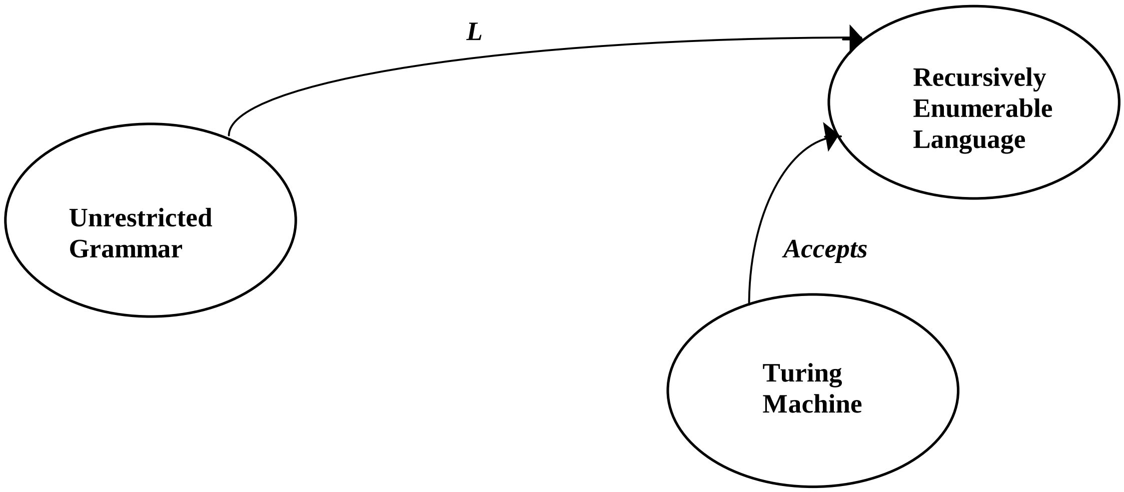
Grammars and Turing Machines

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

Another Example

L = $\{w \in \{a, b, c\}^+$: number of a's, b's and c's is the same}

S → ABCS
S → ABC
AB → BA
BC → CB
AC → CA
BA → AB
CA → AC
CB → BC
A → a
B → b
C → 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 $\rightarrow$ TM): by construction of a nondeterministic Turing machine.

If (TM $\rightarrow$ grammar): by construction of a grammar that mimics backward computations of M.

Proof that Grammar $\rightarrow$ Turing Machine

Given a grammar G, produce a Turing machine M that semidecides L(G).

M will be nondeterministic and will use two tapes:

<table>
<tr><td rowspan="4">◊</td><td>◊</td><td>❑</td><td>a</td><td>b</td><td>a</td><td>❑</td><td>❑</td><td rowspan="4">❑</td><td rowspan="4">❑</td><td rowspan="4"></td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>◊</td><td>a</td><td>S</td><td>T</td><td>a</td><td>b</td><td>❑</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

For each nondeterministic "incarnation":
- Tape 1 holds the input.
- Tape 2 holds the current state of a proposed derivation.

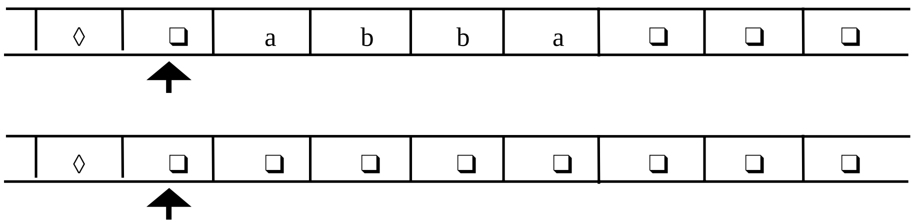
At each step, M nondeterministically chooses a rule to try to apply and a position on tape 2 to start looking for the left hand side of the rule. Or it chooses to check whether tape 2 equals tape 1. If any such machine succeeds, we accept. Otherwise, we keep looking.

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, ◊$\underline{❑}$).

We will define G so that it simulates M' backwards.
We will represent the configuration (q, ◊u$\underline{a}$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 → >❑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<
	>❑s → ε
	< → ε
(1)	❑❑q→ ❑s❑
	❑aq → ❑sa
	❑❑q< → ❑s<
(2)	a❑q → aq❑
	aaq → aqa
	a❑q< → aq<
(3)	t❑❑ → ❑q❑
	t❑a → ❑qa
	t< → ❑q<
(4)	❑p → at
(5)	❑h → ❑t
(6)	t❑❑ → ❑p❑
	t❑a → ❑pa
	t< → ❑p<

Working It Out

	S →>❑h<	1
	>❑s → ε	2
	< → ε	3
(1)	❑❑q→ ❑s❑	4
	❑aq → ❑sa	5
	❑❑q< → ❑s<	6
(2)	a❑q → aq❑	7
	aaq → aqa	8
	a❑q< → aq<	9
(3)	t❑❑ → ❑q❑	10
	t❑a → ❑qa	11
	t< → ❑q<	12
(4)	❑p → at	13
(5)	❑h → ❑t	14
(6)	t❑❑ → ❑p❑	15
	t❑a → ❑pa	16
	t< → ❑p<	17

>❑saa<	1
>❑aqa<	2
>❑aaq<	2
>❑aa❑q<	3
>❑aat<	4
>❑a❑p<	6
>❑at<	4
>❑❑p<	6
>❑t<	5
>❑h<	

S	⇒ >❑h<	1
	⇒ >❑t<	14
	⇒ >❑❑p<	17
	⇒ >❑at<	13
	⇒ >❑a❑p<	17
	⇒ >❑aat<	13
	⇒ >❑aa❑q<	12
	⇒ >❑aaq<	9
	⇒ >❑aqa<	8
	⇒ >❑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

$w = ◊ ◊ ❑ ❑ Qs\ a_1\ a_1\ a_2\ a_2\ a_3\ a_3\ ❑ ❑ \ldots$

The second (test) part of G simulates the execution of M on a particular string w. An example of a partially derived string:

◊ ◊ ❑ ❑ a 1 b 2 c c b 4 Q3 a 3

Examples of rules:

b b Q 4 → b 4 Q 4 (rewrite b as 4)
b 4 Q 3 → 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 → 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 ∈ Σ*,

$SwS \Rightarrow_G^* v$ 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 → 11S
SS → ε

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:
  $f(n_1,n_2,\ldots n_k, \quad 0) = g(n_1,n_2,\ldots n_k)$
  $f(n_1,n_2,\ldots n_k,m+1) = h(n_1,n_2,\ldots n_k, m, f(n_1, n_2,\ldots n_k,m))$

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, ….
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

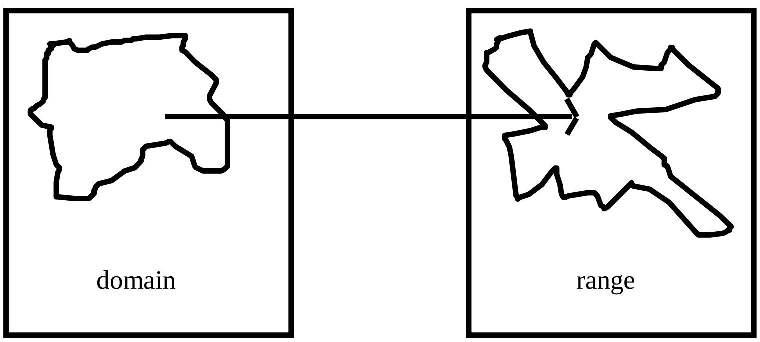
A function g is **minimalizable** iff for every $n_1,n_2,\ldots n_k$, there is an m such that $g(n_1,n_2,\ldots n_k,m)=1$.

Theorem: A function is μ-recursive iff it is recursive (i.e., computable by a Turing machine).

Partial Recursive Functions

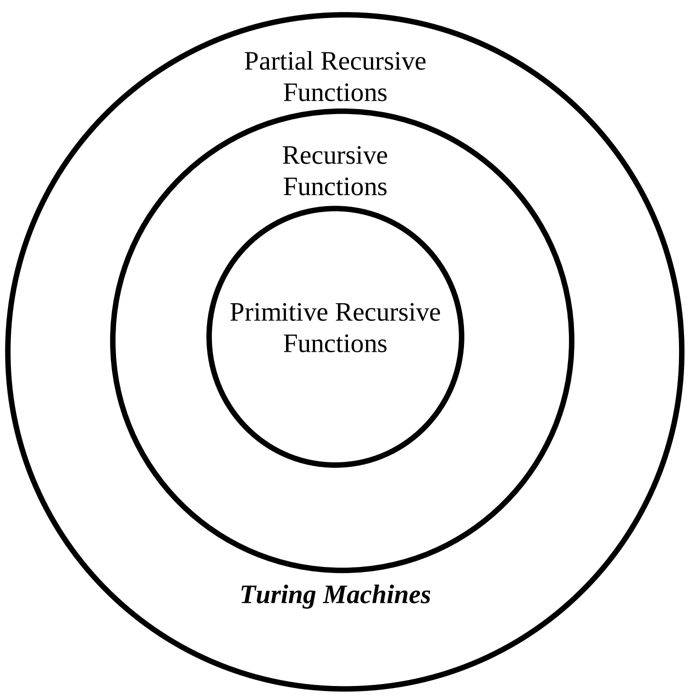
Consider the following function f:
f(n) = 1 if TM(n) halts on a blank tape
0 otherwise

The domain of f is the natural numbers. Is f recursive?

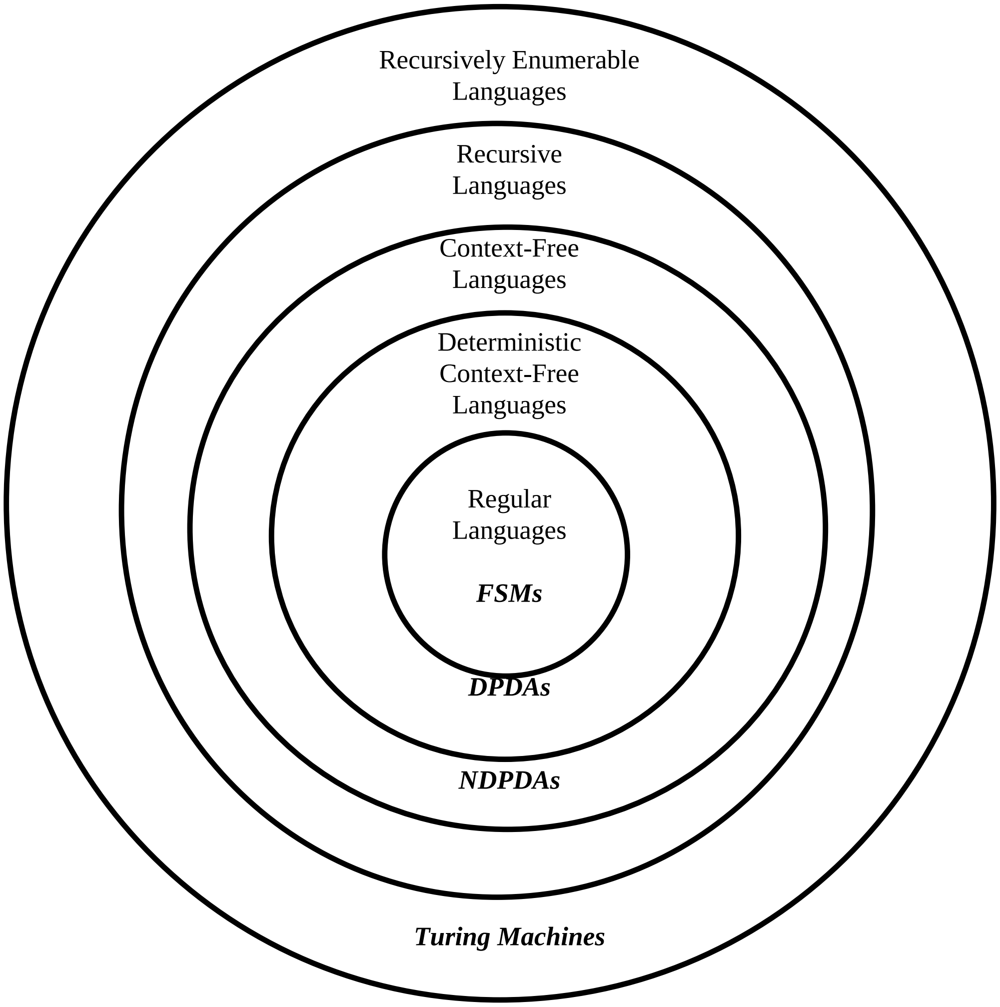


Theorem: There are uncountably many partially recursive functions (but only countably many Turing machines).

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

$x \to y$

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:

$L = \{a^n b^n c^n, n > 0\}$

$L = \{w \in \{a, b, c\}^+$: number of a's, b's and c's is the same$\}$

Context-Sensitive Languages are Recursive

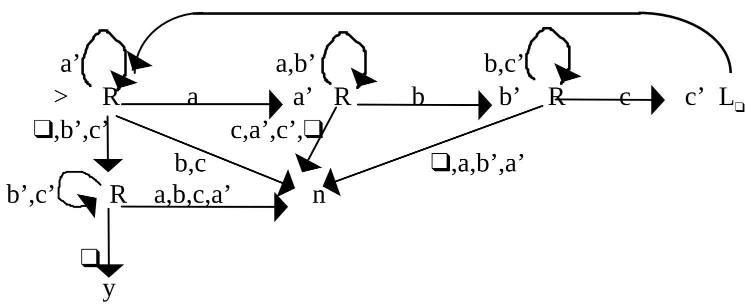
The basic idea: To decide if a string w is in L, start generating strings systematically, shortest first. If you generate w, accept. If you get to strings that are longer than w, reject.

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❑❑❑❑❑❑❑❑❑



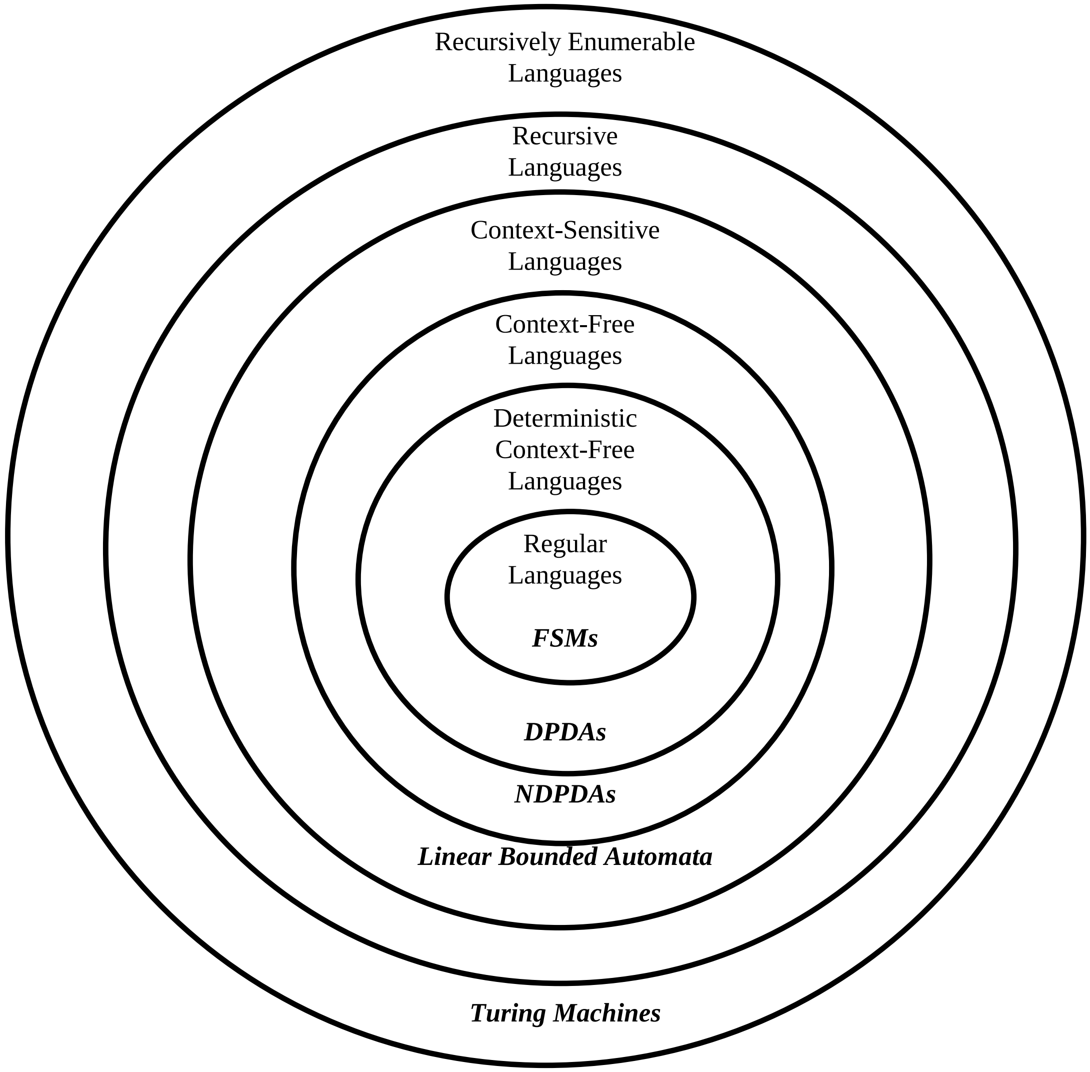
Context-Sensitive Languages and Linear Bounded Automata

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