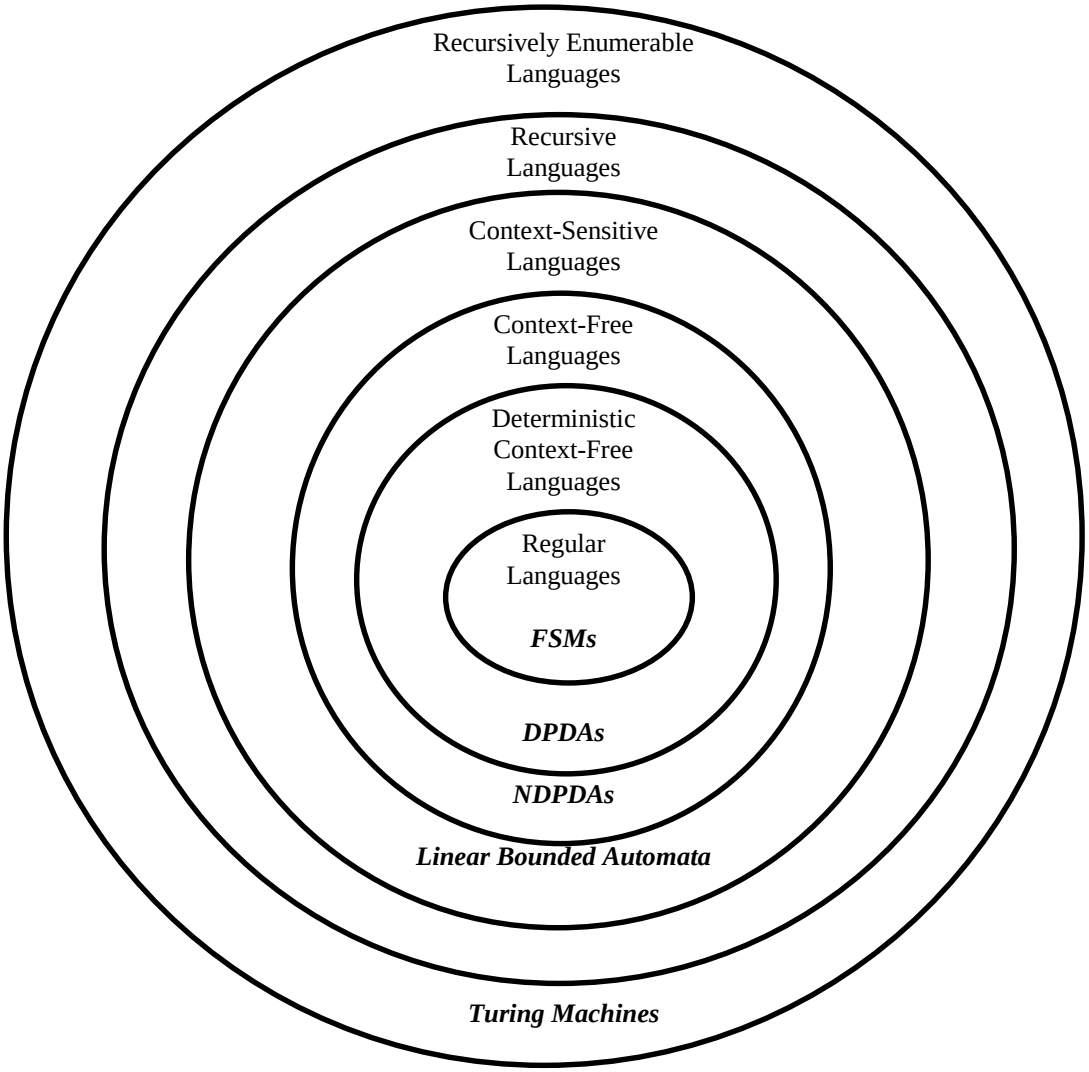
Context-Sensitive Languages and Linear Bounded Automata

Theorem: The set of context-sensitive languages is exactly the set of languages that can be accepted by linear bounded automata.

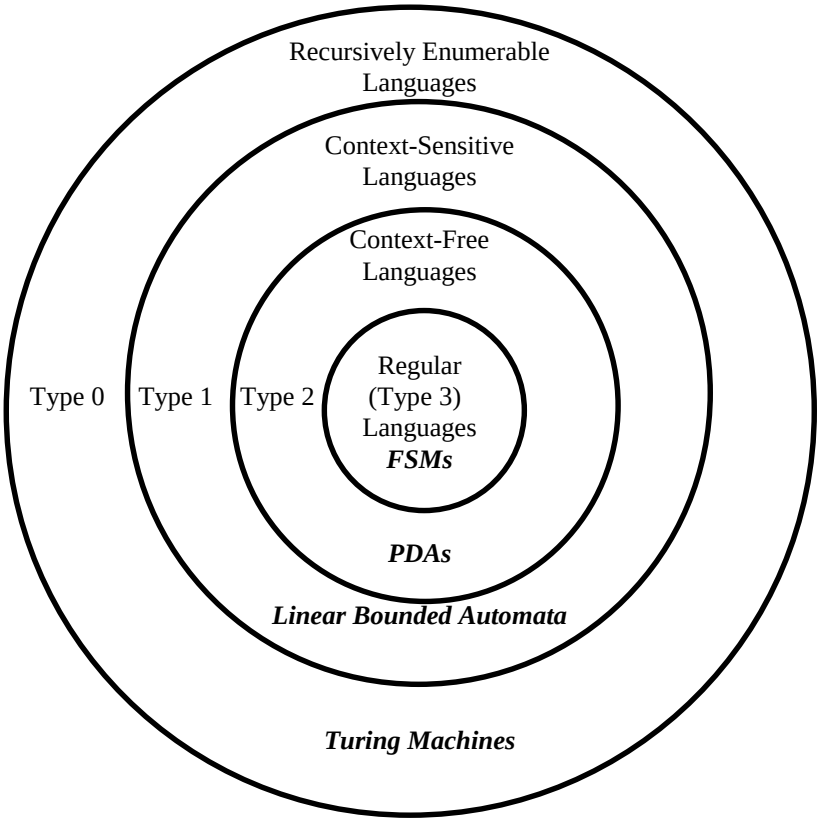
Proof: (sketch) We can construct a linear-bounded automaton B for any context-sensitive language L defined by some grammar G. We build a machine B with a two track tape. On input w, B keeps w on the first tape. On the second tape, it nondeterministically constructs all derivations of G. The key is that as soon as any derivation becomes longer than |w| we stop, since we know it can never get any shorter and thus match w. There is also a proof that from any lba we can construct a context-sensitive grammar, analogous to the one we used for Turing machines and unrestricted grammars.

Theorem: There exist recursive languages that are not context sensitive.

Languages and Machines



The Chomsky Hierarchy



Undecidabilty

Read K & S 5.1, 5.3, & 5.4.
Read Supplementary Materials: Recursively Enumerable Languages, Turing Machines, and Decidability.
Do Homeworks 21 & 22.

Church's Thesis
(Church-Turing Thesis)

An **algorithm** is a formal procedure that halts.

The Thesis: Anything that can be computed by any algorithm can be computed by a Turing machine.

Another way to state it: All "reasonable" formal models of computation are equivalent to the Turing machine.

This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

Examples:
- unrestricted grammars
- lambda calculus
- cellular automata
- DNA computing
- quantum computing (?)

The Unsolvability of the Halting Problem

Suppose we could implement the decision procedure

```
HALTS(M, x)
        M: string representing a Turing Machine
        x: string representing the input for M
        If M(x) halts then True
                        else False
```

Then we could define

```
TROUBLE(x)
        x: string
        If HALTS(x, x) then loop forever
                        else halt
```

So now what happens if we invoke TROUBLE("TROUBLE"), which invokes HALTS("TROUBLE", "TROUBLE")

If HALTS says that TROUBLE halts on itself then TROUBLE loops. IF HALTS says that TROUBLE loops, then TROUBLE halts. Either way, we reach a contradiction, so HALTS(M, x) cannot be made into a decision procedure.

Another View

The Problem View: The halting problem is undecidable.

The Language View: Let H =
{"M" "w" : TM M halts on input string w}
H is recursively enumerable but not recursive.

Why?

H is recursively enumerable because it can be semidecided by U, the Universal Turing Machine.

But *H cannot be recursive*. If it were, then it would be decided by some TM MH. But MH("M" "w") would have to be:
If M is not a syntactically valid TM, then False.
else HALTS("M" "w")

But we know cannot that HALTS cannot exist.

If H were Recursive

H = {"M" "w" : TM M halts on input string w}

Theorem: If H were also recursive, then every recursively enumerable language would be recursive.

Proof: Let L be any RE language. Since L is RE, there exists a TM M that semidecides it.

Suppose H is recursive and thus is decided by some TM O (oracle).

We can build a TM M' from M that decides L:
1. M' transforms its input tape from ◊❑w❑ to ◊❑"M""w"❑.
2. M' invokes O on its tape and returns whatever answer O returns.

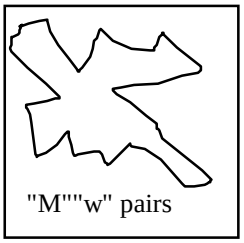
So, if H were recursive, all RE languages would be. **But it isn't.**

Undecidable Problems, Languages that Are Not Recursive, and Partial Functions

The Problem View: The halting problem is undecidable.

The Language View: Let H =
{"M" "w" : TM M halts on input string w}
H is recursively enumerable but not recursive.

The Functional View: Let f (w) = M(w)
f is a partial function on Σ*



Other Undecidable Problems About Turing Machines

- Given a Turing machine M, does M halt on the empty tape?
- Given a Turing machine M, is there any string on which M halts?
- Given a Turing machine M, does M halt on every input string?
- Given two Turing machines M_1 and M_2, do they halt on the same input strings?
- Given a Turing machine M, is the language that M semidecides regular? Is it context-free? Is it recursive?

Post Correspondence Problem

Consider two lists of strings over some alphabet Σ. The lists must be finite and of equal length.

$A = x_1, x_2, x_3, \ldots, x_n$
$B = y_1, y_2, y_3, \ldots, y_n$

Question: Does there exist some finite sequence of integers that can be viewed as indexes of A and B such that, when elements of A are selected as specified and concatenated together, we get the same string we get when elements of B are selected also as specified?

For example, if we assert that 1, 3, 4 is such a sequence, we're asserting that $x_1x_3x_4 = y_1y_3y_4$

Any problem of this form is an instance of the Post Correspondence Problem.

Is the Post Correspondence Problem decidable?

Post Correspondence Problem Examples

i	A	B
1	1	111
2	10111	10
3	10	0

i	A	B
1	10	101
2	011	11
3	101	011

Some Languages Aren't Even Recursively Enumerable

A pragmatically non RE language: L_1={ (i, j) : i, j are integers where the low order five digits of i are a street address number and j is the number of houses with that number on which it rained on November 13, 1946 }

An analytically non RE language: L_2={x : x = "M" of a Turing machine M and M("M") does *not* halt}

Why isn't L_2 RE? Suppose it were. Then there would be a TM M* that semidecides L_2. Is "M*" in L_2?

- If it is, then M*("M*") halts (by the definition of M* as a semideciding machine for L_2)
- But, by the definition of L_2, if "M*" ∈ L_2, then M*("M*") does not halt.

Contradiction. So L_2 is not RE.

Another Non RE Language

$\overline{H}$

Why not?

Reduction

Let $L_1, L_2 \subseteq \Sigma^*$ be languages. A **reduction** from L_1 to L_2 is a recursive function $\tau: \Sigma^* \rightarrow \Sigma^*$ such that

$x \in L_1$ iff $\tau(x) \in L_2$.

Example:

$L_1 = \{a, b : a,b \in N : b = a + 1\}$

$\Downarrow$ τ = Succ

$\Downarrow$ a, b becomes Succ(a), b

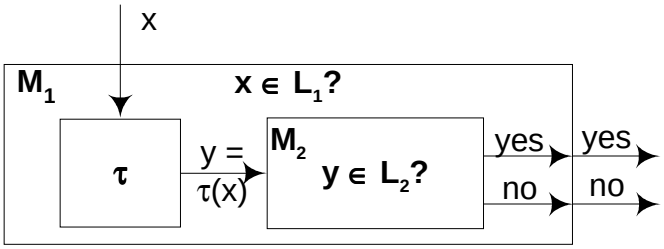
$L_2 = \{a, b : a,b \in N : a = b\}$

If there is a Turing machine M_2 to decide L_2, then I can build a Turing machine M_1 to decide L_1:

1. Take the input and apply Succ to the first number.
2. Invoke M_2 on the result.
3. Return whatever answer M_2 returns.

Reductions and Recursive Languages

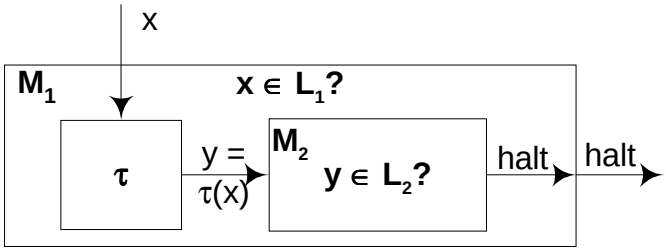
Theorem: If there is a reduction from L_1 to L_2 and L_2 is recursive, then L_1 is recursive.



Theorem: If there is a reduction from L_1 to L_2 and L_1 is not recursive, then L_2 is not recursive.

Reductions and RE Languages

Theorem: If there is a reduction from L_1 to L_2 and L_2 is RE, then L_1 is RE.



Theorem: If there is a reduction from L_1 to L_2 and L_1 is not RE, then L_2 is not RE.

Can it be Decided if M Halts on the Empty Tape?

This is equivalent to, "Is the language L_2 = {"M" : Turing machine M halts on the empty tape} recursive?"

L_1 = H = {s = "M" "w" : Turing machine M halts on input string w}

$\Downarrow$ τ

(?M_2) L_2 = {s = "M" : Turing machine M halts on the empty tape}

Let τ be the function that, from "M" and "w", constructs "M*", which operates as follows on an empty input tape:

1. Write w on the tape.
2. Operate as M would have.

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

A Formal Reduction Proof

Prove that L_2 = {⟨M⟩: Turing machine M halts on the empty tape} is not recursive.

Proof that L_2 is not recursive via a reduction from H = {⟨M, w⟩: Turing machine M halts on input string w}, a non-recursive language. Suppose that there exists a TM, M_2 that decides L_2. Construct a machine to decide H as $M_1(\langle M, w\rangle) = M_2(\tau(\langle M, w\rangle))$. The τ function creates from ⟨M⟩ and ⟨w⟩ a new machine M*. M* ignores its input and runs M on w, halting exactly when M halts on w.

- $\langle M, w\rangle \in H \Rightarrow$ M halts on w $\Rightarrow$ M* always halts $\Rightarrow \varepsilon \in L(M^*) \Rightarrow \langle M^*\rangle \in L_2 \Rightarrow M_2$ accepts $\Rightarrow M_1$ accepts.
- $\langle M, w\rangle \notin H \Rightarrow$ M does not halt on w $\Rightarrow \varepsilon \notin L(M^*) \Rightarrow \langle M^*\rangle \notin L_2 \Rightarrow M_2$ rejects $\Rightarrow M_1$ rejects.

Thus, if there is a machine M_2 that decides L_2, we could use it to build a machine that decides H. Contradiction. $\therefore L_2$ is not recursive.

Important Elements in a Reduction Proof

- A clear declaration of the reduction "from" and "to" languages and what you're trying to prove with the reduction.
- A description of how a machine is being constructed for the "from" language based on an assumed machine for the "to" language and a recursive τ function.
- A description of the τ function's inputs and outputs. If τ is doing anything nontrivial, it is a good idea to argue that it is recursive.
- Note that machine diagrams are not necessary or even sufficient in these proofs. Use them as thought devices, where needed.
- Run through the logic that demonstrates how the "from" language is being decided by your reduction. You must do both accepting and rejecting cases.
- Declare that the reduction proves that your "to" language is not recursive.

The Most Common Mistake: Doing the Reduction Backwards

The right way to use reduction to show that L_2 is not recursive:

1. Given that L_1 is not recursive,
2. Reduce L_1 to L_2, i.e. show how to solve L_1 (the known one) in terms of L_2 (the unknown one)

L_1 $\Downarrow$ L_2

Example: If there exists a machine M_2 that solves L_2, the problem of deciding whether a Turing machine halts on a blank tape, then we could do H (deciding whether M halts on w) as follows:

1. Create M* from M such that M*, given a blank tape, first writes w on its tape, then simulates the behavior of M.
2. Return M_2("M*").

Doing it wrong by reducing L_2 (the unknown one to L_1): If there exists a machine M_1 that solves H, then we could build a machine that solves L_2 as follows:

1. Return (M_1("M", "")).

Why Backwards Doesn't Work

Suppose that we have proved that the following problem L_1 is unsolvable: Determine the number of days that have elapsed since the beginning of the universe.

Now consider the following problem L_2: Determine the number of days that had elapsed between the beginning of the universe and the assassination of Abraham Lincoln.

Reduce L_1 to L_2:
$L_1 = L_2 + (\text{now} - 4/9/1865)$

L_1 ⇓ L_2

Reduce L_2 to L_1:
$L_2 = L_1 - (\text{now} - 4/9/1865)$

L_2 ⇓ L_1

Why Backwards Doesn't Work, Continued

L_1 = days since beginning of universe
L_2 = elapsed days between the beginning of the universe and the assassination of Abraham Lincoln.
L_3 = days between the assassination of Abraham Lincoln and now.

Considering L_2:
Reduce L_1 to L_2:
$L_1 = L_2 + (\text{now} - 4/9/1865)$

L_1 ⇓ L_2

Reduce L_2 to L_1:
$L_2 = L_1 - (\text{now} - 4/9/1865)$

L_2 ⇓ L_1

Considering L_3:
Reduce L_1 to L_3:
$L_1 = \text{oops}$

L_1 ⇓ L_3

Reduce L_3 to L_1:
$L_3 = L_1 - 365 - (\text{now} - 4/9/1866)$

L_3 ⇓ L_1

Is There Any String on Which M Halts?

$L_1 = H =$ {s = "M" "w" : Turing machine M halts on input string w}

⇓ τ

(?M_2) $L_2 =$ {s = "M" : there exists a string on which Turing machine M halts}

Let τ be the function that, from "M" and "w", constructs "M*", which operates as follows:

1. M* examines its input tape.
2. If it is equal to w, then it simulates M.
3. If not, it loops.

Clearly the only input on which M* has a chance of halting is w, which it does iff M would halt on w.

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Does M Halt on All Inputs?

L_1 = {s = "M" : Turing machine M halts on the empty tape}

$\Downarrow \tau$

(?M_2) L_2 = {s = "M" : Turing machine M halts on all inputs}

Let τ be the function that, from "M", constructs "M*", which operates as follows:

1. Erase the input tape.
2. Simulate M.

Clearly M* either halts on all inputs or on none, since it ignores its input.

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Rice's Theorem

Theorem: No nontrivial property of the recursively enumerable languages is decidable.

Alternate statement: Let P: $2^{\Sigma^*} \rightarrow$ {true, false} be a nontrivial property of the recursively enumerable languages. The language {"M": P(L(M)) = True} is not recursive.

By "nontrivial" we mean a property that is not simply true for all languages or false for all languages.

Examples:

- L contains only even length strings.
- L contains an odd number of strings.
- L contains all strings that start with "a".
- L is infinite.
- L is regular.

Note:
Rice's theorem applies to languages, not machines. So, for example, the following properties of machines are decidable:

- M contains an even number of states
- M has an odd number of symbols in its tape alphabet

Of course, we need a way to define a language. We'll use machines to do that, but the properties we'll deal with are properties of L(M), not of M itself.

Proof of Rice's Theorem

Proof: Let P be any nontrivial property of the RE languages.

L_1 = H = {s = "M" "w" : Turing machine M halts on input string w}

$\Downarrow \tau$

(?M_2) L_2 = {s = "M" : P(L(M)) = true}

Either P(∅) = true or P(∅) = false. Assume it is false (a matching proof exists if it is true). Since P is nontrivial, there is some language L_P such that P(L_P) is true. Let M_P be some Turing machine that semidecides L_P.

Let τ construct "M*", which operates as follows:

1. Copy its input y to another track for later.
2. Write w on its input tape and execute M on w.
3. If M halts, put y back on the tape and execute M_P.
4. If M_P halts on y, accept.

Claim: If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Why?

Two cases to consider:

- "M" "w" $\in$ H $\Rightarrow$ M halts on w $\Rightarrow$ M* will halt on all strings that are accepted by M_P $\Rightarrow$ L(M*) = L(M_P) = L_P $\Rightarrow$ P(L(M*)) = P(L_P) = true $\Rightarrow$ M_2 decides P, so M_2 accepts "M*" $\Rightarrow$ M_1 accepts.

- "M" "w" $\notin$ H $\Rightarrow$ M doesn't halt on w $\Rightarrow$ M* will halt on nothing $\Rightarrow$ L(M*) = $\varnothing$ $\Rightarrow$ P(L(M*)) = P($\varnothing$) = false $\Rightarrow$ M_2 decides P, so M_2 rejects "M*" $\Rightarrow$ M_1 rejects.

Using Rice's Theorem

Theorem: No nontrivial property of the recursively enumerable languages is decidable.

To use Rice's Theorem to show that a language L is not recursive we must:

- Specify a language property, P(L)

- Show that the domain of P is the set of recursively enumerable languages.

- Show that P is nontrivial:
  - P is true of at least one language
  - P is false of at least one language

Using Rice's Theorem: An Example

L = {s = "M" : there exists a string on which Turing machine M halts}.
= {s = "M" : L(M) $\neq \varnothing$ }

- Specify a language property, P(L):
  P(L) = True iff L $\neq \varnothing$

- Show that the domain of P is the set of recursively enumerable languages.
  The domain of P is the set of languages semidecided by some TM. This is exactly the set of RE languages.

- Show that P is nontrivial:
  P is true of at least one language: P({ε}) = True
  P is false of at least one language: P($\varnothing$) = False

Inappropriate Uses of Rice's Theorem

Example 1:
L = {s = "M" : M writes a 1 within three moves}.

- Specify a language property, P(L)
  P(M?) = True if M writes a 1 within three moves,
  False otherwise
- Show that the domain of P is the set of recursively enumerable languages.
  ??? The domain of P is the set of all TMs, not their languages

Example 2:
L = {s = "M1" "M2": L(M1) = L(M2)}.

- Specify a language property. P(L)
  P(M1?, M2?) = True if L(M1) = L(M2)
  False otherwise
- Show that the domain of P is the set of recursively enumerable languages.
  ??? The domain of P is RE $\times$ RE

Given a Turing Machine M, is L(M) Regular (or Context Free or Recursive)?

Is this problem decidable?

No, by Rice's Theorem, since being regular (or context free or recursive) is a nontrivial property of the recursively enumerable languages.

We can also show this directly (via the same technique we used to prove the more general claim contained in Rice's Theorem):

Given a Turing Machine M, is L(M) Regular (or Context Free or Recursive)?

$L_1 = H = \{s = \text{"M" "w"} : \text{Turing machine M halts on input string w}\}$

$\Downarrow \tau$

$(?M_2) \quad L_2 = \{s = \text{"M"} : L(M) \text{ is regular}\}$

Let τ be the function that, from "M" and "w", constructs "M*", whose own input is a string
t = "M_*" "w_*"
M*("M_*" "w_*") operates as follows:

1. Copy its input to another track for later.
2. Write w on its input tape and execute M on w.
3. If M halts, invoke U on "M_*" "w_*".
4. If U halts, halt and accept.

If M_2 exists, then $\neg M_2(M^*(s))$ decides L_1 (H).

Why?
If M does not halt on w, then M* accepts $\varnothing$ (which is regular).
If M does halt on w, then M* accepts H (which is not regular).

Undecidable Problems About Unrestricted Grammars

- Given a grammar G and a string w, is $w \in L(G)$?
- Given a grammar G, is $\varepsilon \in L(G)$?
- Given two grammars G_1 and G_2, is $L(G_1) = L(G_2)$?
- Given a grammar G, is $L(G) = \varnothing$?

Given a Grammar G and a String w, Is $w \in L(G)$?

$L_1 = H = \{s = \text{"M" "w"} : \text{Turing machine M halts on input string w}\}$

$\Downarrow \tau$

$(?M_2) \quad L_2 = \{s = \text{"G" "w"} : w \in L(G)\}$

Let τ be the construction that builds a grammar G for the language L that is semidecided by M. Thus
$w \in L(G)$ iff M(w) halts.

Then $\tau(\text{"M" "w"}) = \text{"G" "w"}$

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Undecidable Problems About Context-Free Grammars

- Given a context-free grammar G, is $L(G) = \Sigma^*$?
- Given two context-free grammars G_1 and G_2, is $L(G_1) = L(G_2)$?
- Given two context-free grammars G_1 and G_2, is $L(G_1) \cap L(G_2) = \varnothing$?
- Is context-free grammar, G ambiguous?
- Given two pushdown automata M_1 and M_2, do they accept precisely the same language?
- Given a pushdown automaton M, find an equivalent pushdown automaton with as few states as possible.

Given Two Context-Free Grammars G_1 and G_2, Is $L(G_1) = L(G_2)$?

$L_1 =$ {s = "G" a CFG G and $L(G) = \Sigma^*$}

$\Downarrow \quad \tau$

($?M_2$) $L_2 =$ {s = "G_1" "G_2" : G_1 and G_2 are CFGs and $L(G_1) = L(G_2)$}

Let τ append the description of a context free grammar G_{Σ^*} that generates Σ^*.

Then, τ("G") = "G" "G_{Σ^*}"

If M_2 exists, then $M_1 = M_2(M_\tau(s))$ decides L_1.

Non-RE Languages

There are an uncountable number of non-RE languages, but only a countably infinite number of TM's (hence RE languages).
$\therefore$The class of non-RE languages is <u>much</u> bigger than that of RE languages!

Intuition: Non-RE languages usually involve either infinite search or knowing a TM will infinite loop to accept a string.

{⟨M⟩: M is a TM that does not halt on the empty tape}
{⟨M⟩: M is a TM and $L(M) = \Sigma^*$}
{⟨M⟩: M is a TM and there does not exist a string on which M halts}

Proving Languages are not RE

- Diagonalization
- Complement RE, not recursive
- Reduction from a non-RE language
- Rice's theorem for non-RE languages (not covered)

Diagonalization

L={⟨M⟩: M is a TM and M(⟨M⟩) does not halt} is not RE

Suppose L is RE. There is a TM M* that semidecides L. Is ⟨M*⟩ in L?
- If it is, then M*(⟨M*⟩) halts (by the definition of M* as a semideciding machine for L)
- But, by the definition of L, if ⟨M*⟩ $\in$ L, then M*(⟨M*⟩) does not halt.

Contradiction. So L is not RE.

(This is a very "bare-bones" diagonalization proof.)

Diagonalization can only be easily applied to a few non-RE languages.

Complement of an RE, but not Recursive Language

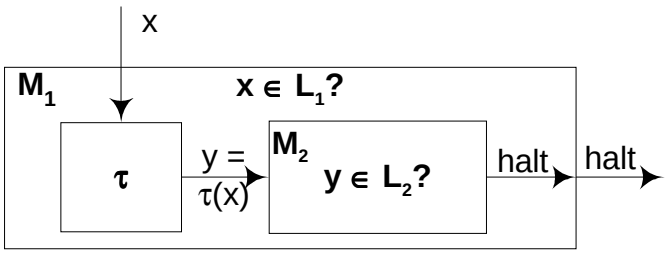
Example**:** $\overline{H} = \{\langle M, w\rangle: M \text{ does not accept } w\}$
Consider $H = \{\langle M, w\rangle: M \text{ is a TM that accepts } w\}$:

- H is RE—it is semidecided by U, the Universal Turing Machine.
- H is not recursive—it is equivalent to the halting problem, which is undecidable.

From the theorem, $\overline{H}$ is not RE.

Reductions and RE Languages

Theorem: If there is a reduction from L_1 to L_2 and L_2 is RE, then L_1 is RE.



Theorem: If there is a reduction from L_1 to L_2 and L_1 is not RE, then L_2 is not RE.

Reduction from a known non-RE Language

Using a reduction from a non-RE language:

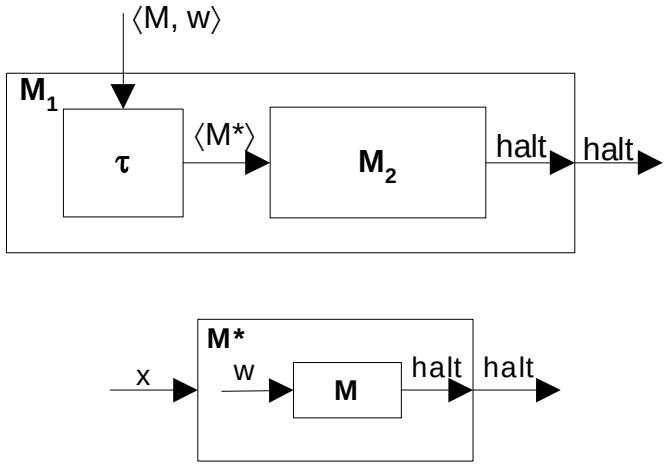
$L_1 = \overline{H} = \{\langle M, w\rangle: \text{Turing machine M does not halt on input string w}\}$

$\Downarrow \tau$

(?M_2) $L_2 = \{\langle M\rangle: \text{there does not exist a string on which Turing machine M halts}\}$

Let τ be the function that, from $\langle M\rangle$ and $\langle w\rangle$, constructs $\langle M^*\rangle$, which operates as follows:

1. Erase the input tape (M* ignores its input).
2. Write w on the tape
3. Run M on w.

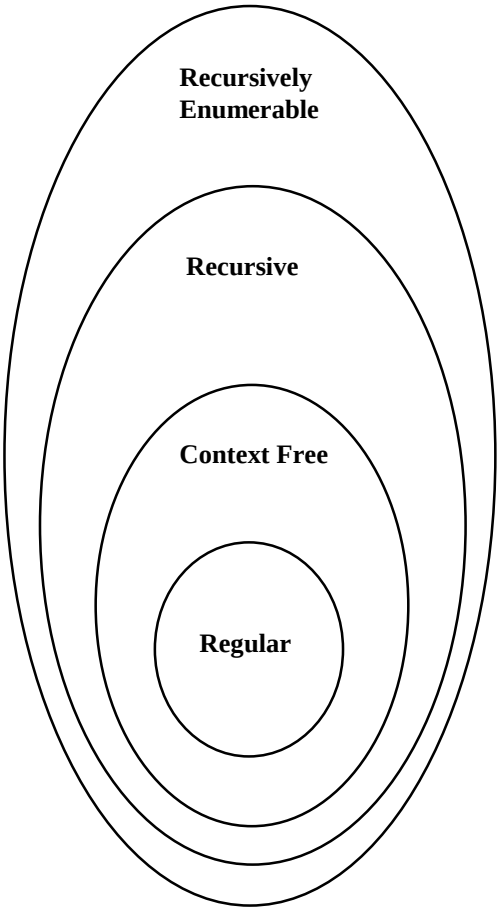


$\langle M, w\rangle \in \overline{H} \Rightarrow$ M does not halt on w $\Rightarrow$ M* does not halt on any input $\Rightarrow$ M* halts on nothing $\Rightarrow M_2$ accepts (halts).
$\langle M, w\rangle \notin \overline{H} \Rightarrow$ M halts on w $\Rightarrow$ M* halts on everything $\Rightarrow M_2$ loops.

If M_2 exists, then $M_1(\langle M, w\rangle) = M_2(M_\tau(\langle M, w\rangle))$ and M_1 semidecides L_1. Contradiction. L_1 is not RE. $\therefore L_2$ is not RE.

Language
Summary

IN		OUT
Semidecidable Enumerable Unrestricted grammar	**Recursively Enumerable**	
Decision procedure Lexicicographically enumerable Complement is recursively enumer.	**Recursive**	Diagonalization Reduction
CF grammar PDA Closure	**Context Free**	Pumping Closure
Regular expression FSM Closure	**Regular**	Pumping Closure



Introduction to Complexity Theory

Read K & S Chapter 6.

Most computational problems you will face your life are solvable (decidable). We have yet to address whether a problem is "easy" or "hard". Complexity theory tries to answer this question.

Recall that a computational problem can be recast as a language recognition problem.

Some "easy" problems:
- Pattern matching
- Parsing
- Database operations (select, join, etc.)
- Sorting

Some "hard" problems:
- Traveling salesman problem
- Boolean satisfiability
- Knapsack problem
- Optimal flight scheduling

"Hard" problems usually involve the examination of a large search space.

Big-O Notation

- Gives a quick-and-dirty measure of function size
- Used for time and space metrics

A function f(n) is O(g(n)) whenever there exists a constant c, such that $|f(n)| \leq c \cdot |g(n)|$ for all $n \geq 0$.

(We are usually most interested in the "smallest" and "simplest" function, g.)

Examples:

$2n^3 + 3n^2 \cdot \log(n) + 75n^2 + 7n + 2000$ is $\underline{O(n^3)}$

$75 \cdot 2^n + 200n^5 + 10000$ is $\underline{O(2^n)}$

A function f(n) is *polynomial* if f(n) is O(p(n)) for some polynomial function p.

If a function f(n) is not polynomial, it is considered to be *exponential*, whether or not it is O of some exponential function (e.g. $n^{\log n}$).

In the above two examples, the first is polynomial and the second is exponential.

Comparison of Time Complexities

Speed of various time complexities for different values of n, taken to be a measure of *problem size*. (Assumes 1 step per microsecond.)

f(n)\n	10	20	30	40	50	60
n	.00001 sec.	.00002 sec.	.00003 sec.	.00004 sec.	.00005 sec.	.00006 sec.
n^2	.0001 sec.	.0004 sec.	.0009 sec.	.0016 sec.	.0025 sec.	.0036 sec.
n^3	.001 sec.	.008 sec.	.027 sec.	.064 sec.	.125 sec.	.216 sec.
n^5	.1 sec.	3.2 sec.	24.3 sec.	1.7 min.	5.2 min.	13.0 min.
2^n	.001 sec.	1.0 sec.	17.9 min.	12.7 days	35.7 yr.	366 cent.
3^n	.059 sec.	58 min.	6.5 yr.	3855 cent.	$2x10^8$ cent.	$1.3x10^{13}$ cent.

Faster computers don't really help. Even taking into account Moore's Law, algorithms with exponential time complexity are considered *intractable*. ∴Polynomial time complexities are strongly desired.

Polynomial Land

If $f_1(n)$ and $f_2(n)$ are polynomials, then so are:

- $f_1(n) + f_2(n)$
- $f_1(n) \cdot f_2(n)$
- $f_1(f_2(n))$

This means that we can sequence and compose polynomial-time algorithms with the resulting algorithms remaining polynomial-time.

Computational Model

For formally describing the time (and space) complexities of algorithms, we will use our old friend, the deciding TM (decision procedure).

There are two parts:

- The problem to be solved must be translated into an equivalent language recognition problem.
- A TM to solve the language recognition problem takes an encoded instance of the problem (of size *n* symbols) as input and decides the instance in at most $T_M(n)$ steps.

We will classify the time complexity of an algorithm (TM) to solve it by its big-O bound on $T_M(n)$.

We are most interested in polynomial time complexity algorithms for various types of problems.

Encoding a Problem

Traveling Salesman Problem: Given a set of cities and the distances between them, what is the minimum distance tour a salesman can make that covers all cities and returns him to his starting city?

Stated as a decision question over graphs: Given a graph G = (V, E), a positive distance function for each edge d: E→N+, and a bound B, is there a circuit that covers all V where $\sum d(e) \leq B$? (Here a *minimization* problem was turned into a *bound* problem.)

A possible encoding the problem:

- Give |V| as an integer.
- Give B as an integer.
- Enumerate all (v_1, v_2, d) as a list of triplets of integers (this gives both E and d).
- All integers are expressed as Boolean numbers.
- Separate these entries with commas.

Note that the sizes of most "reasonable" problem encodings are polynomially related.

What about Turing Machine Extensions?

Most TM extensions are can be simulated by a standard TM in a time polynomially related to the time of the extended machine.

- k-tape TM can be simulated in $O(T^2(n))$
- Random Access Machine can be simulated in $O(T^3(n))$

(Real programming languages can be polynomially related to the RAM.)

BUT… The nondeterminism TM extension is different.

A nondeterministic TM can be simulated by a standard TM in $O(2^{p(n)})$ for some polynomial p(n).
Some faster simulation method might be possible, but we don't know it.

Recall that a nondeterministic TM can use a "guess and test" approach, which is computationally efficient at the expense of many parallel instances.

The Class P

P = { L : there is a polynomial-time deterministic TM, M that decides L }

Roughly speaking, P is the class of problems that can be solved by deterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

The way the problem is encoded or the computational abilities of the machine carrying out the algorithm are not very important.

Example: Given an integer n, is there a positive integer m, such that n = 4m?

Problems in P are considered tractable or "easy".

The Class NP

NP = { L: there is a polynomial time nondeterministic TM, M that decides L }

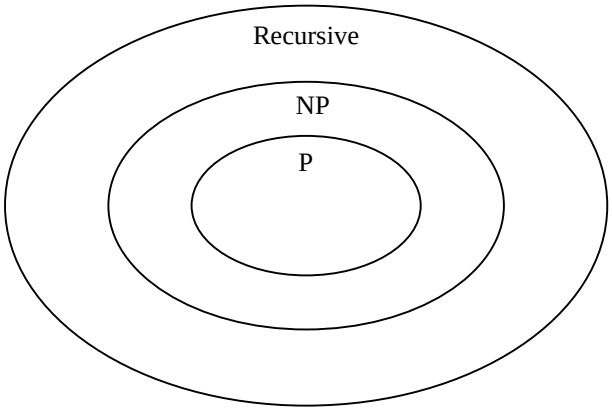
Roughly speaking, NP is the class of problems that can be solved by nondeterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

Many problems in NP are considered "intractable" or "hard".

Examples:

- **Traveling salesman problem**: Given a graph G = (V, E), a positive distance function for each edge d: E→N+, and a bound B, is there a circuit that covers all V where $\sum d(e) \leq B$?
- **Subgraph isomorphism problem**: Given two graphs G_1 and G_2, does G_1 contain a subgraph isomorphic to G_2?

The Relationship of P and NP



We're considering only solvable (decidable) problems.

Clearly P ⊆ NP.

P is closed under complement.

NP probably isn't closed under complement. Why?

Whether P = NP is considered computer science's greatest unsolved problem.

Why NP is so Interesting

- To date, nearly all decidable problems with polynomial bounds on the size of the solution are in this class.
- Most NP problems have simple nondeterministic solutions.
- The hardest problems in NP have exponential deterministic time complexities.
- Nondeterminism doesn't influence decidability, so maybe it shouldn't have a big impact on complexity.
- Showing that P = NP would dramatically change the computational power of our algorithms.

Stephen Cook's Contribution (1971)

- Emphasized the importance of polynomial time reducibility.
- Pointed out the importance of NP.
- Showed that the Boolean Satisfiability (SAT) problem has the property that every other NP problem can be polynomially reduced to it. Thus, SAT can be considered the hardest problem in NP.
- Suggested that other NP problems may also be among the "hardest problems in NP".

This "hardest problems in NP" class is called the class of "NP-complete" problems.

Further, if any of these NP-complete problems can be solved in deterministic polynomial time, they all can and, by implication, P = NP.

Nearly all of complexity theory relies on the assumption that $P \neq NP$.

Polynomial Time Reducibility

A language L_1 is *polynomial time reducible* to L_2 if there is a polynomial-time recursive function τ such that $\forall x \in L_1$ iff $\tau(x) \in L_2$.

If L_1 is polynomial time reducible to L_2, we say L_1 reduces to L_2 ("polynomial time" is assumed) and we write it as $L_1 \propto L_2$.

Lemma: If $L_1 \propto L_2$, then $(L_2 \in P) \Rightarrow (L_1 \in P)$. And conversely, $(L_1 \notin P) \Rightarrow (L_2 \notin P)$.

Lemma: If $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$.

L_1 and L_2 are *polynomially equivalent* whenever both $L_1 \propto L_2$ and $L_2 \propto L_1$.

Polynomially equivalent languages form an equivalence class. The partitions of this equivalence class are related by the partial order $\propto$.
P is the "least" element in this partial order.

What is the "maximal" element in the partial order?

The Class NP-Complete

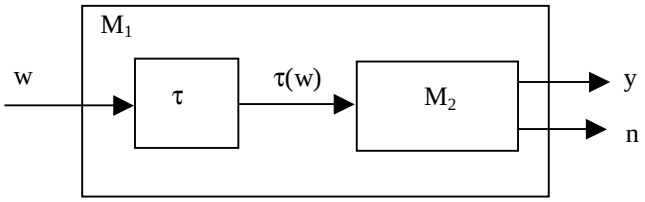
A language L is *NP-complete* if $L \in NP$ and for all other languages $L' \in NP$, $L' \propto L$.

NP-Complete problems are the "hardest" problems in NP.

Lemma: If L_1 and L_2 belong to NP, L_1 is NP-complete and $L_1 \propto L_2$, then L_2 is NP-complete.

Thus to prove a language L_2 is NP-complete, you must do the following:

1. Show that $L_2 \in NP$.
2. Select a known NP-complete language L_1.
3. Construct a reduction τ from L_1 to L_2.
4. Show that τ is polynomial-time function.



How do we get started? Is there a language that is NP-complete?

Boolean Satisfiability (SAT)

Given a set of Boolean variables $U = \{u_1, u_2, \ldots, u_m\}$ and a Boolean expression in conjunctive normal form (conjunctions of clauses—disjunctions of variables or their negatives), is there a truth assignment to U that makes the Boolean expression true (satisfies the expression)?

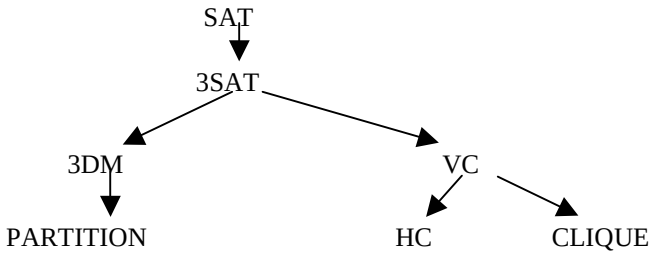
Note: All Boolean expressions can be converted to conjunctive normal form.
Example: $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_4 \vee \neg x_2)$

Cook's Theorem: SAT is NP-complete.

1. Clearly $SAT \in NP$.
2. The proof constructs a complex Boolean expression that satisfied exactly when a NDTM accepts an input string x where $|w| = n$. Because the NDTM is in NP, its running time is O(p(n)). The number of variables is polynomially related to p(n).

SAT is *NP-complete* because $SAT \in NP$ and for all other languages $L' \in NP$, $L' \propto SAT$.

Reduction Roadmap



The early NP-complete reductions took this structure. Each phrase represents a problem. The arrow represents a reduction from one problem to another.

Today, thousands of diverse problems have been shown to be NP-complete.

Let's now look at these problems.

3SAT (3-satisfiability)

Boolean satisfiability where each clause has exactly 3 terms.

3DM (3-Dimensional Matching)

Consider a set $M \subseteq X \times Y \times Z$ of disjoint sets, X, Y, & Z, such that $|X| = |Y| = |Z| = q$. Does there exist a *matching*, a subset $M' \subseteq M$ such that $|M'| = q$ and M' partitions X, Y, and Z?

This is a generalization of the marriage problem, which has two sets men & women and a relation describing acceptable marriages. Is there a pairing that marries everyone acceptably?

The marriage problem is in P, but this "3-sex version" of the problem is NP-complete.

PARTITION

Given a set A and a positive integer size, $s(a) \in N^+$, for each element, $a \in A$. Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A-A'} s(a) \ ?$$

VC (Vertex Cover)

Given a graph G = (V, E) and an integer K, such that $0 < K \leq |V|$, is there a *vertex cover* of size K or less for G, that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge, $(u, v) \in E$, at least one of u and v belongs to V'?

CLIQUE

Given a graph G = (V, E) and an integer J, such that
$0 < J \leq |V|$, does G contain a *clique* of size J or more, that is a subset $V' \subseteq V$ such that $|V'| \geq J$ and every two vertices in V' are joined by an edge in E?

HC (Hamiltononian Circuit)

Given a graph G = (V, E), does there exist a *Hamiltonian circuit*, that is an ordering $\langle v_1, v_2, \ldots, v_n \rangle$ of all V such that $(v_{|V|}, v_1) \in E$ and $(v_i, v_{i+1}) \in E$ for all i, $1 \leq i < |V|$?

Traveling Salesman Prob. is NP-complete

Given a graph G = (V, E), a positive distance function for each edge d: E→N+, and a bound B, is there a circuit that covers all V where $\sum d(e) \leq B$?

To prove a language TSP is NP-complete, you must do the following:

1. Show that $TSP \in NP$.
2. Select a known NP-complete language L_1.
3. Construct a reduction τ from L_1 to TSP.
4. Show that τ is polynomial-time function.

TSP ∈ NP: Guess a set of roads. Verify that the roads form a tour that hits all cities. Answer "yes" if the guess is a tour and the sum of the distances is ≤ B.

Reduction from HC: Answer the Hamiltonian circuit question on G = (V, E) by constructing a complete graph where "roads" have distance 1 if the edge is in E and 2 otherwise. Pose the TSP problem, is there a tour of length ≤ |V|?

Notes on NP-complete Proofs

The more NP-complete problems are known, the easier it is to find a NP-complete problem to reduce from.

Most reductions are somewhat complex.

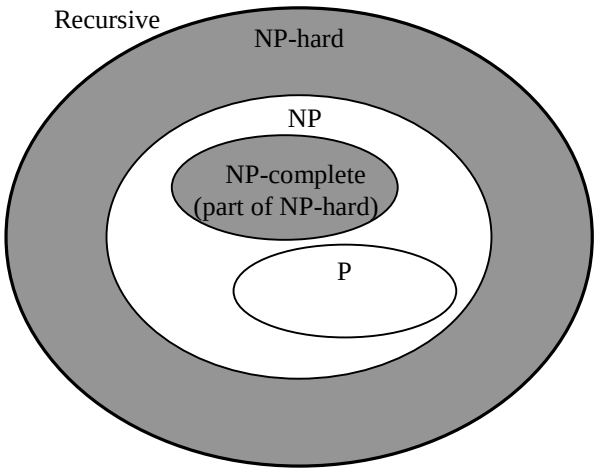
It is sufficient to show that a restricted version of the problem is NP-complete.

More Theory

NP has a rich structure that includes more than just P and NP-complete. This structure is studied in later courses on the theory of computation.

The set of recursive problems outside of NP (and including NP-complete) are called *NP-hard*. There is a proof technique to show that such problems are at least as hard as NP-complete problems.

Space complexity addresses how much tape does a TM use in deciding a language. There is a rich set of theories surrounding space complexity.



Dealing with NP-completeness

You will likely run into NP-complete problems in your career. For example, most optimization problems are NP-complete.

Some techniques for dealing with intractable problems:

- Recognize when there is a tractable special case of the general problem.
- Use other techniques to limit the search space.
- For optimization problems, seek a near-optimal solution.

The field of *linear optimization* springs out of the latter approach. Some linear optimization solutions can be proven to be "near" optimal.

A branch of complexity theory deals with solving problems within some error bound or probability.

For more: Read Computers and Intractability: A Guide to the Theory of NP-Completeness by Michael R. Garey and David S. Johnson, 1979.