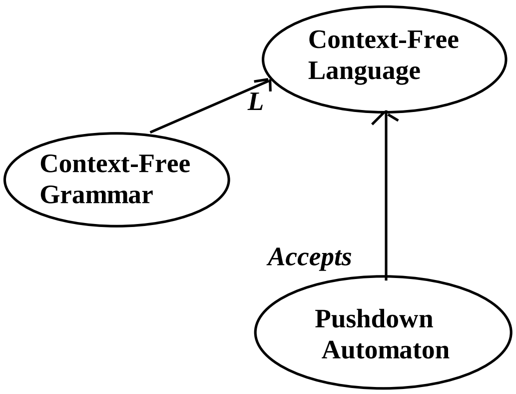
Context-Free Grammars

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.

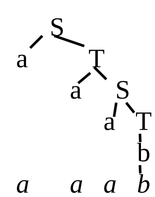
Context-Free Grammars, Languages, and Pushdown Automata



Grammars Define Languages

Think of grammars as either generators or acceptors.

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

	Regular Expression	**Regular Grammar**
	$(aa \cup ab \cup ba \cup bb)^*$	$S \rightarrow \varepsilon$ $S \rightarrow aT$ $S \rightarrow bT$ $T \rightarrow a$ $T \rightarrow b$ $T \rightarrow aS$ $T \rightarrow bS$
Derivation (Generate)	choose aa choose ab yields *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

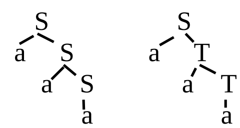
$(a \cup b)^*a\,(a \cup b)^*$

choose a from $(a \cup b)$
choose a from $(a \cup b)$
choose a

choose a
choose a from $(a \cup b)$
choose a from $(a \cup b)$

Regular Grammar

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



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

$S \rightarrow aBa$ $B \rightarrow \varepsilon$ $B \rightarrow bB$	vs.	$S \rightarrow aB$ $B \rightarrow a$ $B \rightarrow bB$

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

$S \rightarrow B$
$S \rightarrow aSa$
$B \rightarrow \varepsilon$
$B \rightarrow bB$

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.

$S \rightarrow abDeFGab$

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

$S \rightarrow$

but not $ASB \rightarrow$ or $aSb \rightarrow$ or $ab \rightarrow$

Examples:

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

$a^n b^n$
$S \rightarrow a\,S\,b$
$S \rightarrow \varepsilon$

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 $V - \Sigma$.

$\mathbf{x \Rightarrow_G y}$ 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

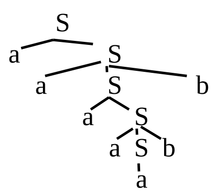
W = {S} ∪ Σ,

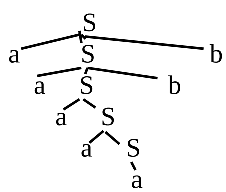
Σ = {a, b},

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

$S \rightarrow NP\ \ VP$
$NP \rightarrow$ the $NP1 \mid NP1$
$NP1 \rightarrow ADJ\ \ NP1 \mid N$
$ADJ \rightarrow$ big | youngest | oldest
$N \rightarrow$ boy | boys
$VP \rightarrow V \mid V\ \ NP$
$V \rightarrow$ run | runs

the boys run
big boys run
the youngest boy runs

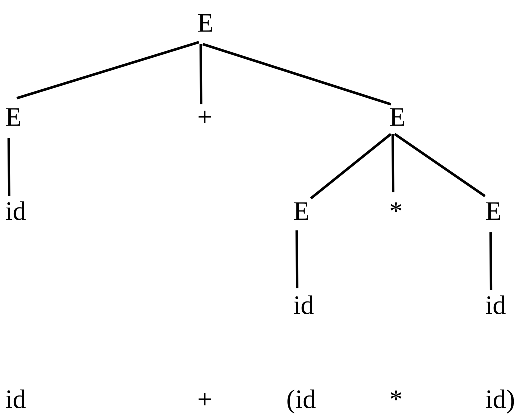
the youngest oldest boy runs
the boy run

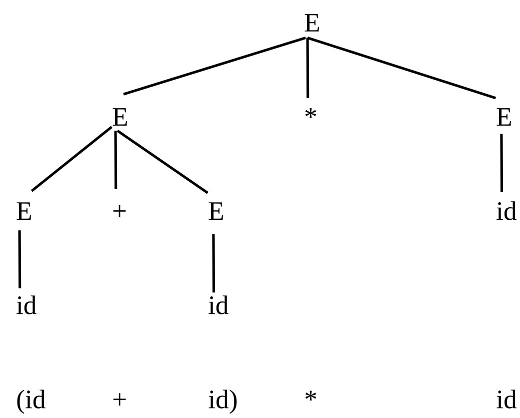
Who did you say Bill saw coming out of the hotel?

Arithmetic Expressions

The Language of Simple Arithmetic Expressions

$G = (V, \Sigma, R, E)$, where
$V = \{+, *, id, T, F, E\}$,
$\Sigma = \{+, *, id\}$,
$R = \{\quad E \rightarrow id$
$E \rightarrow E + E$
$E \rightarrow E * E\ \}$





Arithmetic Expressions -- A Better Way

The Language of Simple Arithmetic Expressions

$G = (V, \Sigma, R, E)$, where
$V = \{+, *, (,), id, T, F, E\}$,
$\Sigma = \{+, *, (,), id\}$,
$R = \{\quad E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id\quad \}$

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:

$\langle\text{expression}\rangle \rightarrow \langle\text{expression}\rangle + \langle\text{term}\rangle$
$\langle\text{expression}\rangle \rightarrow \langle\text{term}\rangle$
$\langle\text{term}\rangle \rightarrow \langle\text{term}\rangle * \langle\text{factor}\rangle$
$\langle\text{term}\rangle \rightarrow \langle\text{factor}\rangle$
$\langle\text{factor}\rangle \rightarrow (\langle\text{expression}\rangle)$
$\langle\text{factor}\rangle \rightarrow \langle\text{id}\rangle$

The Language of Boolean Logic

$G = (V, \Sigma, R, E)$, where

$V = \{\wedge, \vee, \neg, \Rightarrow, (,), \text{id}, E, E1, E2, E3, E4\}$,
$\Sigma = \{\wedge, \vee, \neg, \Rightarrow, (,), \text{id}\}$,
$R = \{$ $E \rightarrow E \Rightarrow E1$
$E \rightarrow E1$
$E1 \rightarrow E1 \vee E2$
$E1 \rightarrow E2$
$E2 \rightarrow E2 \wedge E3$
$E2 \rightarrow E3$
$E3 \rightarrow \neg E4$
$E3 \rightarrow E4$
$E4 \rightarrow (E)$
$E4 \rightarrow \text{id}$ $\}$

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 $2N + 1 + 2|\text{id}|$ of the form:

$$w = \underbrace{\underline{((((((}}_{N}\ \text{id}\ ))))))\ \Rightarrow \text{id}$$

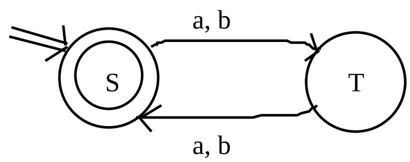
x y

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

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:

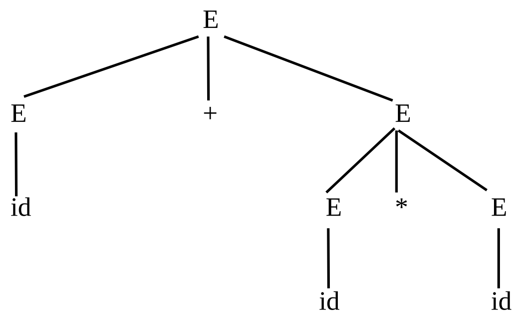
We care about recognizing patterns and taking appropriate actions.

Example: A parity checker

Structure

Context free languages:

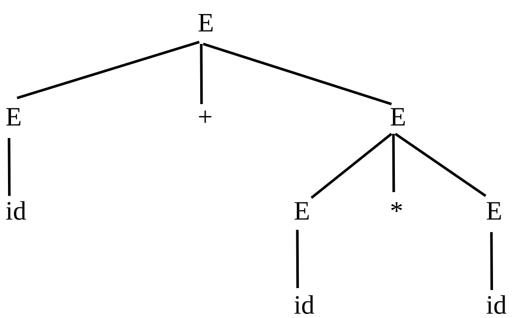
We care about structure.

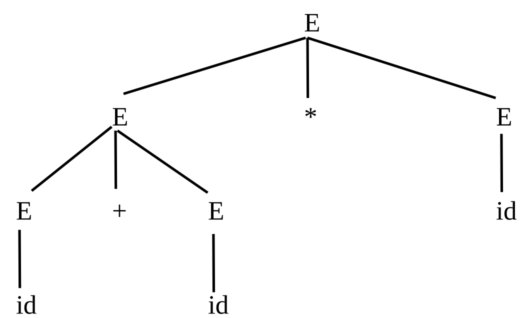


id + (id * id)

Parse Trees Capture Essential Structure

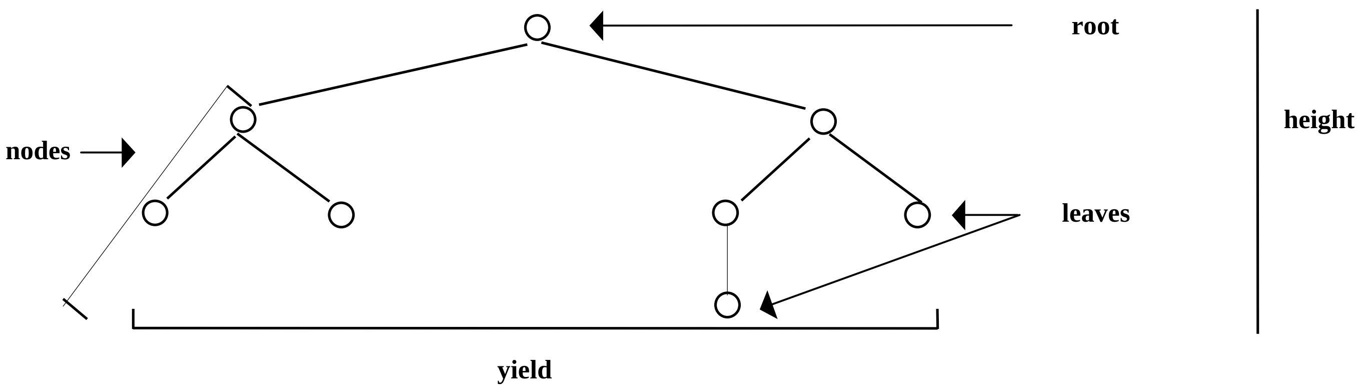
$E \rightarrow id$
$E \rightarrow E + E$
$E \rightarrow E * E$





id + (id * id)   (id + id) * id

Parse Trees are Just Trees



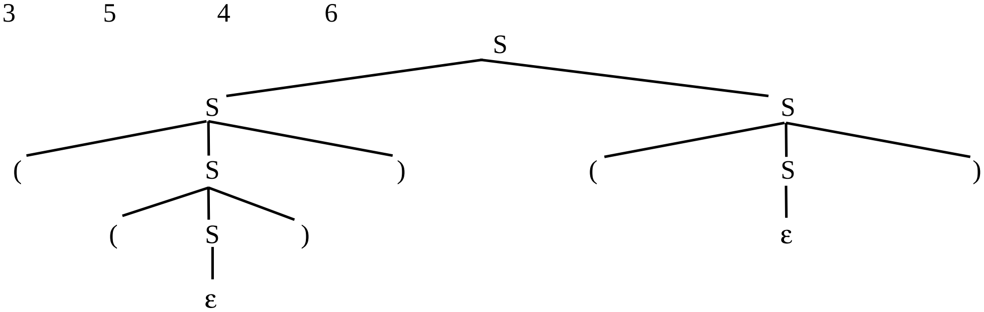
Leaves are all labeled with terminals or ε.
Other nodes are labeled with nonterminals.
A **path** is a sequence of nodes, starting at the root, ending at a leaf, and following branches in the tree.
The length of the yield of any tree T with height H and branching factor (fanout) B is ≤

Derivations

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

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

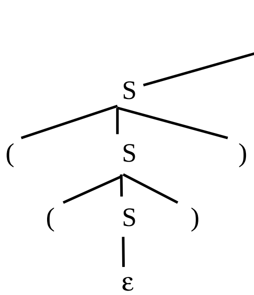
1 2 3 4 5 6
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$
1 2 3 5 4 6

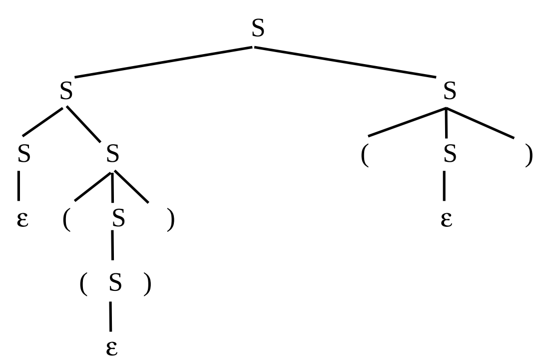


Alternative Derivations

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

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





Ordering Derivations

Consider two derivations:

1 2 3 4 5 6 7

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

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

1 2 3 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 $i \neq k$, $x_i = x_i'$
  - $x_{k-1} = x'_{k-1} = uAvBw$: $u, v, w \in V^*$, and $A, B \in V - \Sigma$
  - $x_k = uyvBw$, 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) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$

(2) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

(3) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$

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

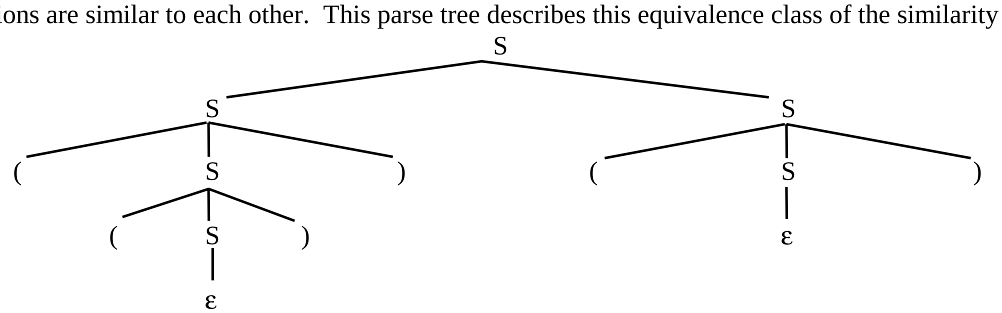
(2) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

(3) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$

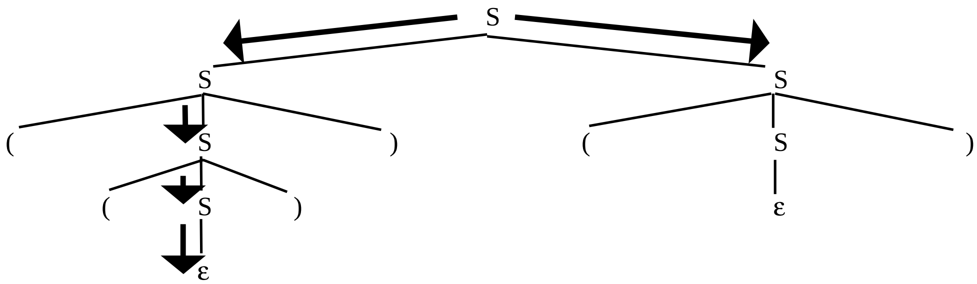
D1 < D2

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.

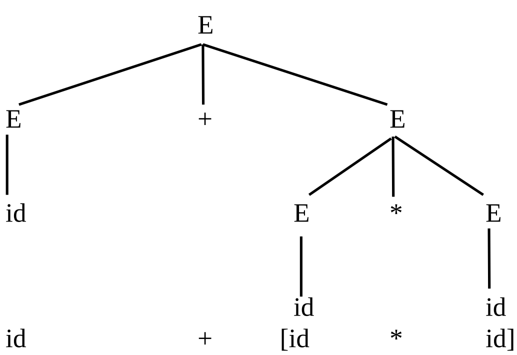
We call this the **leftmost derivation**. There is a corresponding rightmost derivation.

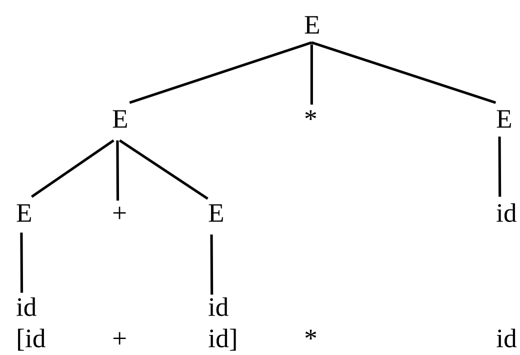
The leftmost (rightmost) derivation can be used to construct the parse tree and the parse tree can be used to construct the leftmost (rightmost) derivation.

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   Parse :   id + id * id

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

Σ = {+, *, (,), id},

R = { $E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$ }

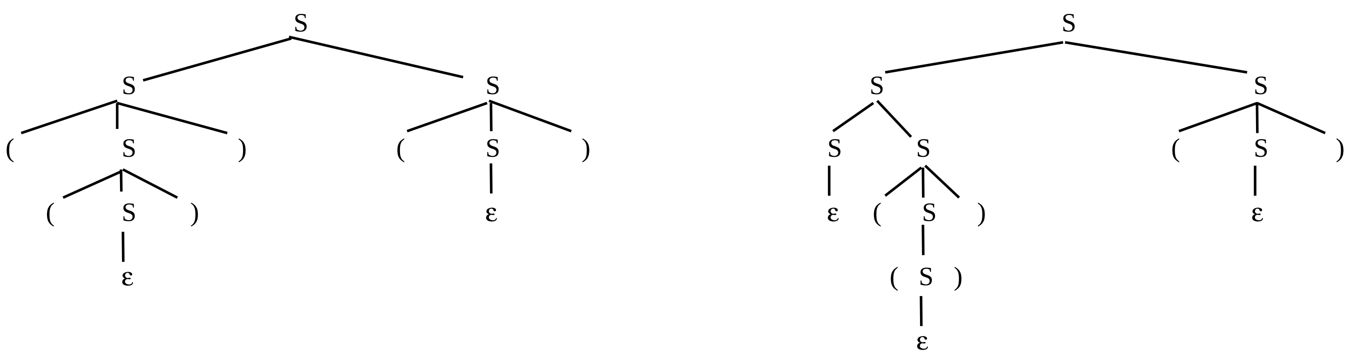
Another Example

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:

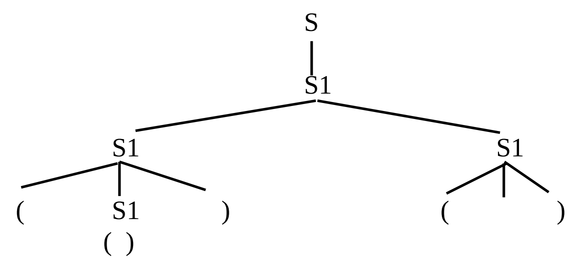
$S \rightarrow \varepsilon$

$S \rightarrow S_1$

$S_1 \rightarrow S_1 S_1$

$S_1 \rightarrow (S_1)$

$S_1 \rightarrow ()$



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.

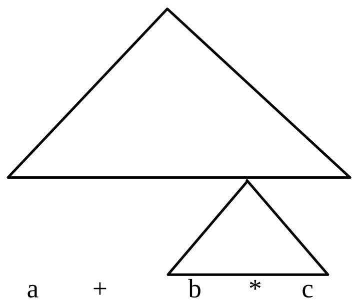
Pushdown Automata

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:

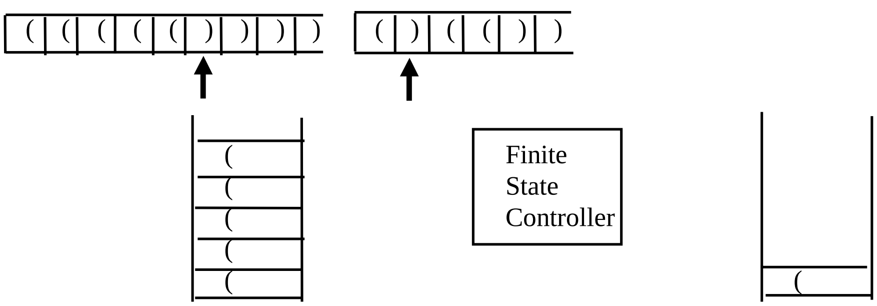
(1) Say yes or no, just like with FSMs
(2) Say yes or no, AND
if yes, describe the structure



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

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

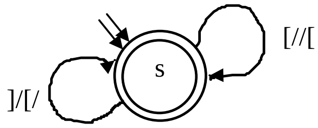
K is a finite set of states
Σ is the input alphabet
Γ is the stack alphabet
$s \in K$ is the initial state
$F \subseteq K$ is the set of final states, and
Δ is the transition relation. It is a finite subset of

$$\underbrace{(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*)}_{\text{state} \quad \text{input or } \varepsilon \quad \text{string of symbols to pop from top of stack}} \times \underbrace{(K \times \Gamma^*)}_{\text{state} \quad \text{string of symbols to push on top of stack}}$$

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:

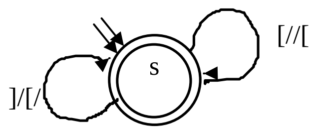
- K = {s} the states
- Σ = {[,]} the input alphabet
- Γ = {[} the stack alphabet
- F = {s}
- Δ contains:

((s, [, ε), (s, [ ))
((s,], [ ), (s, ε))

Important:

This does not mean that the stack is empty.

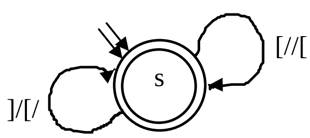
An Example of Accepting



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 <u>reject</u>.

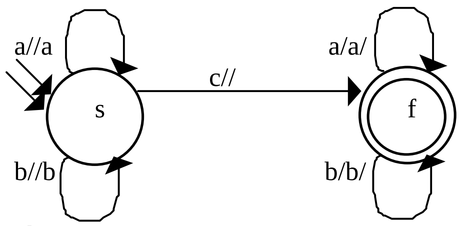
A PDA for $a^n b^n$

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:



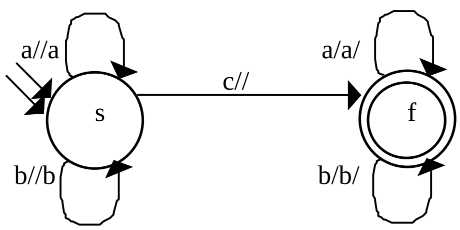
M = (K, Σ, Γ, Δ, s, F), where:

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

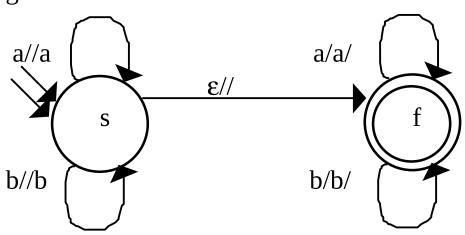
trans	***state***	***unread input***	***stack***
	s	b a c a b	ε
2	s	a c a b	b
1	s	c a b	ab
3	f	a b	ab
5	f	b	b
6	f	ε	ε

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:



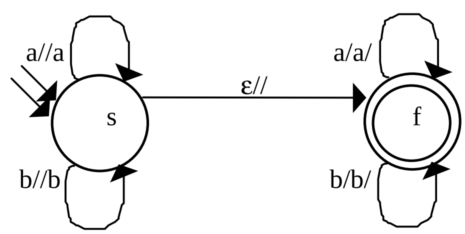
M = (K, Σ, Γ, Δ, s, F), where:

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

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

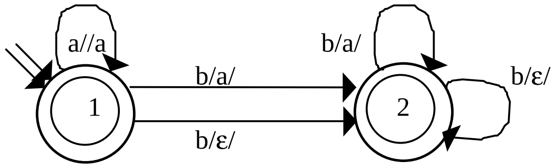
A context-free grammar for L:

$S \rightarrow \varepsilon$

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

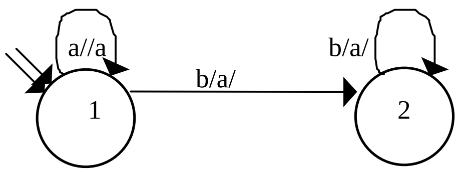
$S \rightarrow aSb$

A PDA to accept L:

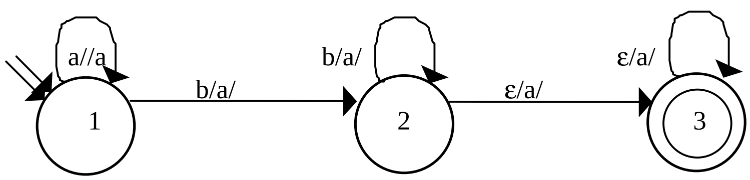


Accepting Mismatches

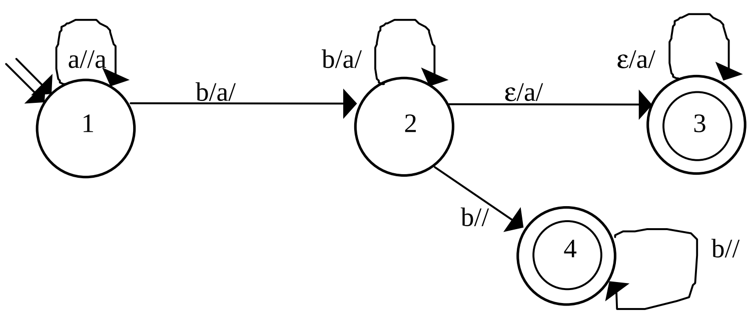
$L = \{a^m b^n \; m \neq n; \; m, n > 0\}$



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

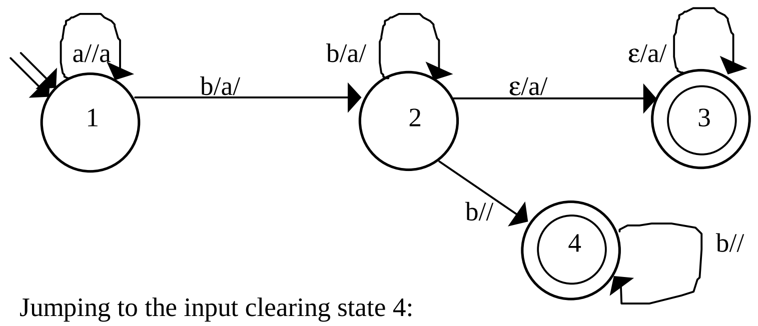


- If stack is empty but input is not (m < n) (accept):

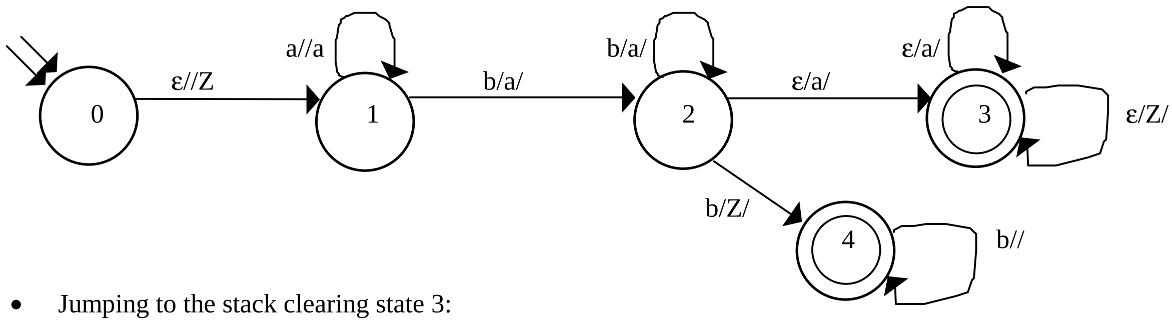


Eliminating Nondeterminism

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.

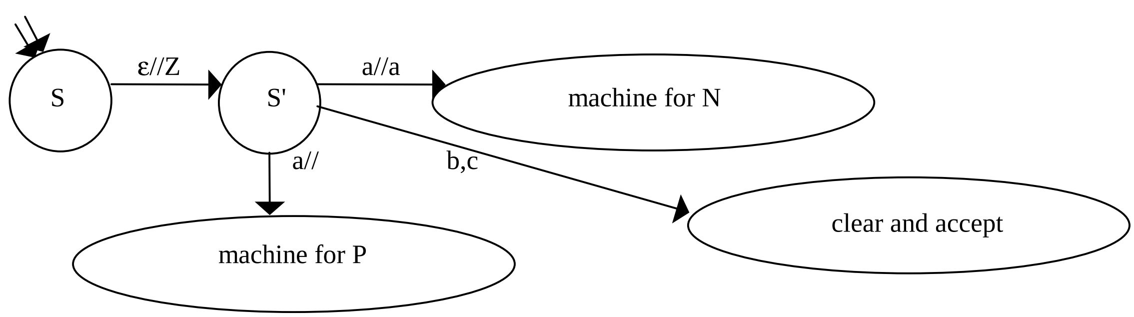


- Jumping to the stack clearing state 3:
  Need to detect end of input. To do that, we actually need to modify the definition of L to add a termination character (e.g., $)

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

$S \rightarrow NC$	/* $n \neq m$, then arbitrary c's	$C \rightarrow \varepsilon \mid cC$	/* add any number of c's
$S \rightarrow QP$	/* arbitrary a's, then $p \neq m$	$P \rightarrow B'$	/* more b's than c's
$N \rightarrow A$	/* more a's than b's	$P \rightarrow C'$	/* more c's than b's
$N \rightarrow B$	/* more b's than a's	$B' \rightarrow b$	
$A \rightarrow a$		$B' \rightarrow bB'$	
$A \rightarrow aA$		$B' \rightarrow bB'c$	
$A \rightarrow aAb$		$C' \rightarrow c \mid C'c$	
$B \rightarrow b$		$C' \rightarrow C'c$	
$B \rightarrow Bb$		$C' \rightarrow bC'c$	
$B \rightarrow aBb$		$Q \rightarrow \varepsilon \mid aQ$	/* 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:

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

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

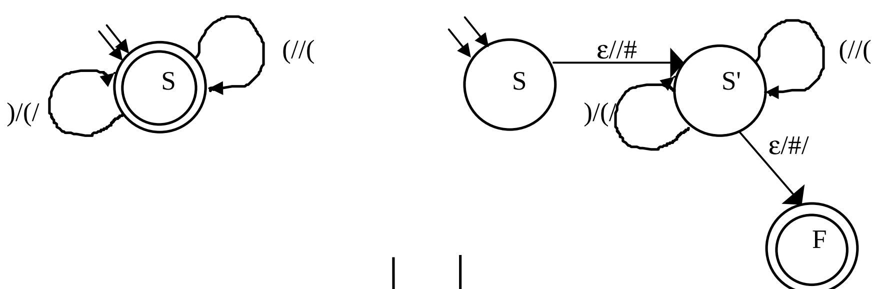
We construct a PDA M' = (K, Σ, Γ, Δ, s, F)
where Γ = ∅ /* stack alphabet
and
each transition (p, i, q) becomes
((p, i, ε), (q, ε))
old state, input, don't look at stack new state don't push on stack

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

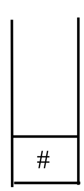
Alternative (but Equivalent) Definitions of a NDPDA

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:



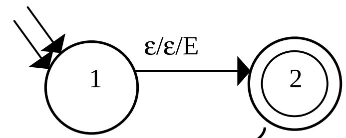
The new machine is nondeterministic:
( ) ( )
⇑
The stack will be:



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 $\Leftrightarrow$ Can accept by PDA

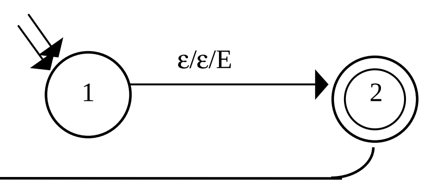
Going One Way

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 $G = (V, \Sigma, R, S)$
Construct M such that $L(M) = L(G)$

$M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

(1) $((p, \varepsilon, \varepsilon), (q, S))$
push the start symbol on the stack

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

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

input = a a b b a a

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

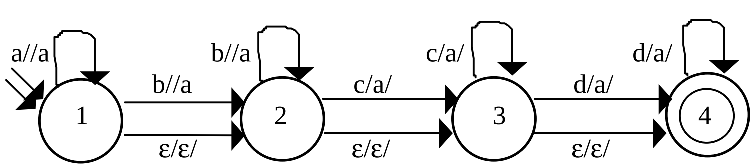
(1)	$S \to aSd$	0	(p, ε, ε), (q, S)
(2)	$S \to T$	1	(q, ε, S), (q, aSd)
(3)	$S \to U$	2	(q, ε, S), (q,T)
(4)	$T \to aTc$	3	(q, ε, S), (q,U)
(5)	$T \to V$	4	(q, ε, T), (q, aTc)
(6)	$U \to bUd$	5	(q, ε, T), (q, V)
(7)	$U \to V$	6	(q, ε, U), (q, bUd)
(8)	$V \to bVc$	7	(q, ε, U), (q, V)
(9)	$V \to \varepsilon$	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)

input = a a b c d d

The Other Way—Build a PDA Directly

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

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



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) $((p, \varepsilon, \varepsilon), (q, S))$
(1) $((q, \varepsilon, S), (q, aSb))$
(2) $((q, \varepsilon, S), (q, \varepsilon))$
(3) $((q, a, a), (q, \varepsilon))$
(4) $((q, b, b), (q, \varepsilon))$

But transitions 1 and 2 make M nondeterministic.

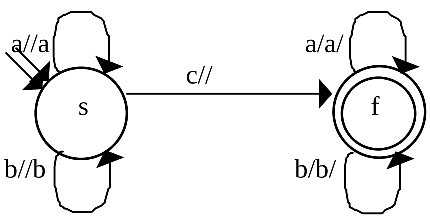
A **nondeterministic transition group** is a set of two or more transitions out of the same state that can fire on the same configuration. A **PDA is nondeterministic** if it has any nondeterministic transition groups.

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



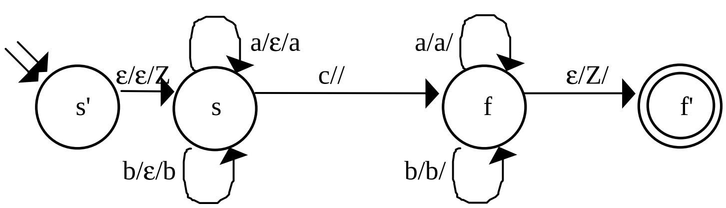
Δ contains:
$((s, a, \varepsilon), (s, a))$
$((s, b, \varepsilon), (s, b))$
$((s, c, \varepsilon), (f, \varepsilon))$
$((f, a, a), (f, \varepsilon))$
$((f, b, b), (f, \varepsilon))$

$M = (\{s, f\}, \{a, b, c\}, \{a, b\}, \Delta, s, \{f\})$, where:

First Step: Make M Simple

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

Step 1: Add s' and f':

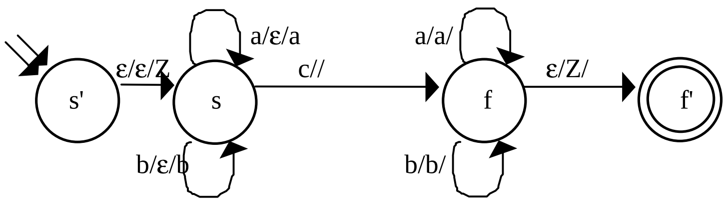


Step 2:
(1) Assure that $|\beta| \le 1$.

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

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

Making M Simple

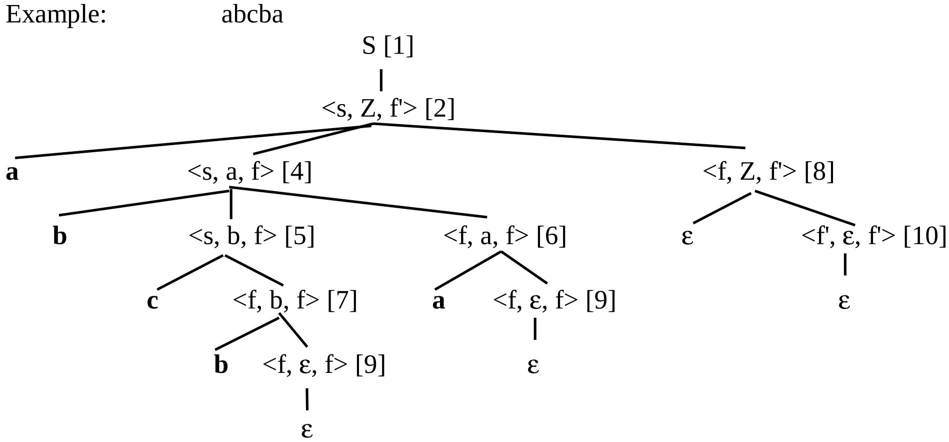


M = ({s, f, s', f'}, {a, b, c}, {a, b, Z}, Δ, s',{f'}), Δ=

	((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 → <s, Z, f'>	[1]
((s', ε, ε), (s, Z))		
((s, a, Z), (s, aZ))	<s, Z, f'> → a <s, a, f> <f, Z, f'>	[2]
	<s, Z, s> → a <s, a, f> <f, Z, s>	[x]
	<s, Z, f> → a <s, a, s> <s, Z, f>	[x]
	<s, Z, s> → a <s, a, s> <s, Z, f>	[x]
	<s, Z, s'> → a <s, a, f> <f, Z, s'>	[x]
((s, a, a), (s, aa))	<s, a, f> → a <s, a, f> <f, a, f>	[3]
((s, a, b), (s, ab))	…	
((s, b, Z), (s, bZ))	…	
((s, b, a), (s, ba))	<s, a, f> → b <s, b, f> <f, a, f>	[4]
((s, b, b), (s, bb))	…	
((s, c, Z), (f, Z))	…	
((s, c, a), (f, a))	<s, a, f> → c <f, a, f>	
((s, c, b), (f, b))	<s, b, f> → c <f, b, f>	[5]
((f, a, a), (f, ε))	<f, a, f> → a <f, ε, f>	[6]
((f, b, b), (f, ε))	<f, b, f> → b <f, ε, f>	[7]
((f, ε, Z), (f', ε))	<f, Z, f'> → ε <f', ε, f'>	[8]
	<f, ε, f> → ε	[9]
	<f' ε, f'> → 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.

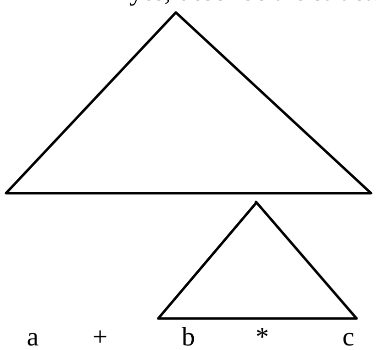
Recognizing Context-Free Languages

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.,
    - $A \to aB$, $A \to aC$ become $A \to aD$, $D \to B \mid C$
- 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

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

becomes

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

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 $M_2, M_3, \ldots M_{n-1}$.
   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:

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

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

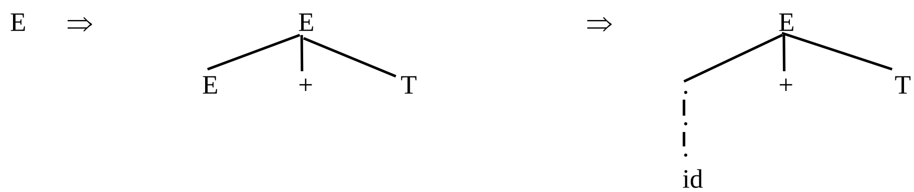
Top Down Parsing

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

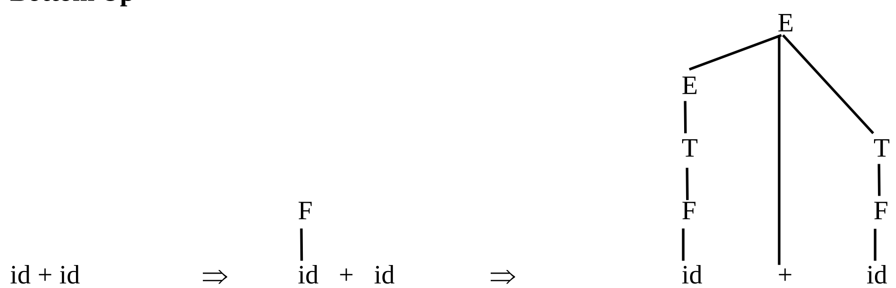
Parsing

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]	$F \rightarrow id(E)$

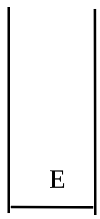
A PDA that does a top down parse:

(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)

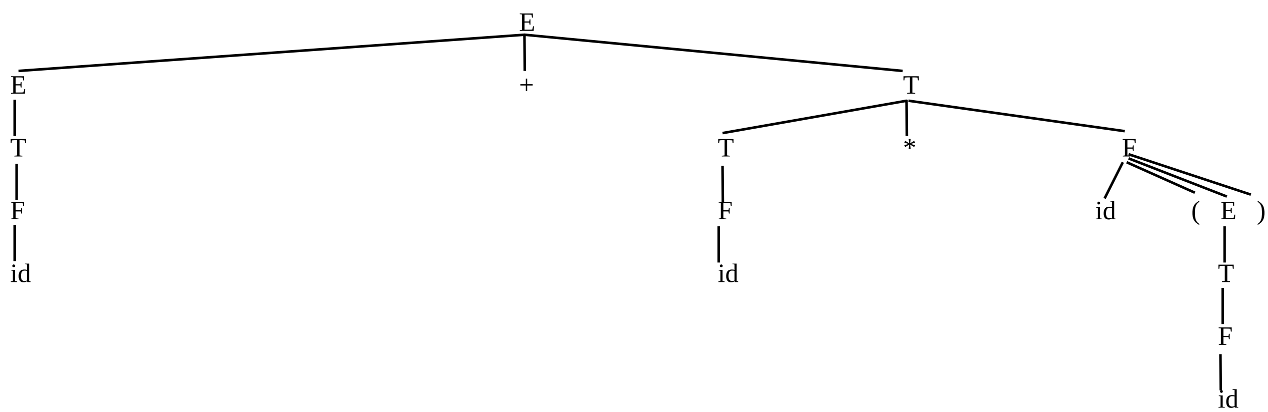
Stack:



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) nondeterministic
(2) (2, ε, E), (2, T)
(3) (2, ε, T), (2, T*F) nondeterministic
(4) (2, ε, T), (2, F)
(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id) nondeterministic
(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)

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)
$\Uparrow$

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

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:

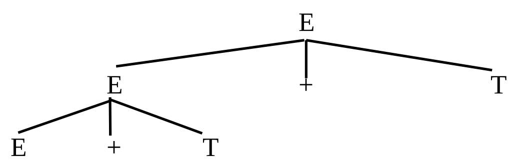
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1$
$A' \rightarrow \beta_2 \ldots$
$A' \rightarrow \beta_n$

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:

$A \rightarrow A\alpha_1$
$A \rightarrow A\alpha_2$ …
$A \rightarrow A\alpha_3$
$A \rightarrow A\alpha_n$

$A \rightarrow \beta_1$ (where β's do not start with $A\alpha$)
$A \rightarrow \beta_2$
…
$A \rightarrow \beta_m$

and $n > 0$, then

Replace them with:

$A' \rightarrow \alpha_1 A'$
$A' \rightarrow \alpha_2 A'$ …
$A' \rightarrow \alpha_3 A'$
$A' \rightarrow \alpha_n A'$
$A' \rightarrow \varepsilon$
$A \rightarrow \beta_1 A'$
$A \rightarrow \beta_2 A'$
…
$A \rightarrow \beta_m A'$

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 → E + T
[2] E → T

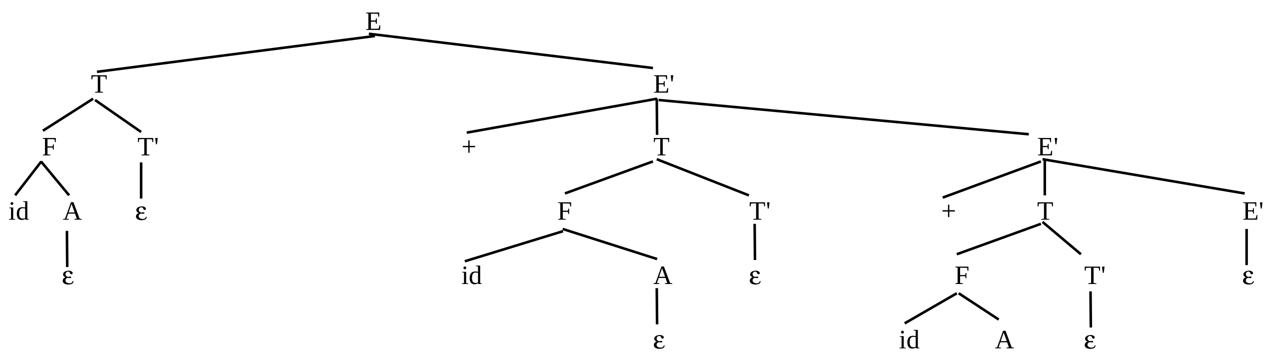
[3] T → T * F
[4] T → F

[5] F → (E)
[6] F → id
[7] F → id(E)

input = id + id + id

New Grammar

E → TE'
E' → +TE'
E' → ε
T → FT'
T' → *FT'
T' → ε
F → (E)
F → idA
A → ε
A → (E)



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

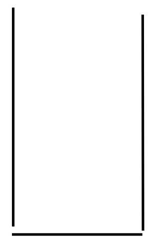
Bottom Up Parsing

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]	$F \rightarrow id$



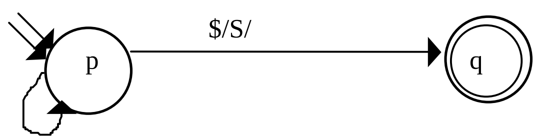
id + id * id $

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):

$((p, \$, S), (q, \varepsilon))$

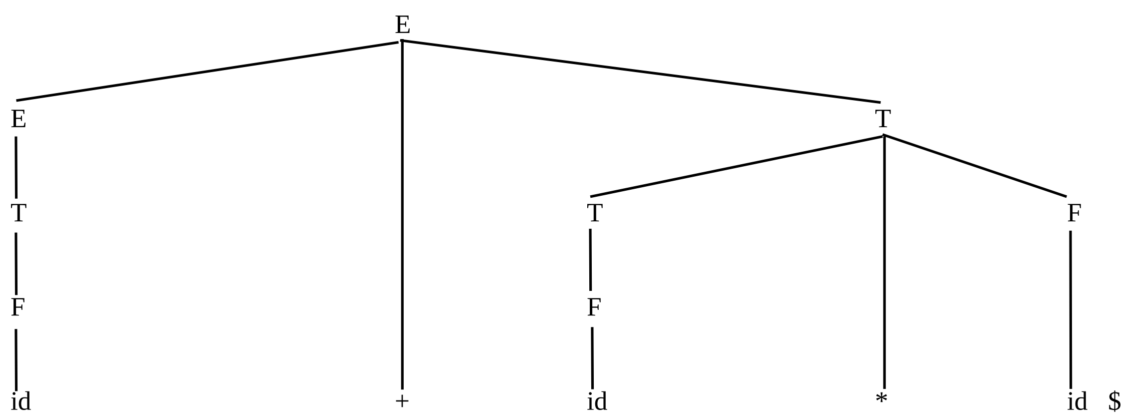
(This is the “bottom-up” CFG to PDA conversion algorithm.)

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

$$P \subseteq (\underbrace{V}_{\text{top stack symbol}} \times \underbrace{\{\Sigma \cup \$\}}_{\text{next input symbol}})$$

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

a
γ

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
⇑*   Input: id * id * id
E

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

if C_1 then if C_2 then ST_1 else ST_2

↑1 ↑2

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:

ST2	2
else	
ST1	1
then	
C2	
if	
then	
C1	
if	

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 $\overset{T}{\overset{\Uparrow}{\overline{\underset{\underset{T}{\Downarrow}}{F} \; * \; T}}} \; + \; E$

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 **<u>L</u>**eft to right and build a **<u>R</u>**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.

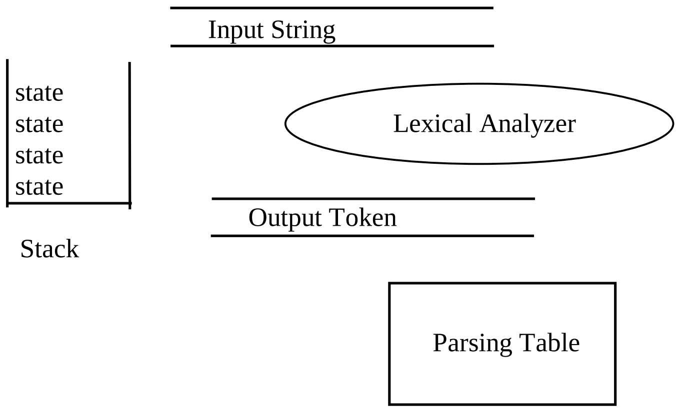
An LALR (or Look Ahead LR) parser is a specific kind of LR parser that has two desirable properties:
- The parsing table is not huge.
- Most useful languages can be parsed.

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   ⇐ *the rule this came from*
    DING shift 3   *state 3*
    . error   *current position of input*
    rhyme goto 1   *if none of the others match*
    sound goto 2   *push state 2*
state 1 *(rhyme)*
    $accept : rhyme_$end
    $end accept   *if we see EOF, 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   *by rule 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)
↑

id
(
procname

Should we reduce id by rule 4 or rule 7?

procid
proc(
procname

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:

Input: -(17 * 83.56) + 72 / 12

↓

Lexical analysis of the input	✷1

↓

A string of input tokens, corresponding to the primitive objects of which the input is composed:

-(id * id) + id / id

↓

Parse the input	✷2

↓

A tree of actions, whose structure corresponds to the structure of the input.

↓

Compute the answer

Output: -1414.52

yacc and lex

Lexical analysis of the input	✷1
Parse the input	✷2

Where do the procedures to do these things come from?

regular expressions that describe patterns

↓

lex

↓

lexical analyzer ✷1

grammar rules and other facts about the language

↓

yacc

↓

parser ✷2

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:	<u>integers</u>	take action 2
	<u>integer</u>	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

$$ST \rightarrow \text{if } C \text{ then } ST \text{ else } ST$$
$$ST \rightarrow \text{if } C \text{ then } ST$$

What if the input is

if C_1 then if C_2 then ST_1 else ST_2

↑1 ↑2

Which bracketing (rule) should we choose?

yacc will choose to shift rather than reduce.

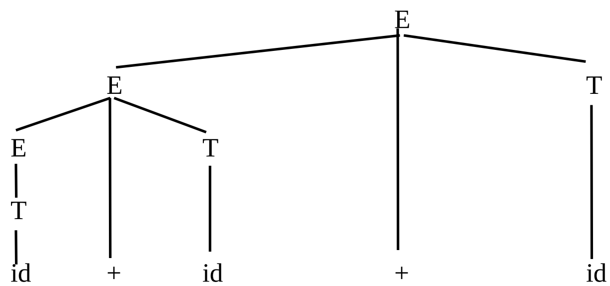
ST2	2
else	
ST1	1
then	
C2	
if	
then	
C1	
if	

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:

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

Shift/Reduce Conflicts - Operator Precedence

Recall the problem: input: id + id * id

T
+
E

Should we reduce or shift on * ?

The "always shift" rule solves this problem.

But what about: id * id + id

T
*
E

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.

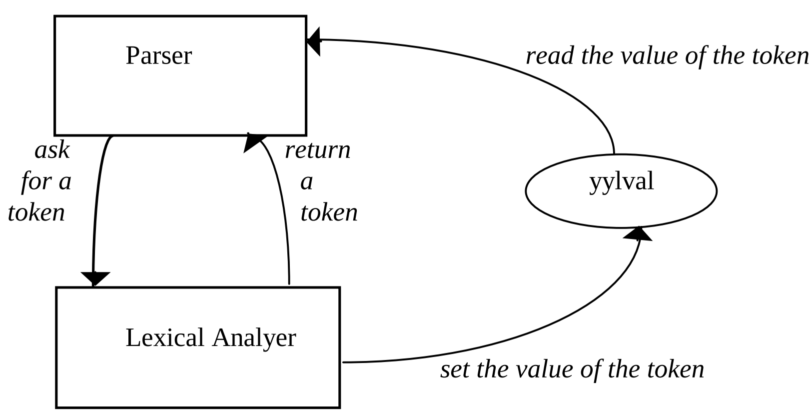
Step 2:

$ 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: $\{a^nb^nc^n\}$

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:

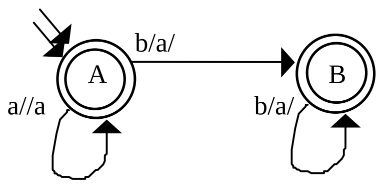
For each rule $((q_1,\ a, \beta),\ (p_1,\ \gamma))$ in Δ_1,
and each rule $(q_2,\ a,\ p_2)$ in δ,
$(((q_1, q_2),\ a, \beta), ((p_1, p_2), \gamma))$

For each rule $((q_1,\ \varepsilon, \beta),\ (p_1,\ \gamma)$ in Δ_1,
and each state q_2 in K_2,
$(((q_1, q_2), \varepsilon, \beta), ((p_1, q_2), \gamma))$

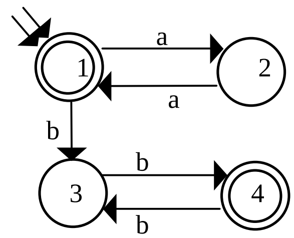
This works because: we can get away with only one stack.

Example

$L = a^nb^n \cap (aa)^*(bb)^*$



((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

$$L_3 = \underline{L_1} \cup \underline{L_2}.$$

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

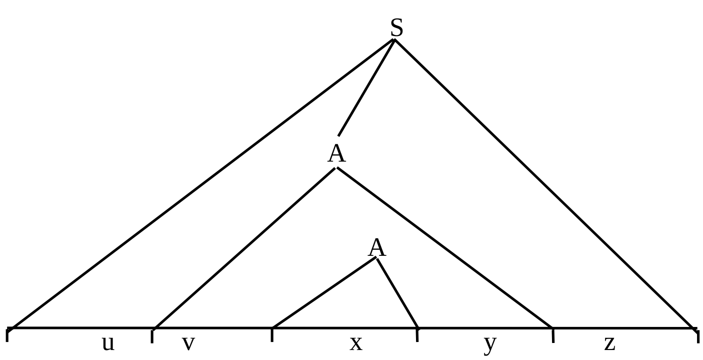
$$\underline{L_3} = \underline{L_1} \cup L_2.$$

Example:

$$a^nb^nc^* = a^nb^nc^* \cup a^nb^nc^n$$

The Context-Free Pumping Lemma

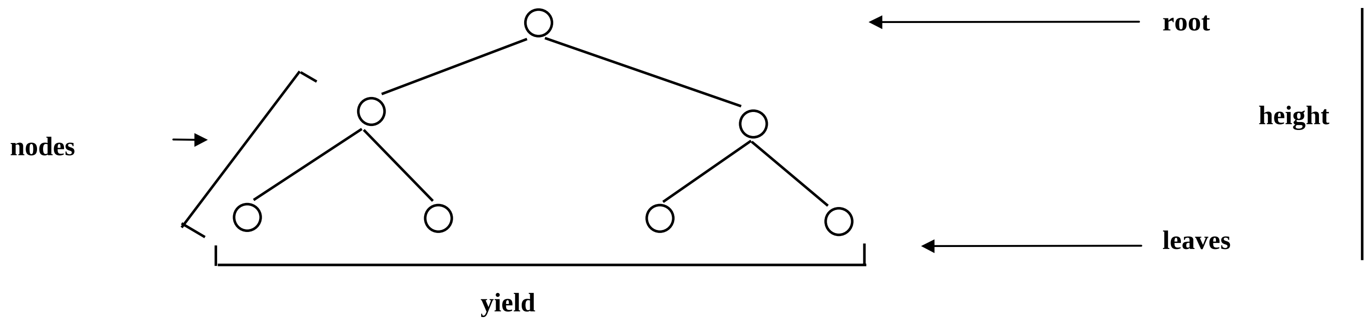
This time we use parse trees, not automata as the basis for our argument.



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 $|vxy| \leq M$, for some value of M.

uxz, uvxyz, uvvxyyz, uvvvxyyyz, etc. (i.e., uv^nxy^nz, for $n \geq 0$) are all in L.

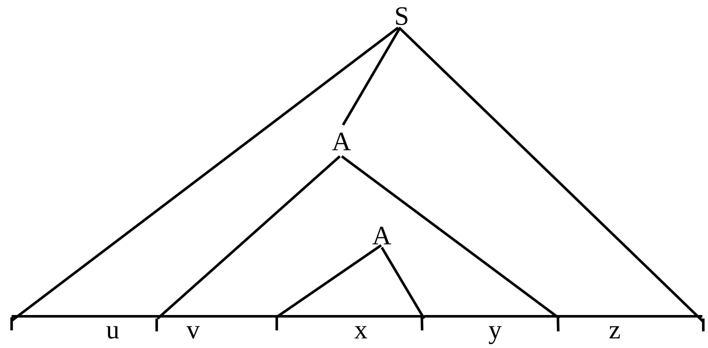
Some Tree Basics



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 $\leq$ n (so induction hypothesis holds) and yield $\leq B^n$. The number of subtrees $\leq$ B. So the yield must be $\leq B(B^n)$ or B^{n+1}.

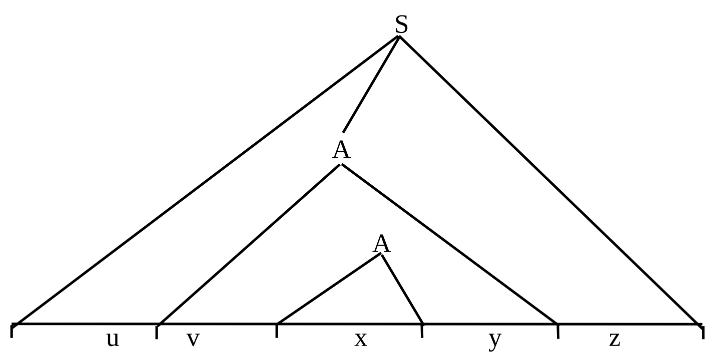
What Is K?



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:

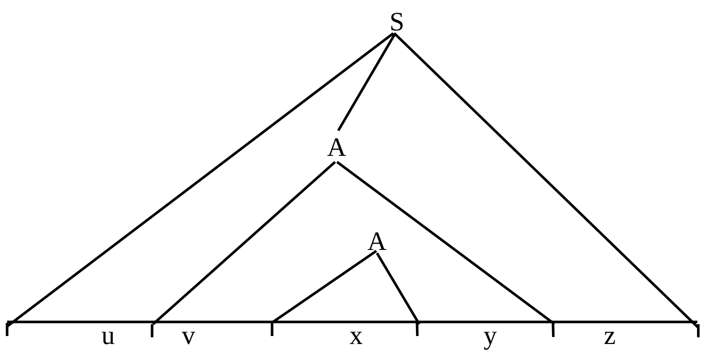
- $|vy| > 0$,
- $|vxy| \le M$ (B^{T+1}), (making this the "strong" form),
- for every $n \ge 0$, uv^nxy^nz is in L(G).

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, $|vxy| \le B^{T+1}$.

An Example of Pumping

$L = \{a^nb^nc^n : n \ge 0\}$

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 $M \ge K$) 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
        $|vxy| \le M$, and
        for all $n \ge 0$, uv^nxy^nz is in L.

We need to **<u>pick w</u>**, 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 **<u>show that all possible picks for v and y violate at least one of the criteria</u>**.

<u>Write out a single string, w</u> (in terms of K or M) **<u>Divide w into regions</u>**.

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^nxy^nz 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 ≥ 1 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 = aaaaaaaaaaaaaaaaaaaaaaaaa

u $\overline{\text{v}}$ x $\overline{\text{y}}$ 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 ∈ {a, b, c}* : w 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 ∈ {a, b}* }

Let's try pumping: |w| > K

t | t

u v x y z

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:

```
        t        |        t
   u    v            x        y          z
```

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:

```
   1           2          3         4
aaaaaaaaaabbbbbbbbbbaaaaaaaaaabbbbbbbbbb
               t    |          t
   u           v  x  y          z
```

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:

```
   1           2          3         4
aaaaaaaaaabbbbbbbbbbaaaaaaaaaabbbbbbbbbb
               t    |          t
   u           v  x  y          z
```

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

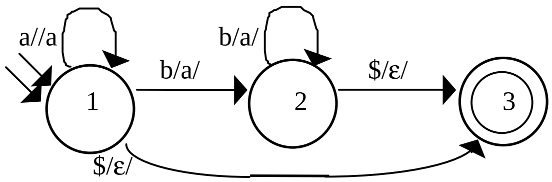
Proof:

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:

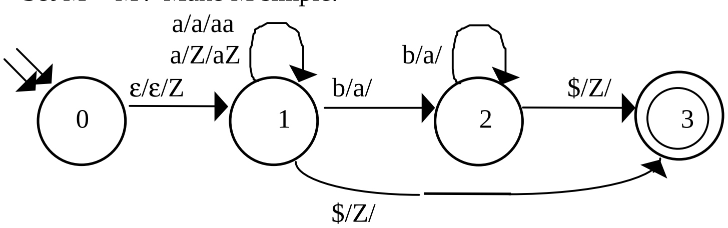
1. Initially, let M' = M.
2. M' is already deterministic.
3. Make M' simple. Why?
4. Complete M' by adding a dead state, if necessary, and adding all required transitions into it, including:
   - Transitions that are required to assure that for all input, stack combinations some transition can be followed.
   - 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

$L = a^nb^n$ M accepts L$ (and is deterministic):

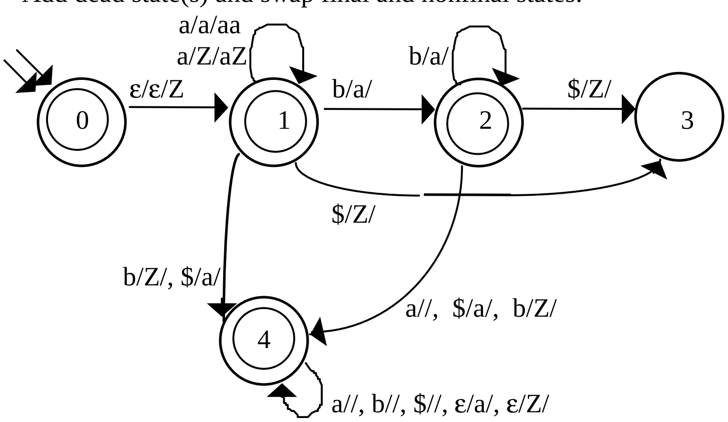


Set M = M'. Make M simple.



The Construction, Continued

Add dead state(s) and swap final and nonfinal states:



Issues: 1) Never having the machine die
2) ¬(L$) ≠ (¬L)$
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 $L = \{a^nb^mc^p : m \neq n \text{ or } m \neq p\}$ 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

$$L_2 = L_1 \cap a^*b^*c^* \text{ (a regular language)}$$

would be context free. But

$$L_2 = \{a^nb^nc^n : n \geq 0\}, \text{ which we know is not context free.}$$

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