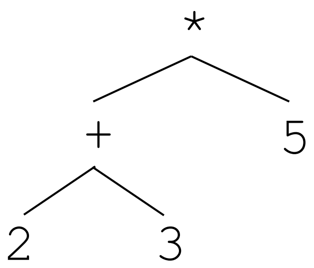
Context-free languages

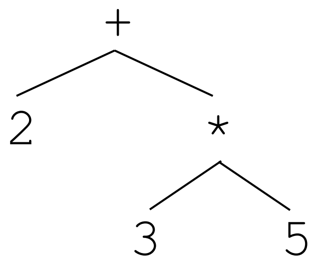
Precedence in arithmetic expressions

```
bash-3.2$ python
>>> 2+3*5
17
```



= 25

or

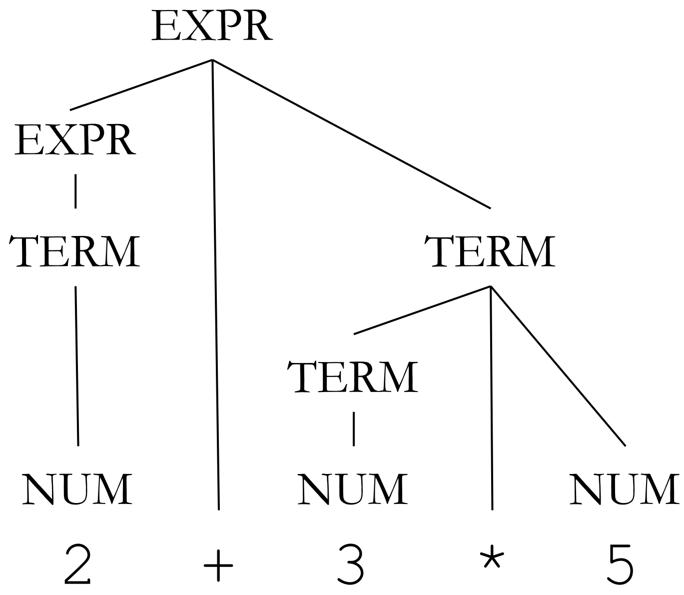


= 17

Grammars describe meaning

EXPR → EXPR + TERM
EXPR → TERM
TERM → TERM * NUM
TERM → NUM
NUM → 0-9

rules for valid (simple) arithmetic expressions



These rules always yield the correct meaning

The grammar of English

SENTENCE → NOUN-PHRASE VERB-PHRASE

$\underbrace{\texttt{a girl}}_{\text{NOUN-PHRASE}}\ \underbrace{\texttt{likes the cat}}_{\text{VERB-PHRASE}}$

NOUN-PHRASE → A-NOUN

or → A-NOUN PREP-PHRASE

$\underbrace{\texttt{a girl}}_{\text{A-NOUN}}$

$\underbrace{\texttt{a girl}}_{\text{A-NOUN}}\ \underbrace{\texttt{with a flower}}_{\text{PREP-PHRASE}}$

The grammar of English

NOUN-PHRASE → A-NOUN

or → A-NOUN PREP-PHRASE

$\underbrace{\texttt{a girl}}_{\text{A-NOUN}}$ $\underbrace{\texttt{a girl}}_{\text{A-NOUN}}\ \underbrace{\texttt{with a flower}}_{\text{PREP-PHRASE}}$

recursive structure

PREP-PHRASE → PREP NOUN-PHRASE

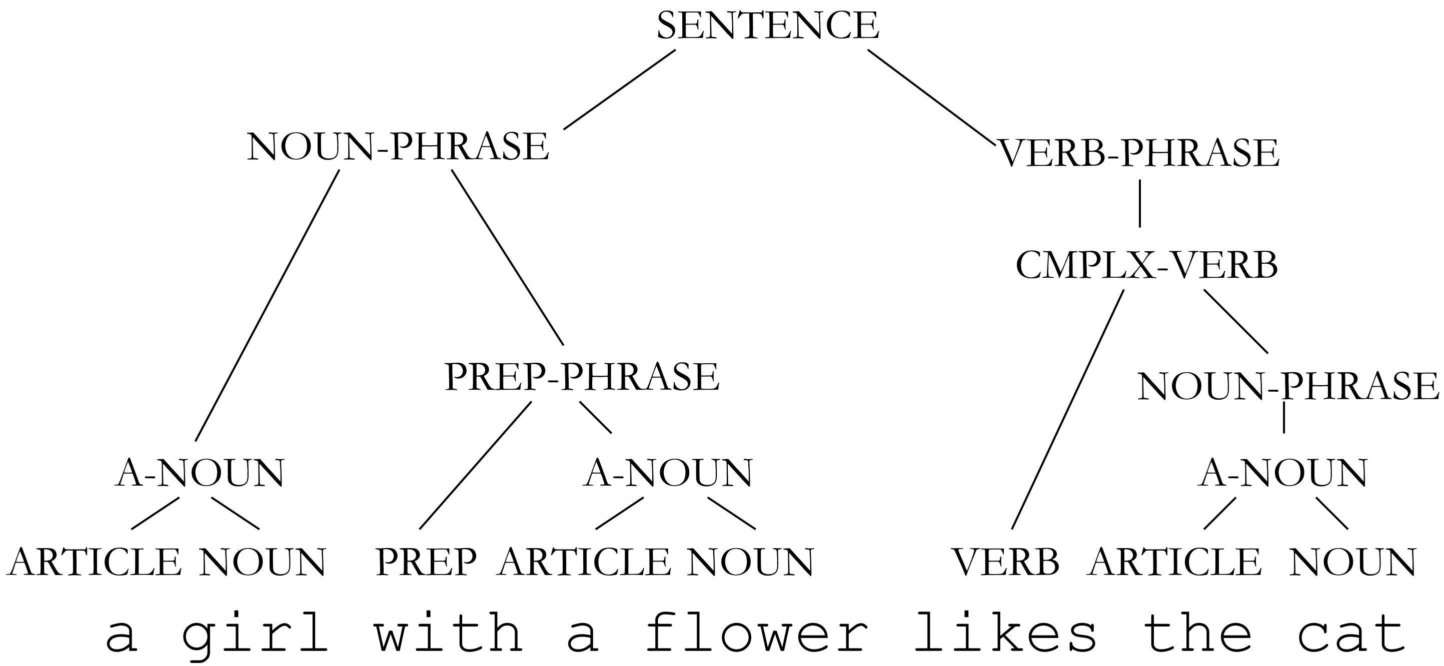
$\underbrace{\texttt{with}}_{\text{PREP}}\ \underbrace{\texttt{a flower}}_{\text{NOUN-PHRASE}}$

The grammar of (parts of) English

SENTENCE → NOUN-PHRASE VERB-PHRASE
NOUN-PHRASE → A-NOUN
NOUN-PHRASE → A-NOUN PREP-PHRASE
VERB-PHRASE → CMPLX-VERB
VERB-PHRASE → CMPLX-VERB PREP-PHRASE
PREP-PHRASE → PREP A-NOUN
A-NOUN → ARTICLE NOUN
CMPLX-VERB → VERB NOUN-PHRASE
CMPLX-VERB → VERB

ARTICLE → `a`
ARTICLE → `the`
NOUN → `cat`
NOUN → `girl`
NOUN → `flower`
VERB → `likes`
VERB → `touches`
VERB → `sees`
PREP → `with`

The meaning of sentences



Context-free grammar

start variable

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

variables

terminals

productions

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111$$
$$\Rightarrow 000B111 \Rightarrow 000\#111$$

derivation

Context-free grammar

- A context-free grammar is given by (V, Σ, R, S) where
  - V is a finite set of variables or non-terminals
  - Σ is a finite set of terminals
  - R is a set of productions or substitution rules of the form

    $$\mathrm{A} \to \alpha$$

    A is a variable and α is a string of variables and terminals
  - S is a variable called the start variable

Notation and conventions

$E \rightarrow E + E$
$E \rightarrow (E)$
$E \rightarrow N$

$N \rightarrow 0N$
$N \rightarrow 1N$
$N \rightarrow 0$
$N \rightarrow 1$

Variables: E, N
Terminals: +, (,), 0, 1
Start variable: E

shorthand:

$E \rightarrow E + E \mid (E) \mid N$
$N \rightarrow 0N \mid 1N \mid 0 \mid 1$

conventions:

Variables in UPPERCASE

Start variable comes first (typically denoted by S)

Derivation

- A derivation is a sequential application of productions:

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow (E) + E \\
&\Rightarrow (E) + N \\
&\Rightarrow (E + E) + 1 \\
&\Rightarrow (E + E) + 1 \\
&\Rightarrow (E + N) + 1 \\
&\Rightarrow (N + N) + 1 \\
&\Rightarrow (N + 1N) + 1 \\
&\Rightarrow (N + 10) + 1 \\
&\Rightarrow (1 + 10) + 1
\end{aligned}
$$

derivation

$$E \stackrel{*}{\Rightarrow} (1 + 10) + 1$$

$$
\boxed{\begin{aligned}
&E \rightarrow E + E \mid (E) \mid N \\
&N \rightarrow 0N \mid 1N \mid 0 \mid 1
\end{aligned}}
$$

$\alpha \Rightarrow \beta$ one production

$\alpha \stackrel{*}{\Rightarrow} \beta$ derivation

(Two derivations are different if they use two different rules at any point.)

Context-free languages

- The language *generated* by a CFG G is the set of all strings at the end of a derivation

$$L(G) = \{w : w \in \Sigma^* \text{ and } S \overset{*}{\Rightarrow} w\}$$

- Questions we will ask:

Given a CFG, what is its language?

Given a language, write a CFG for it.

Analysis example 1

$A \rightarrow 0A1 \mid B$
$B \rightarrow \#$

$$\boxed{L(G) = \{0^n\#1^n : n \geq 0\}}$$

- Can you derive:

00#11 $\quad A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00\#11$

$\quad A \Rightarrow B \Rightarrow \#$

00#111 $\quad$ No, there is an uneven number of 0s and 1s

00##11 $\quad$ No, there are too many #

Analysis example 2

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

- Can you derive

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (\,) \end{aligned}$$

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow (\,(S)\,S) \\ &\Rightarrow (\,(S)\,(S)\,) \\ &\Rightarrow (\,(\,)\,(S)\,) \\ &\Rightarrow (\,(\,)\,(\,)\,) \end{aligned}$$

$(\,)$

$(\,(\,)\,(\,)\,)$

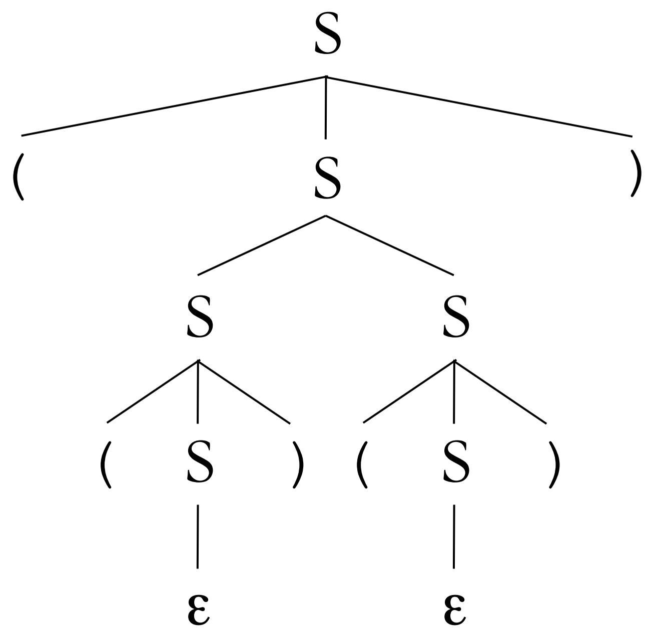
Parse trees

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

- A parse tree gives a more compact representation:

$$
\begin{aligned}
S &\Rightarrow (S) \\
&\Rightarrow (SS) \\
&\Rightarrow (\,(S)\,S) \\
&\Rightarrow (\,(S)\;(S)\,) \\
&\Rightarrow (\,(\,)\;(S)\,) \\
&\Rightarrow (\,(\,)\;(\,)\,)
\end{aligned}
$$

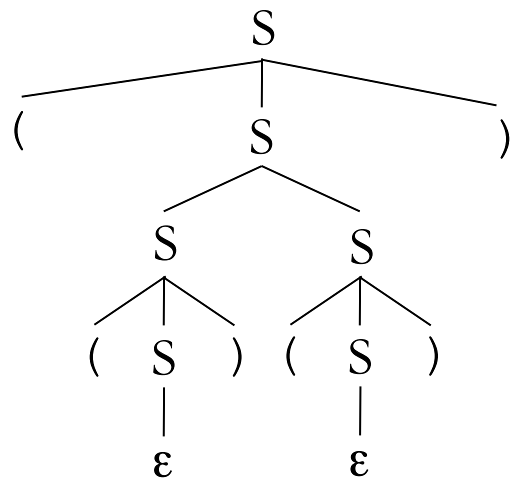
$(\,(\,)\;(\,)\,)$



Parse trees

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow ((S)S) \\ &\Rightarrow ((S)\ (S)) \\ &\Rightarrow (()\ (S)) \\ &\Rightarrow (()\ ()) \end{aligned}$$

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow ((S)S) \\ &\Rightarrow (()S) \\ &\Rightarrow (()\ (S)) \\ &\Rightarrow (()\ ()) \end{aligned}$$



$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow (S(S)) \\ &\Rightarrow ((S)\ (S)) \\ &\Rightarrow (()\ (S)) \\ &\Rightarrow (()\ ()) \end{aligned}$$

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow (S(S)) \\ &\Rightarrow (S()) \\ &\Rightarrow ((S)\ ()) \\ &\Rightarrow (()\ ()) \end{aligned}$$

One parse tree can represent many derivations

Analysis example 2

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

- Can you derive

(( ) ( ) No, because there is an uneven number of (and)

$\underline{(\,)\,)}\,(\,(\,)$ No, because there is a prefix with an excess of)

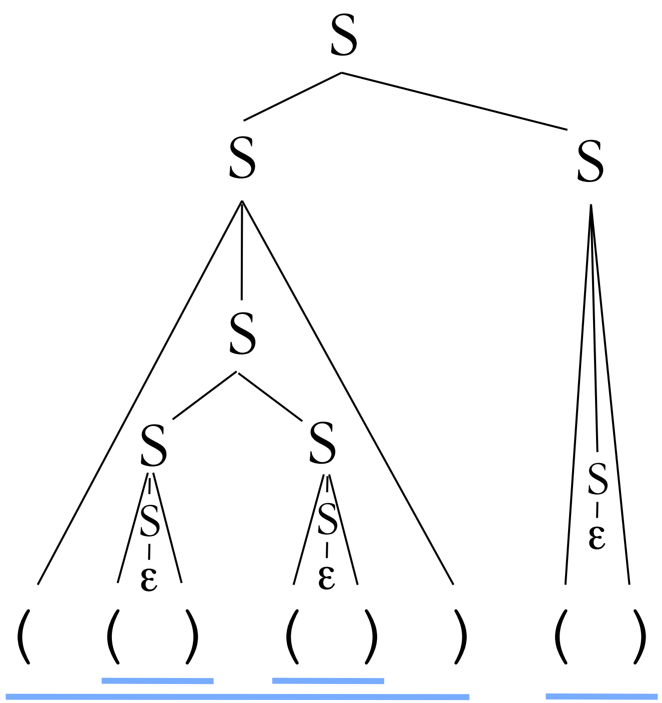
Analysis example 2

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

$L(G) = \{w:$

w has the same number of (and)

no prefix of w has more) than ($\}$



Parsing rules:

Divide w up in blocks with same number of (and)

Each block is in $L(G)$

Parse each block recursively

Design example 1

$L_1 = \{0^n 1^n \mid n \geq 0\}$

These strings have recursive structure:

$$
\begin{array}{c}
000000111111 \\
0000011111 \\
00001111 \\
000111 \\
0011 \\
01 \\
\varepsilon
\end{array}
$$

$$\boxed{S \rightarrow 0S1 \mid \varepsilon}$$

Design example 2

$L_2 =$ all natural numbers in decimal notation

0, 109, 2, 23

allowed

ε, 01, 003

not allowed

$$\boxed{\begin{aligned} S &\rightarrow 0 \mid LN \\ N &\rightarrow DN \mid \varepsilon \\ D &\rightarrow 0 \mid L \\ L &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}}$$

$$\underset{\text{leading digit } \mathrm{L}}{1}\underbrace{052870032}_{\text{any number } \mathrm{N}}$$

Design examples

$L_3 = \{0^n1^n0^m1^m \mid n \geq 0, m \geq 0\}$

010011
00110011
000111

These strings have two parts:

$$L_3 = L_pL_s$$
$$L_p = \{0^n1^n \mid n \geq 0\}$$
$$L_s = \{0^m1^m \mid m \geq 0\}$$

rules for L_p: $S_1 \rightarrow 0S_11 \mid \varepsilon$

L_s is the same as L_p

$$\boxed{\begin{aligned} &S \rightarrow S_1S_1 \\ &S_1 \rightarrow 0S_11 \mid \varepsilon \end{aligned}}$$

Design examples

$L_4 = \{0^n 1^m 0^m 1^n \mid n \geq 0, m \geq 0\}$

011001
0011
1100
00110011

These strings have nested structure:

outer part: $0^n 1^n$
inner part: $1^m 0^m$

$$\boxed{\begin{aligned} S &\rightarrow 0S1 \mid I \\ I &\rightarrow 1I0 \mid \varepsilon \end{aligned}}$$

Design examples

$L_5 = \{x\text{: } x$ has two 0-blocks with same number of 0s$\}$

01011, 001011001, 1001010101001   01001000, 01111

allowed   not allowed

$$\underbrace{1001}_{\substack{\text{initial part}\\ A}}\underbrace{0011010010}_{\substack{\text{middle part}\\ B}}\underbrace{10110}_{\substack{\text{final part}\\ C}}$$

A: ε, or ends in 1

C: ε, or begins with 1

Design examples

$$\underbrace{1001}_{A}\underbrace{00110100}_{B}\underbrace{10110}_{C}$$

A: ε, or ends in 1

C: ε, or begins with 1

U: any string

$$
\begin{aligned}
S &\rightarrow ABC \\
A &\rightarrow \varepsilon \mid U1 \\
U &\rightarrow 0U \mid 1U \mid \varepsilon \\
C &\rightarrow \varepsilon \mid 1U \\
B &\rightarrow 0D0 \mid 0B0 \\
D &\rightarrow 1U1 \mid 1
\end{aligned}
$$

B has recursive structure:

$$00\underbrace{\qquad\qquad}_{D}00$$

same number of 0s
at least one 0

D: begins and ends in 1

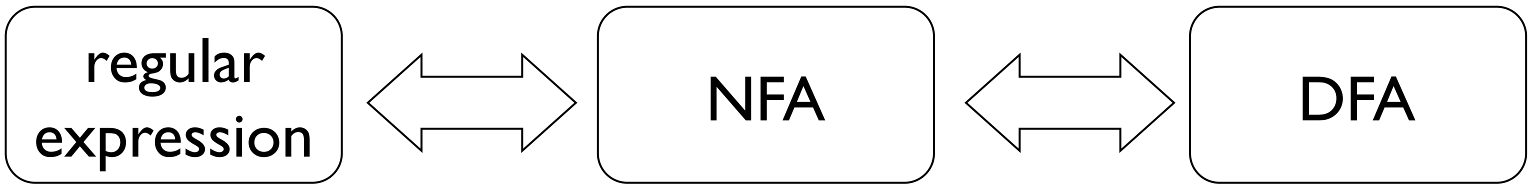
Context-free versus regular

- Write a CFG for the language $(0 + 1)^*111$

$$S \rightarrow U111$$
$$U \rightarrow 0U \mid 1U \mid \varepsilon$$

- Can you do so for every regular language?

Every regular language is context-free



From regular to context-free

regular expression	$\Rightarrow$	CFG
$\varnothing$		grammar with no rules
ε		$S \to \varepsilon$
a (alphabet symbol)		$S \to a$
$E_1 + E_2$		$S \to S_1 \mid S_2$
$E_1 E_2$		$S \to S_1 S_2$
$E_1{}^*$		$S \to S S_1 \mid \varepsilon$

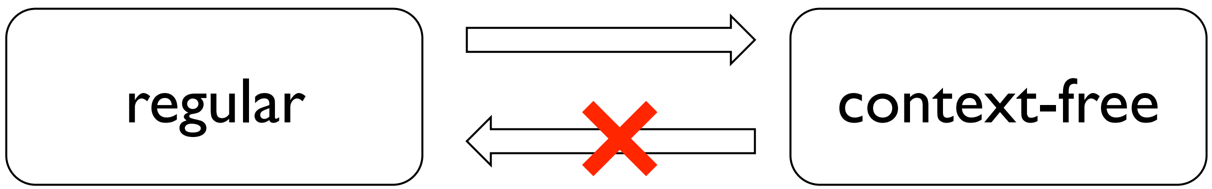
(S becomes the new start symbol)

Context-free versus regular

- Is every context-free language regular?

$$S \rightarrow 0S1 \mid \varepsilon \qquad\qquad L_1 = \{0^n 1^n \colon n \geq 0\}$$

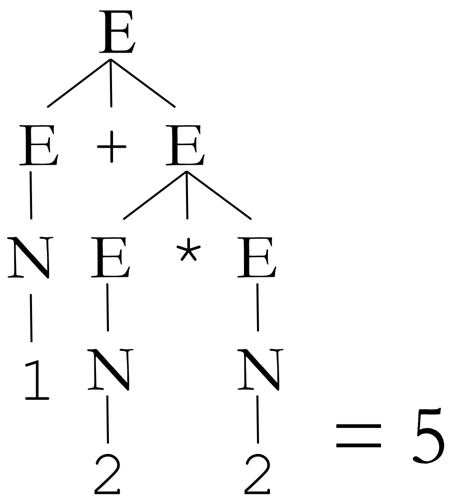
Is context-free but not regular



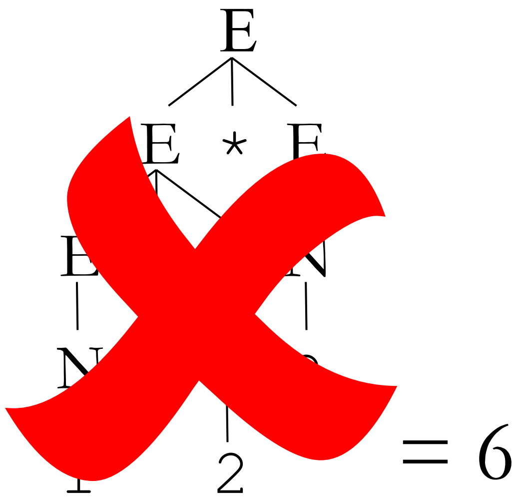
Ambiguity
Parsing algorithms
Probabilistic Context-Free Grammars

Ambiguity

$$E \rightarrow E + E \mid E * E \mid (E) \mid N$$
$$N \rightarrow 1N \mid 2N \mid 1 \mid 2$$



1+2*2

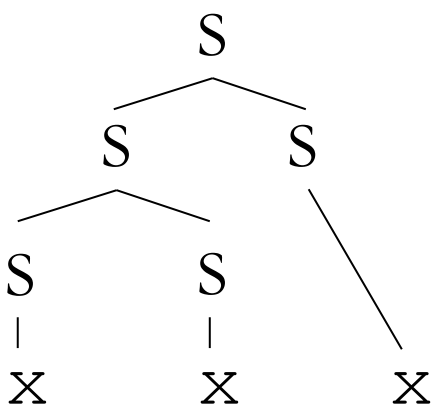


A CFG is ambiguous if some string has more than one parse tree

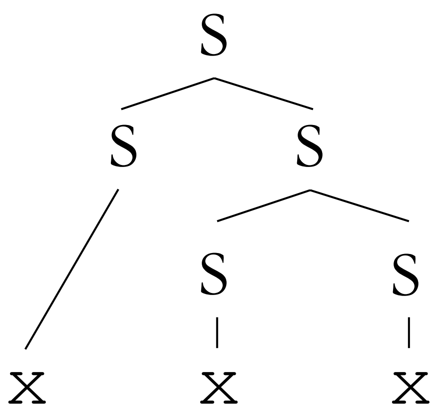
Example

Is $S \rightarrow SS \mid x$ ambiguous?

Yes, because

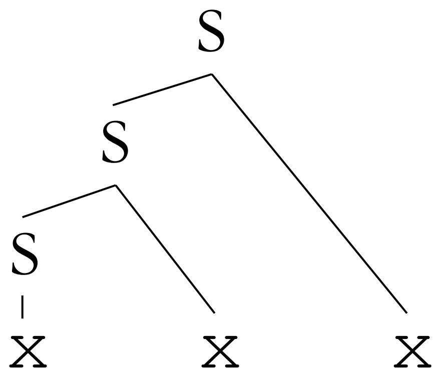


xxx



Disambiguation

$S \to SS \mid x \quad \Rightarrow \quad S \to Sx \mid x$



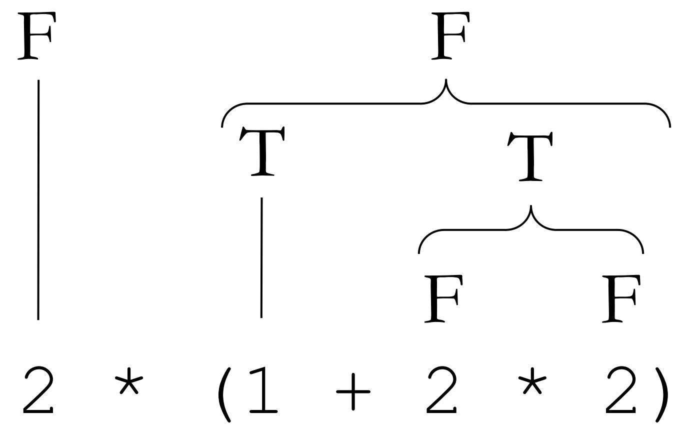
Sometimes we can rewrite the grammar to remove ambiguity

Disambiguation

$$\begin{array}{|l|}\hline E \rightarrow E + E \mid E * E \mid (E) \mid N \\ N \rightarrow 1N \mid 2N \mid 1 \mid 2 \\ \hline \end{array}$$

same precedence!

Divide expression
into terms and factors



Disambiguation

$$E \rightarrow E + E \mid E * E \mid (E) \mid N$$
$$N \rightarrow 1N \mid 2N \mid 1 \mid 2$$

An expression is a sum of one or more terms

Each term is a product of one or more factors

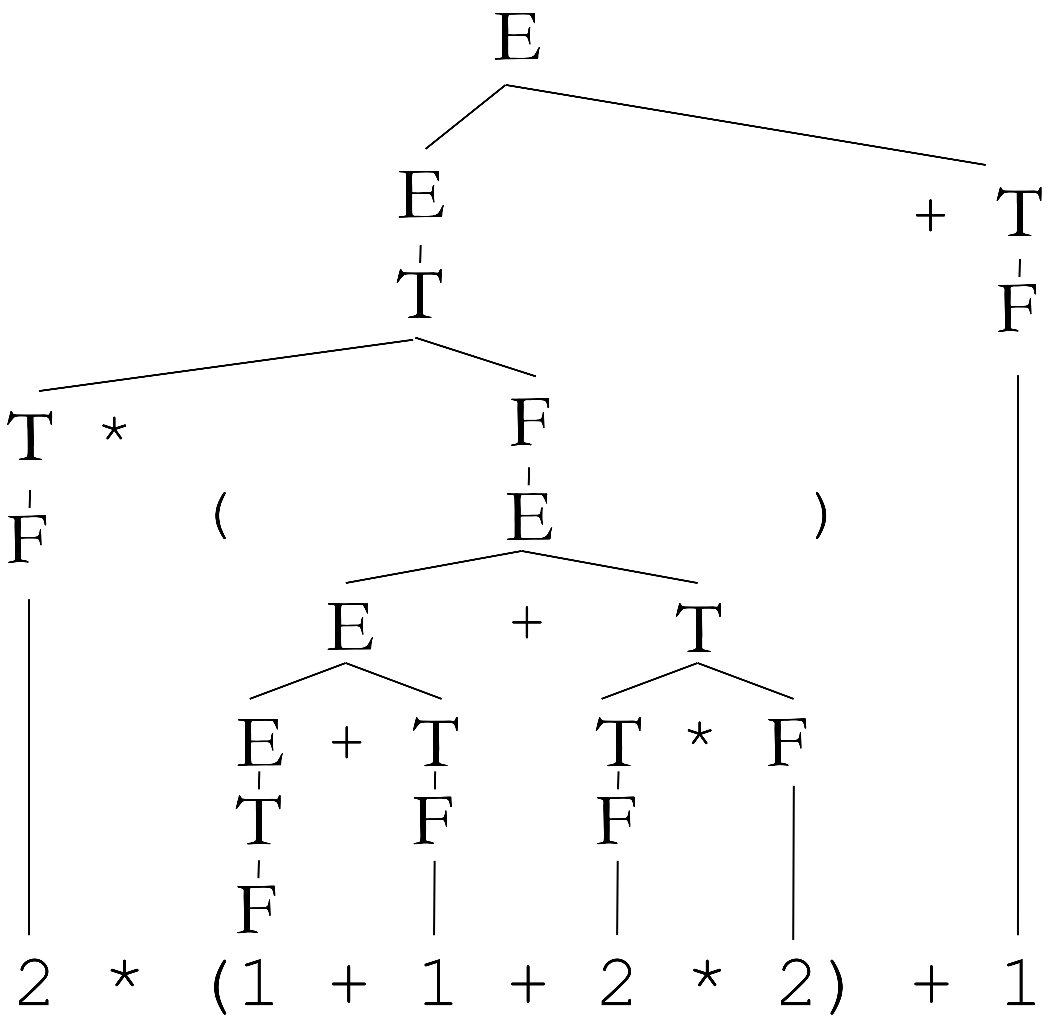
Each factor is a parenthesized expression or a number

$$E \rightarrow T \mid E + T$$

$$T \rightarrow F \mid T * F$$

$$F \rightarrow (E) \mid 1 \mid 2$$

Parsing example



$E \rightarrow T \mid E + T$
$T \rightarrow F \mid T * F$
$F \rightarrow (E) \mid 1 \mid 2$

Unique parse tree
(because +/* is
right-binding

Disambiguation

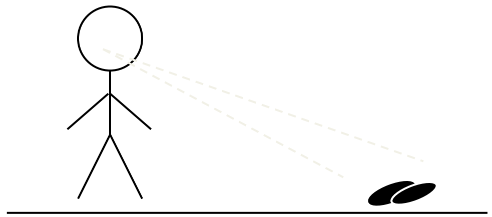
- Disambiguation is not always possible because
  - There exist inherently ambiguous languages
  - There is no general procedure for disambiguation

- In programming languages, ambiguity comes from precedence rules, and we can deal with it like in the previous example

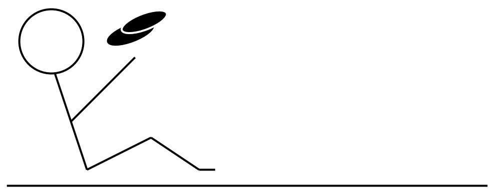
- In English, ambiguity is sometimes a problem:

```
He ate the cookies on the floor.
```

Ambiguity in English



He ate the cookies on the floor.



Parsing

$S \rightarrow \mathtt{0S1} \mid \mathtt{1S0S} \mid T$
$T \rightarrow S \mid \varepsilon$

input: `0011`

How would we program the computer to build a parse tree for us?

Parsing

$S \to 0S1 \mid 1S0S \mid T$
$T \to S \mid \varepsilon$

input: 0011

$$
\begin{array}{ccccccccc}
S & \Rightarrow & 0S1 & \Rightarrow & 00S11 & \Rightarrow & 000S111 & \Rightarrow & \cdots \\
 & & & & 01S0S1 & \Rightarrow & \vdots & & \\
 & & & & 0T1 & \Rightarrow & 00T11 & \Rightarrow & 00S11 \\
 & \Rightarrow & 1S0S & \Rightarrow & 10S10S & \Rightarrow & & & 0011 \checkmark \\
 & & & & \vdots & \vdots & & & \\
 & \Rightarrow & T & \Rightarrow & S & \Rightarrow & & & \\
 & & & & \varepsilon & & & &
\end{array}
$$

First idea: Try all derivations

Problems

① Trying all derivations may take a very long time

② If input is not in the language, parsing will never stop

When to stop

$S \rightarrow 0S1 \mid 1S0S \mid T$
$T \rightarrow S \mid \varepsilon$

Idea 2: Stop when

$|\text{derived string}| > |\text{input}|$

Problems

$$\underset{1}{S} \Rightarrow \underset{3}{0S1} \Rightarrow \underset{3}{0T1} \Rightarrow \underset{2}{01}$$

Derived strings may shrink
because of "ε-productions"

$$S \Rightarrow T \Rightarrow S \Rightarrow T \Rightarrow \dots$$

Derivation may loop
because of "unit productions"

Task: remove ε− and unit- productions

Removal of ε-productions

- A variable N is nullable if it can derive the empty string

$$\mathrm{N} \overset{*}{\Rightarrow} \varepsilon$$

① Identify all nullable variables N

② Remove all ε-productions carefully (while adding extra productions to compensate for this)

③ If start variable S is nullable:
Add a new start variable S'
Add special productions $\mathrm{S}' \rightarrow \mathrm{S} \mid \varepsilon$

Example

grammar

$S \rightarrow ACD$
$A \rightarrow a$
$B \rightarrow \varepsilon$
$C \rightarrow ED \mid \varepsilon$
$D \rightarrow BC \mid b$
$E \rightarrow b$

nullable variables

B C D

Repeat the following:

If $X \rightarrow \varepsilon$, mark X as nullable

If $X \rightarrow YZ\ldots W$, all marked nullable, mark X as nullable also.

① Identify all nullable variables

Eliminating ε-productions

$S \rightarrow ACD$
$A \rightarrow a$
$\cancel{B \rightarrow \varepsilon}$
$C \rightarrow ED \mid \cancel{\varepsilon}$
$D \rightarrow BC \mid b$
$E \rightarrow b$

nullable: B, C, D

$D \rightarrow C$
$S \rightarrow AD$
$D \rightarrow B$
$\cancel{D \rightarrow \varepsilon}$
$S \rightarrow AC$
$S \rightarrow A$
$C \rightarrow E$

For every nullable N:

If you see $\mathrm{X} \rightarrow \alpha\mathrm{N}\beta$, add $\mathrm{X} \rightarrow \alpha\beta$

If you see $\mathrm{N} \rightarrow \varepsilon$, remove it.

② Remove all ε-productions carefully

The end result

The old grammar

$S \rightarrow ACD$
$A \rightarrow a$
$B \rightarrow \varepsilon$
$C \rightarrow ED \mid \varepsilon$
$D \rightarrow BC \mid b$
$E \rightarrow b$

The new grammar

$S \rightarrow ACD$
$A \rightarrow a$
$C \rightarrow ED$
$D \rightarrow BC \mid b$
$E \rightarrow b$
$D \rightarrow C$
$S \rightarrow AD$
$D \rightarrow B$
$S \rightarrow AC$
$S \rightarrow A$
$C \rightarrow E$

② Remove all ε-productions carefully

Eliminating unit productions

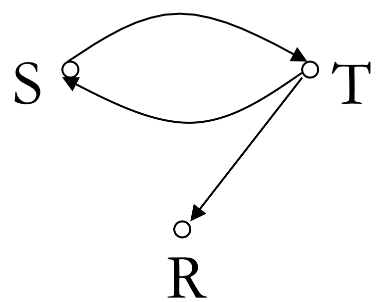
- A unit production is a production of the form

$$A \rightarrow B$$

grammar:

$$S \rightarrow \mathtt{0}S\mathtt{1} \mid \mathtt{1}S\mathtt{0}S \mid T$$
$$T \rightarrow S \mid R \mid \varepsilon$$
$$R \rightarrow \mathtt{0}SR$$

unit productions graph:



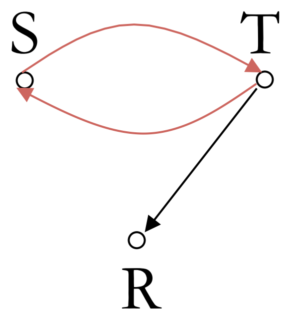
Removal of unit productions

① If there is a cycle of unit productions

$$A \rightarrow B \rightarrow \dots \rightarrow C \rightarrow A$$

delete it and replace everything with A

$$\begin{aligned} S &\rightarrow 0S1 \mid 1S0S \mid \cancel{T} \\ T &\rightarrow \cancel{S} \mid R \mid \varepsilon \\ R &\rightarrow 0SR \end{aligned}$$



$$\begin{aligned} S &\rightarrow 0S1 \mid 1S0S \\ S &\rightarrow R \mid \varepsilon \\ R &\rightarrow 0SR \end{aligned}$$

replace T by S

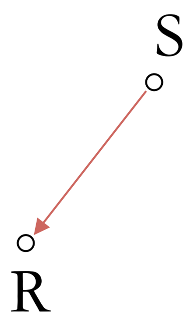
Removal of unit productions

② Replace every chain

$$A \to B \to \ldots \to C \to \alpha$$

by $A \to \alpha$, $B \to \alpha, \ldots, C \to \alpha$

$S \to 0S1 \mid 1S0S$
$\quad \mid R \mid \varepsilon$
$R \to 0SR$



$S \to 0S1 \mid 1S0S$
$\quad \mid 0SR \mid \varepsilon$
$R \to 0SR$

$S \to R \to 0SR$ is replaced by $S \to 0SR$, $R \to 0SR$

Recap

Problem: If input is not in the language, parsing will never stop

Solution:

important to do in this order

Eliminate ε productions
Eliminate unit productions

Try all possible derivations but stop parsing when

$$|\text{derived string}| > |\text{input}|$$

Example

$$S \to 0S1 \mid 0S0S \mid T$$
$$T \to S \mid 0$$

$$\Longrightarrow$$

$$S \to 0S1 \mid 0S0S \mid 0$$

input: 0011

conclusion: $0011 \notin L$

$S \Rightarrow 0$ ✘

$\Rightarrow 0S1 \Rightarrow 001$ ✘

$00S11$ too long

$00S0S1$ too long

$\Rightarrow 0S0S \Rightarrow 000S \Rightarrow 0000$ ✘

$00S10S$ too long

$000S1$ too long

$00S0S0S$ too long

$0000S0S$ too long

Problems

① Trying all derivations may take a very long time

② If input is not in the language, parsing will never stop

Preparations

A faster way to parse:
Cocke-Younger-Kasami algorithm

- To use it we must prepare the CFG and convert it to Chomsky Normal Form

Step 1. Eliminate ε productions

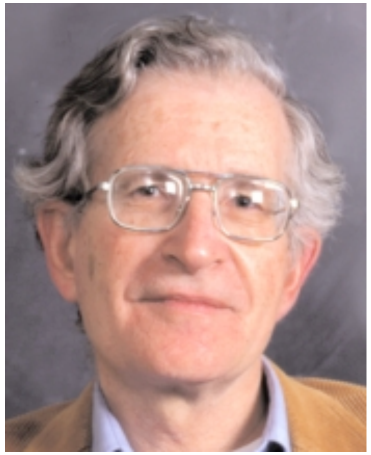
Step 2. Eliminate unit productions

Step 3. Add rules for terminals and split longer sequences of non-terminals (next slide)

Chomsky Normal Form

- A CFG is in Chomsky Normal Form if every production* has the form

$$A \to BC \qquad \text{or} \qquad A \to a$$



Noam Chomsky

3rd step of conversion to Chomsky Normal Form:

$A \to B\mathrm{c}DE$ → replace terminals with new variables → $A \to BCDE$, $C \to \mathrm{c}$ → break up sequences with new variables → $A \to BX$, $X \to CY$, $Y \to DE$, $C \to \mathrm{c}$

* Exception: We allow $S \to \varepsilon$ for the start variable only

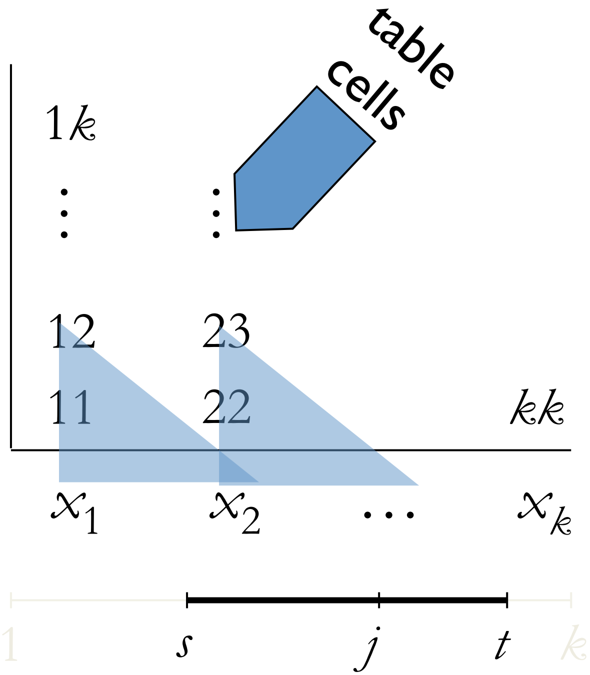
Cocke-Younger-Kasami (CYK) algorithm

Grammar without ε and unit productions
in Chomsky Normal Form

Input string

For all cells in last row
  If there is a production $\mathrm{A} \to x_i$
    Put A in table cell ii

For cells st in other rows (going bottom-up)
  If there is a production $\mathrm{A} \to \mathrm{BC}$
      where B is in cell sj and C is in cell $(j+1)t$
    Put A in cell st

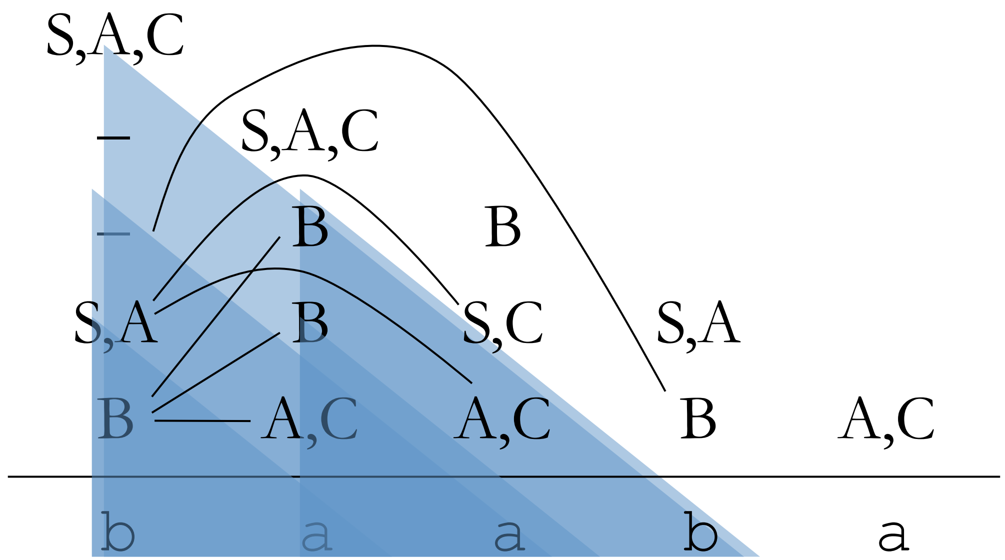


Cell st remembers all variables able to generate substring $x_s \dots x_t$

Cocke-Younger-Kasami algorithm

$S \to AB \mid BC$
$A \to BA \mid \mathtt{a}$
$B \to CC \mid \mathtt{b}$
$C \to AB \mid \mathtt{a}$

$\boxed{x = \mathtt{baaba}}$

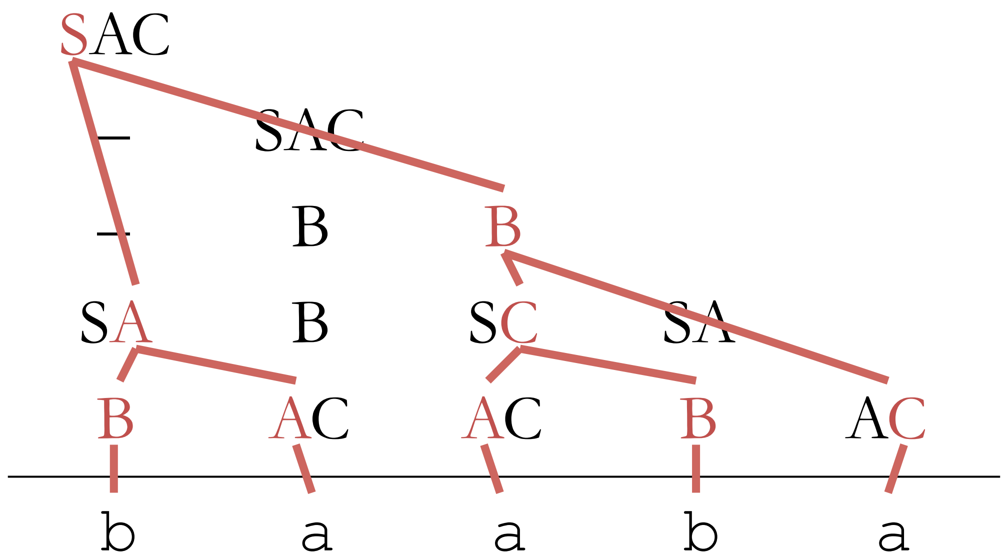


Idea: We generate each substring of x bottom up

Parse tree reconstruction

$S \to AB \mid BC$
$A \to BA \mid \texttt{a}$
$B \to CC \mid \texttt{b}$
$C \to AB \mid \texttt{a}$

$\boxed{x = \texttt{baaba}}$



Tracing back the derivations, we obtain the parse tree

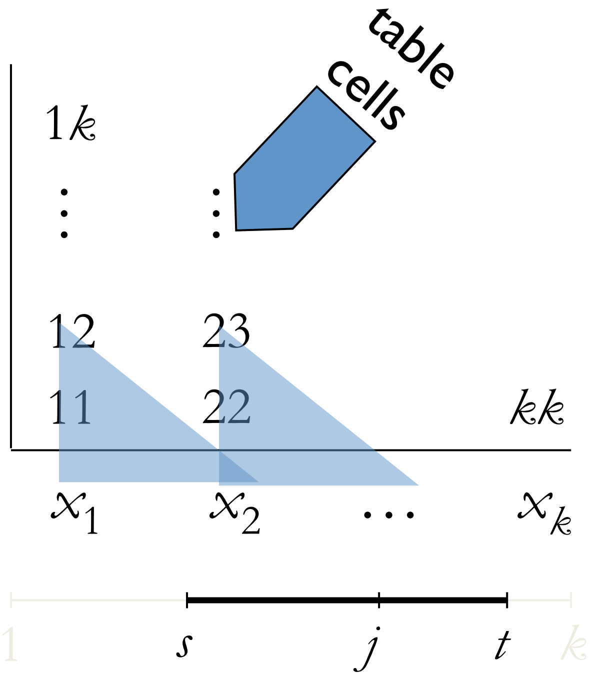
Number of different parse trees

Grammar without ε and unit productions
in Chomsky Normal Form

Input string

For all cells in last row
  If there is a production $A \to x_i$
    Put $A : 1$ in table cell ii

For cells st in other rows (going bottom-up)
  If there is a production $A \to BC$
    where $B : n_B$ is in cell sj and $C : n_C$ is in cell $(j+1)t$ and $A : n_A$ in cell st (if not present assume $n_A = 0$)
    Update $A : n_A + n_B * n_C$ in cell st

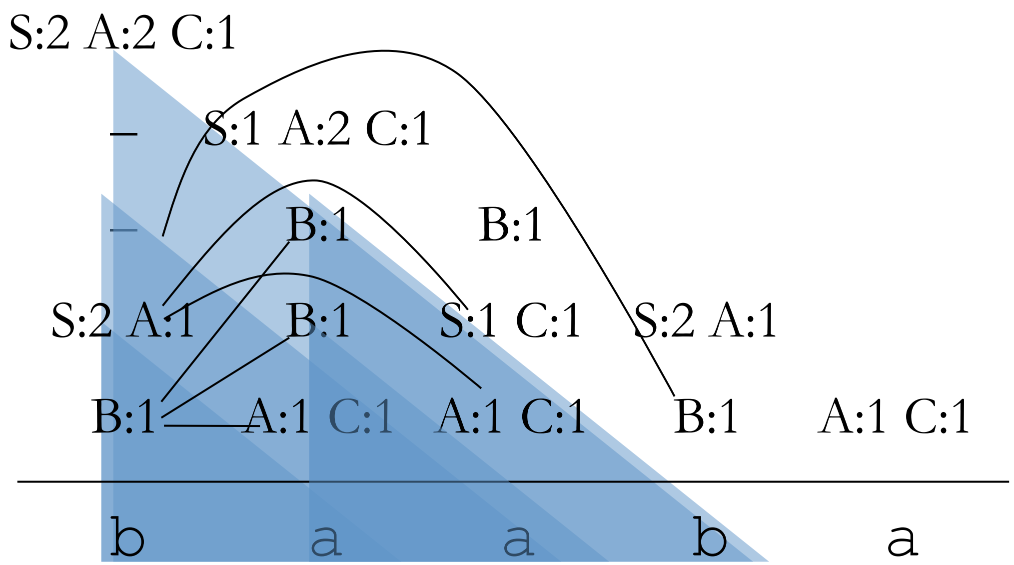


Cell st remembers for each variable the number of parse trees generating substring $x_s \ldots x_t$

Example: Number of parse trees

S → AB | BC
A → BA | a
B → CC | b
C → AB | a

$x = \texttt{baaba}$



Probabilistic Context Free Grammars (PCFG)

- A PCFG is a probabilistic version of a CFG where each production has a probability.

- Probabilities of all productions rewriting a given non-terminal must add up to 1, defining a distribution for each non-terminal.

- String generation is now probabilistic where production probabilities are used to non-deterministically select a production for rewriting a given non-terminal.

Statistical Parsing

- Statistical parsing uses a probabilistic model of syntax in order to assign probabilities to each parse tree.
- Provides a systematic approach to resolving syntactic ambiguity.

…the notion of "probability of a sentence" is an entirely useless one, under any known interpretation of this term.

— Noam Chomsky (famous linguist)

Every time I fire a linguist, the performance of the recognizer improves.

— Fred Jelinek
(former head of IBM speech recognition group)

Simple PCFG for English

Grammar	Prob	
S → NP VP	0.8	+ = 1.0
S → Aux NP VP	0.1	
S → VP	0.1	
NP → Pronoun	0.2	+ = 1.0
NP → Proper-Noun	0.2	
NP → Det Nominal	0.6	
Nominal → Noun	0.3	+ = 1.0
Nominal → Nominal Noun	0.2	
Nominal → Nominal PP	0.5	
VP → Verb	0.2	+ = 1.0
VP → Verb NP	0.5	
VP → VP PP	0.3	
PP → Prep NP	1.0	

Lexicon

Det → the | a | that | this
0.6 0.2 0.1 0.1

Noun → book | flight | meal | money
0.1 0.5 0.2 0.2

Verb → book | include | prefer
0.5 0.2 0.3

Pronoun → I | he | she | me
0.5 0.1 0.1 0.3

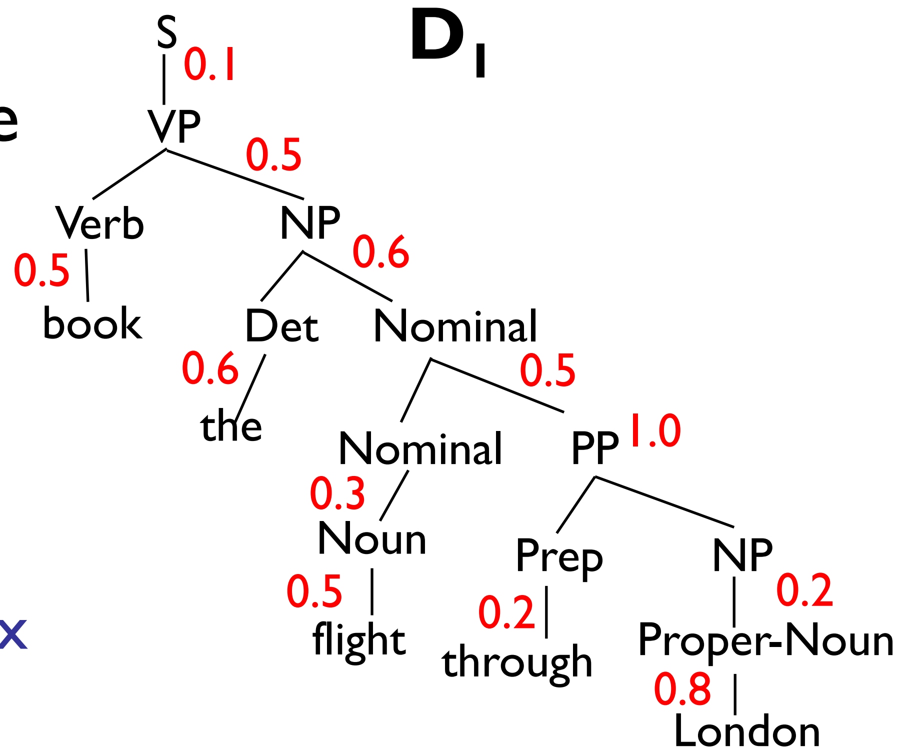
Proper-Noun → London | BA
0.8 0.2

Aux → does
1.0

Prep → from | to | on | near | through
0.25 0.25 0.1 0.2 0.2

Parse Tree Probability

- Assume productions for each node are chosen independently.
- Probability of a parse tree is the product of the probabilities of its productions.

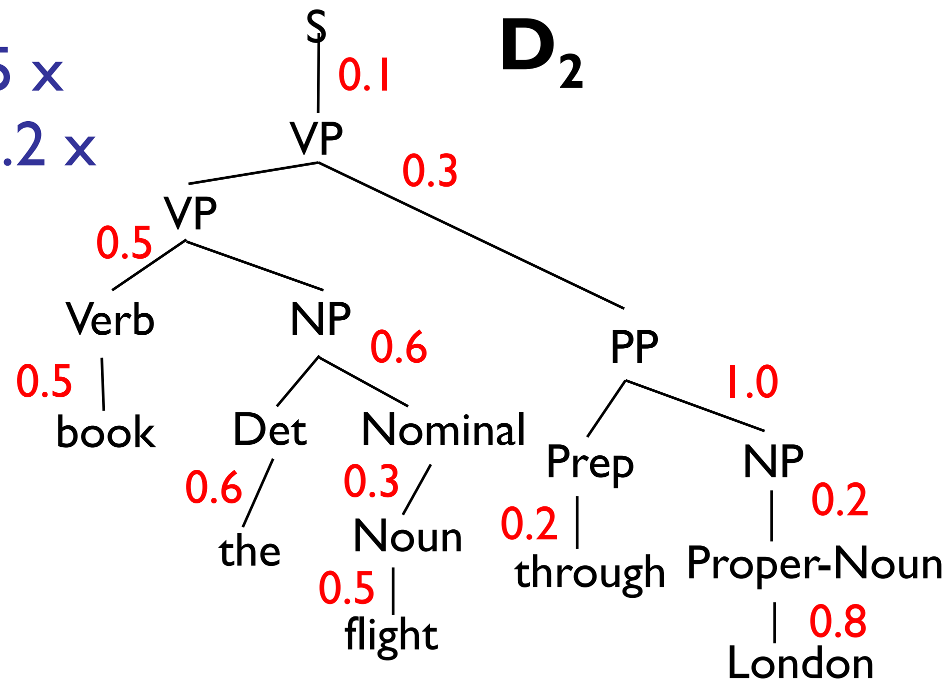


$$P(D_1) = 0.1 \times 0.5 \times 0.5 \times 0.6 \times 0.6 \times 0.5 \times 0.3 \times 1.0 \times 0.2 \times 0.2 \times 0.5 \times 0.8 = 0.0000216$$

Syntactic Disambiguation

- Resolve ambiguity by picking most probable parse tree.

$$P(D_2) = 0.1 \times 0.3 \times 0.5 \times 0.6 \times 0.5 \times 0.6 \times 0.3 \times 1.0 \times 0.5 \times 0.2 \times 0.2 \times 0.8 = 0.00001296$$



In this case D_1 is the best interpretation, because 0.0000216 > 0.00001296

Sentence Probability

- Probability of a given sentence is the sum of the probabilities of all of its parse trees.

  P("book the flight through London") =
  $P(D_1) + P(D_2) = 0.0000216 + 0.00001296$
  $= 0.00003456$

Probabilistic CYK: Most Likely Parse Tree

- CYK can be modified for PCFG parsing by including in each cell a probability for each non-terminal.

- Cell *ij* contains the most probable parse tree of each non-terminal that can generate the part of the input word from *i* through *j* together with its associated probability.

- When transforming the grammar to CNF, we set the production probabilities to preserve the probability of derivations.

Probabilistic Grammar Conversion

Original Grammar		Chomsky Normal Form	
S → NP VP	0.8	S → NP VP	0.8
S → Aux NP VP	0.1	S → X1 VP	0.1
		X1 → Aux NP	1.0
S → VP	0.1	S → book | include | prefer 0.01 0.004 0.006	
		S → Verb NP	0.05
		S → VP PP	0.03
NP → Pronoun	0.2	NP → I | he | she | me 0.1 0.02 0.02 0.06	
NP → Proper-Noun	0.2	NP → London | BA 0.16 .04	
NP → Det Nominal	0.6	NP → Det Nominal	0.6
Nominal → Noun	0.3	Nominal → book | flight | meal | money 0.03 0.15 0.06 0.06	
Nominal → Nominal Noun	0.2	Nominal → Nominal Noun	0.2
Nominal → Nominal PP	0.5	Nominal → Nominal PP	0.5
VP → Verb	0.2	VP → book | include | prefer 0.1 0.04 0.06	
VP → Verb NP	0.5	VP → Verb NP	0.5
VP → VP PP	0.3	VP → VP PP	0.3
PP → Prep NP	1.0	PP → Prep NP	1.0

Probabilistic CYK Parser: Most Likely Parse Tree

S:.03*.0135*.032 =.00001296 S:.0000216				
—	NP:.6*.6*.0024 =.000864			
S:.05*.5*.054 =.00135 VP:.5*.5*.054 =.0135	—	Nominal: .5*.15*.032 =.0024		
—	NP:.6*.6*.15 =.054	—	PP:1.0*.2*.16 =.032	
S :.01, VP:.1, Verb:.5 Nominal:.03 Noun:.1	Det:.6	Nominal:.15 Noun:.5	Prep:.2	NP:.16 PropNoun:.8
Book	**the**	**flight**	**through**	**London**

Most probable parse: use max to combine probabilities of multiple parse trees in each cell.

Probabilistic CYK Parser: Sentence Probability

S: .00001296 +.0000216 =.00003456				
—	NP:.6*.6* .0024 =.000864			
S:.05*.5*.054 =.00135 VP:.5*.5*.054 =.0135	—	Nominal: .5*.15*.032 =.0024		
—	NP:.6*.6*.15 =.054	—	PP:1.0*.2*.16 =.032	
S :.01, VP:.1, Verb:.5 Nominal:.03 Noun:.1	Det:.6	Nominal:.15 Noun:.5	Prep:.2	NP:.16 PropNoun:.8
Book	**the**	**flight**	**through**	**London**

The probability of generating a given sentence is the sum all probabilities of multiple parse trees in each cell.

More precisely

Sentence probability	**For** all cells in last row **If** there is a production $A \rightarrow x_i$ with probability p **Put** $A : p$ in table cell ii **For** cells st in other rows (going bottom-up) **If** there is a production $A \rightarrow BC$ with probability p **where** $B : p_B$ is in cell sj and $C : p_C$ is in cell $(j+1)t$ and $A : p_A$ in cell st (if not present assume $p_A = 0$) **Update** $A : p_A + p * p_B * p_C$ in cell st
Most likely parse tree	**For** all cells in last row **If** there is a production $A \rightarrow x_i$ with probability p **Put** $A : p$ in table cell ii **For** cells st in other rows (going bottom-up) **If** there is a production $A \rightarrow BC$ with probability p **where** $B : p_B$ is in cell sj and $C : p_C$ is in cell $(j+1)t$ and $A : p_A$ in cell st (if not present assume $p_A = 0$) **Update** $A : \max(p_A, p * p_B * p_C)$ in cell st

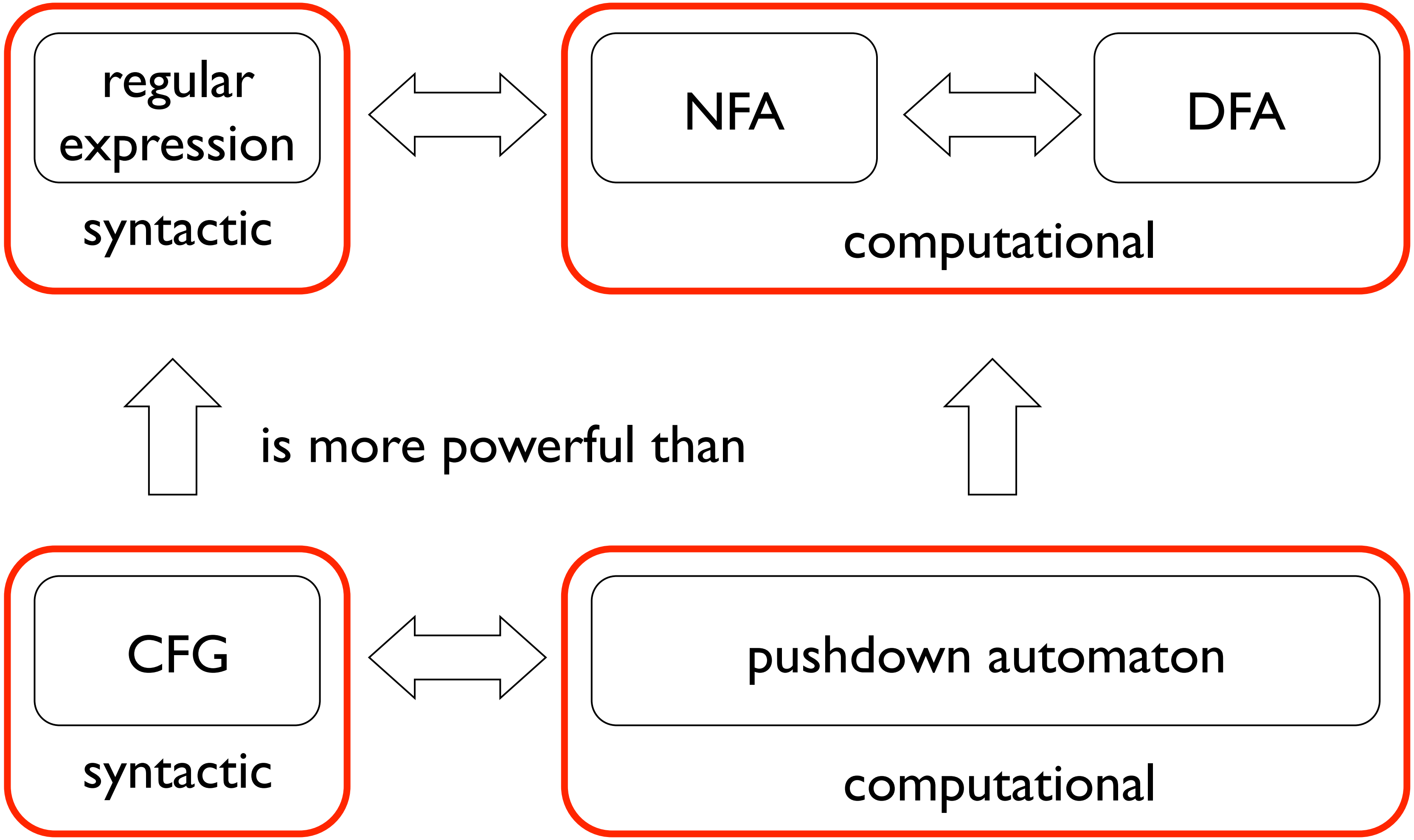
PCFGs and NLP

PCFG is consistent
if and only if
it generates a finite word with probability 1.

name	#prod	max-scc	Jacobi	Gauss Seidel	SOR ω=1.05	DNewton	SNewton
brown	22866 ✘	448	312.084(9277)	275.624(7866)	diverge	2.106(8)	2.115(9)
lemonde	32885 ✓	527	234.715(5995)	30.420(767)	diverge	1.556(7)	2.037(7)
negra	29297 ✓	518	16.995(610)	4.724(174)	4.201(152)	1.017(6)	0.499(6)
swbd	47578 ✘	1123	445.120(4778)	19.321(202)	25.654(270)	6.435(6)	3.978(6)
tiger	52184 ✓	1173	99.286(1347)	16.073(210)	12.447(166)	5.274(6)	1.871(6)
tuebadz	8932 ✓	293	6.894(465)	1.925(133)	6.878(461)	0.477(7)	0.341(7)
wsj	31170 ✓	765	462.378(9787)	68.650(1439)	diverge	2.363(7)	3.616(8)

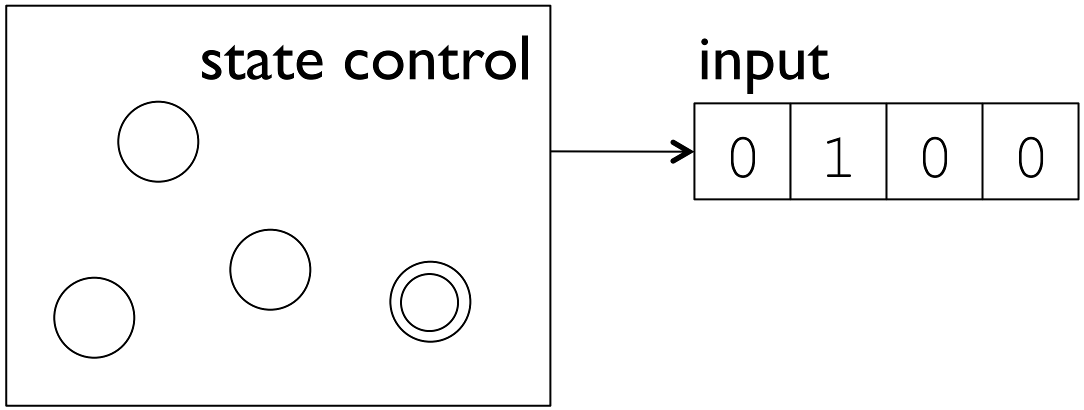
Pushdown automata

Motivation



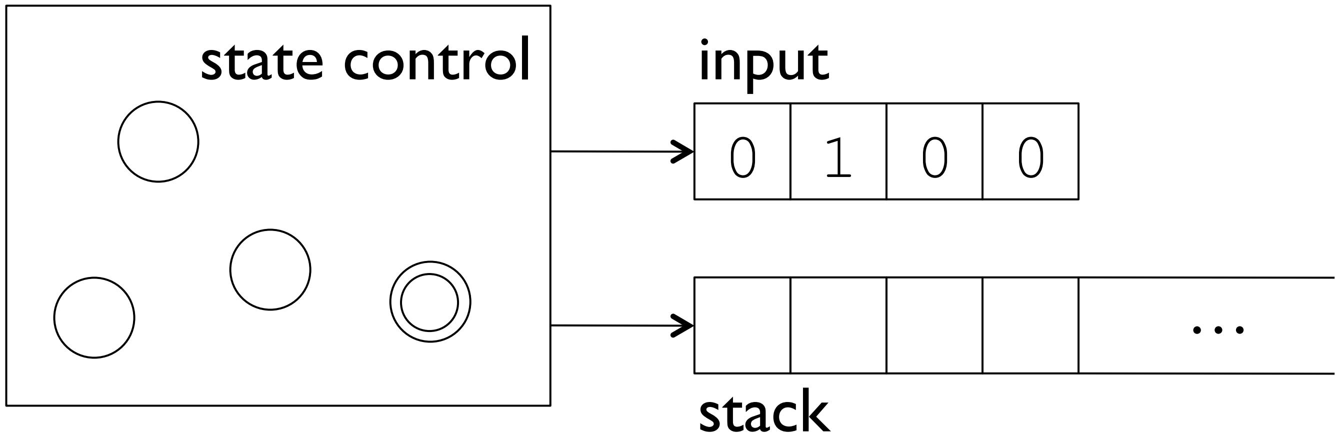
regular expression
syntactic
NFA
DFA
computational
is more powerful than
CFG
syntactic
pushdown automaton
computational

Pushdown automata versus NFA



NFA

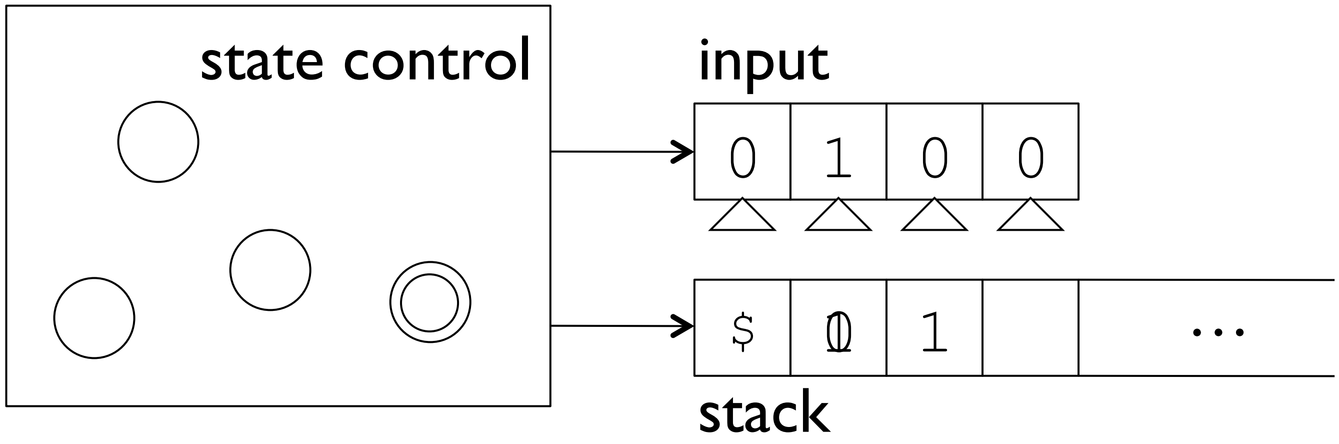
Pushdown automata



pushdown automaton (PDA)

A PDA is like an NFA with but with an infinite stack

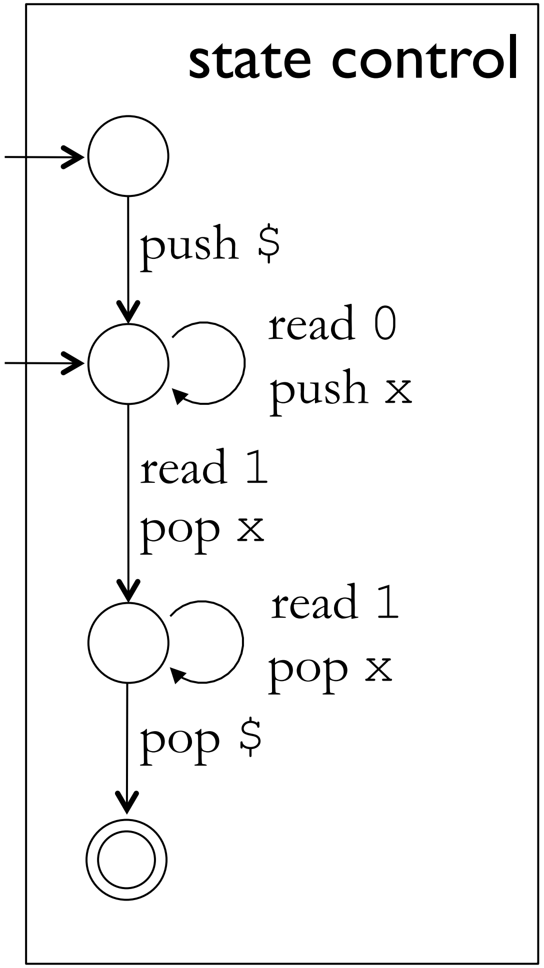
Pushdown automata



pushdown automaton (PDA)

As the PDA is reading the input, it can
push / pop symbols from the top of the stack

Building a PDA



$$L = \{0^n1^n \colon n \geq 1\}$$

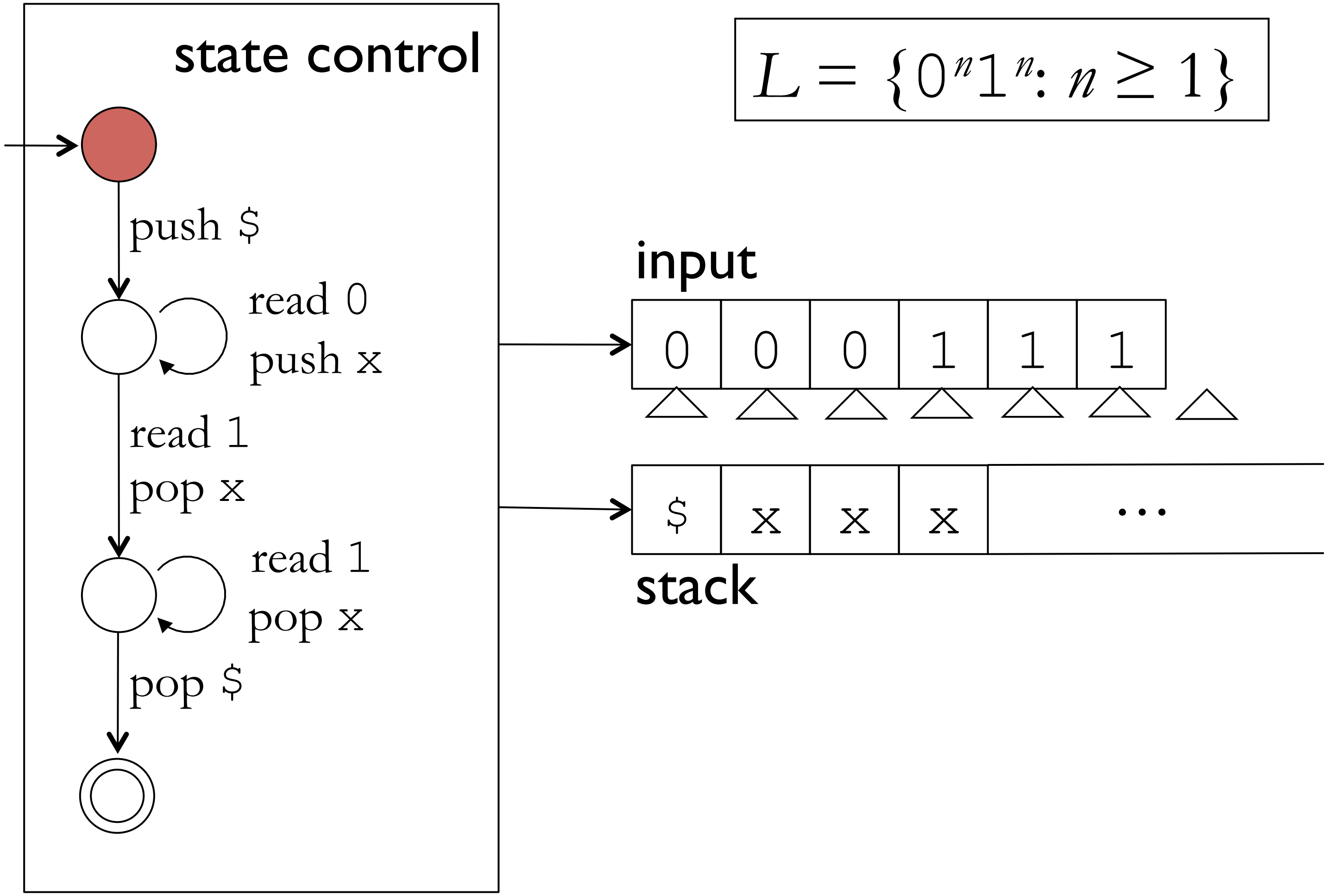
We remember each 0
by pushing x onto the stack

When we see a 1, we
pop an x from the stack

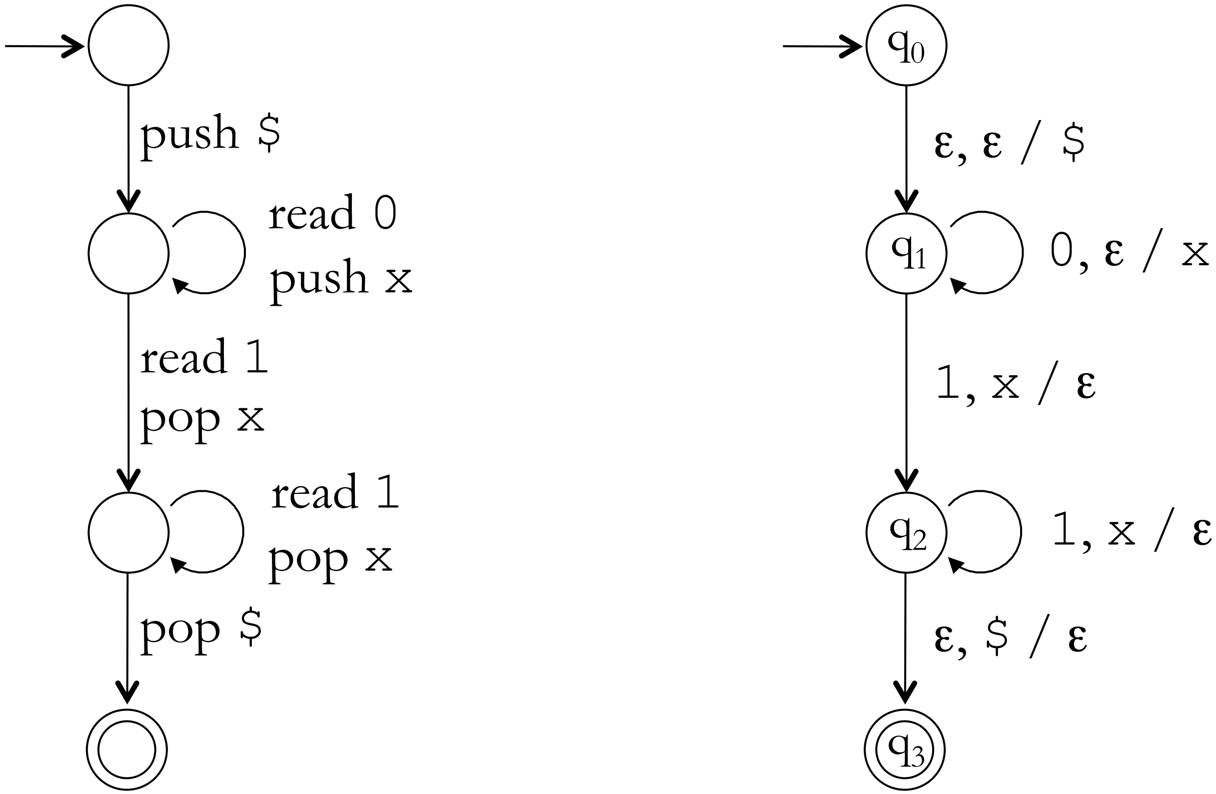
We want to accept when
we hit the stack bottom

We will use $ to mark bottom

A PDA in action



Notation for PDAs



read, pop / push

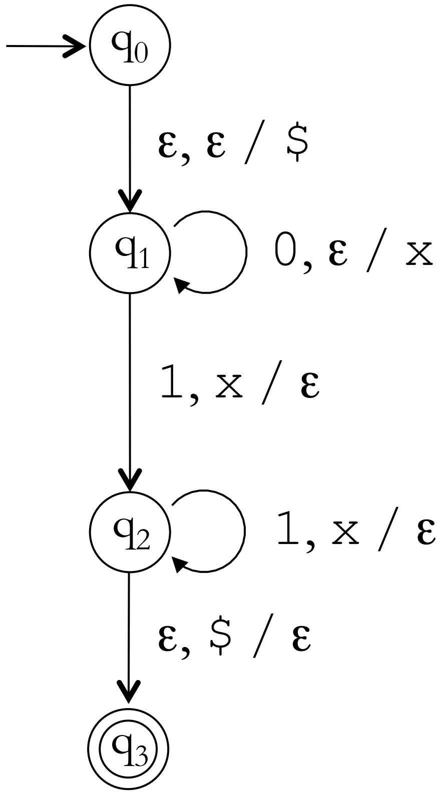
Definition of a PDA

A pushdown automaton is $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

- Q is a finite set of states;
- Σ is the input alphabet;
- Γ is the stack alphabet;
- q_0 in Q is the initial state;
- $F \subseteq Q$ is a set of final states;
- δ is the transition function.

$$\delta\colon \underset{\text{state}}{Q} \times \underset{\text{input symbol}}{(\Sigma \cup \{\varepsilon\})} \times \underset{\text{pop symbol}}{(\Gamma \cup \{\varepsilon\})} \to \text{subsets of } \underset{\text{state}}{Q} \times \underset{\text{push symbol}}{(\Gamma \cup \{\varepsilon\})}$$

Example



$\Sigma = \{0, 1\}$

$\Gamma = \{\$, x\}$

$\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, \$)\}$
$\delta(q_0, \varepsilon, \$) = \varnothing$
$\delta(q_0, \varepsilon, x) = \varnothing$
$\delta(q_0, 0, \varepsilon) = \varnothing$
...

$\underbrace{\delta: Q}_{\text{state}} \times \underbrace{(\Sigma \cup \{\varepsilon\})}_{\text{input symbol}} \times \underbrace{(\Gamma \cup \{\varepsilon\})}_{\text{pop symbol}} \to \text{subsets of } \underbrace{Q}_{\text{state}} \times \underbrace{(\Gamma \cup \{\varepsilon\})}_{\text{push symbol}}$

The language of a PDA

- A PDA is nondeterministic, i.e. multiple transitions on same pop/input are allowed
- Transitions may but do not have to push, pop or read input

The language of a PDA is the set of all strings in Σ^* that can lead the PDA to an accepting state after the whole input is read.

Remark: Sometimes acceptance is defined by emptying the stack or even both: empty stack and accepting state. All of these variants correspond to the same class of languages.

Example 1

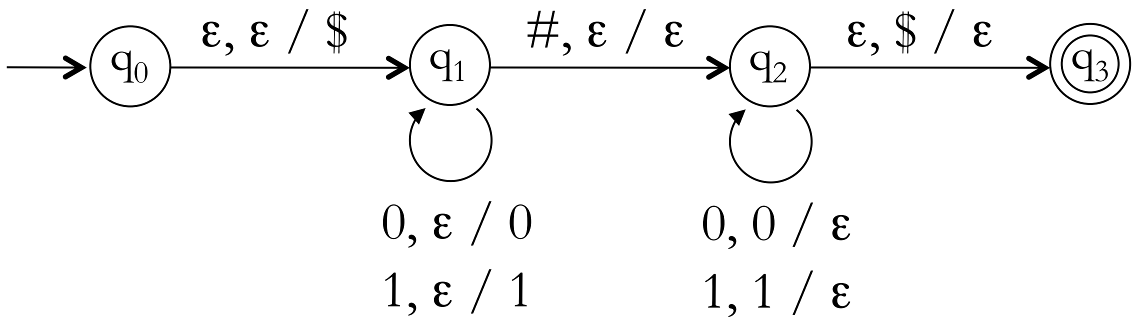
$\boxed{L_1 = \{w\#w^R : w \in \{0, 1\}^*\}}$

$\Sigma = \{0, 1, \#\}$

$\Gamma = \{\$, 0, 1\}$

$\#, 0\#0, 01\#10 \in L_1$

$\varepsilon, 01\#1, 0\#\#0 \notin L_1$



write w on stack

read w^R from stack

Example 2

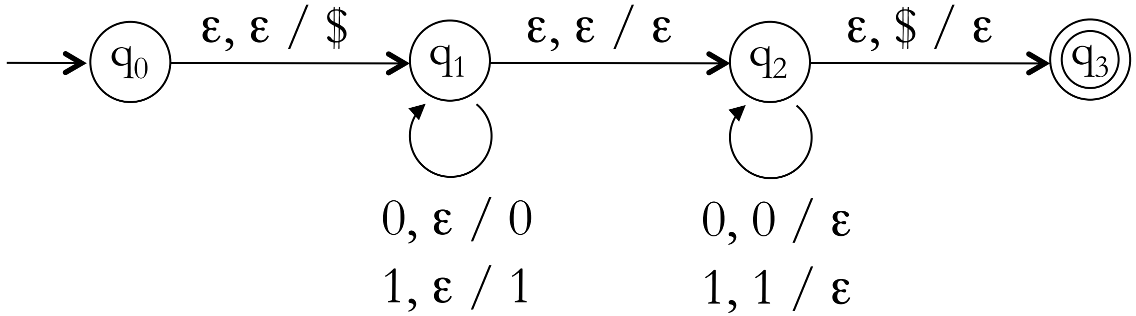
$$\boxed{L_2 = \{ww^R : w \in \Sigma^*\}}$$

$\Sigma = \{0, 1\}$

$\Gamma = \{\$, 0, 1\}$

$\varepsilon, 00, 0110 \in L_2$

$1, 011, 010 \notin L_2$



guess the middle of a string

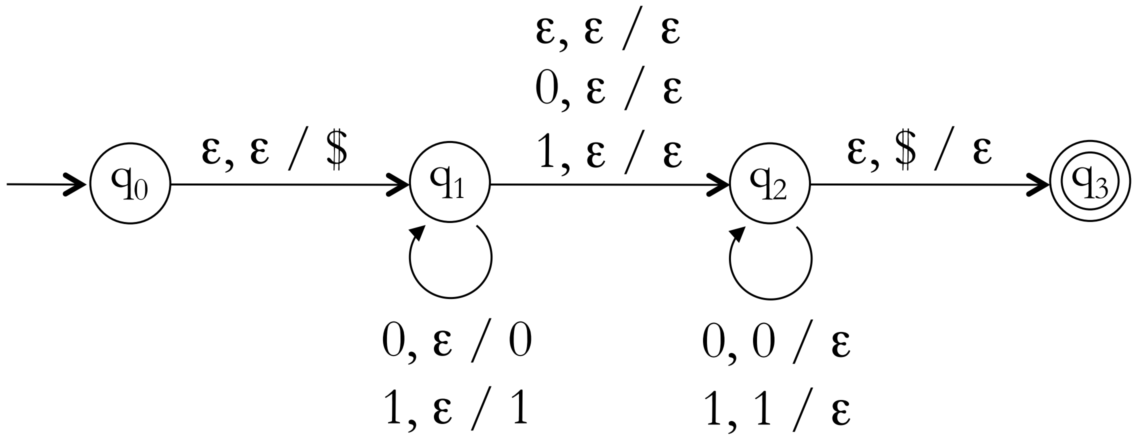
Example 3

$\boxed{L_3 = \{w\colon w = w^R,\ w \in \Sigma^*\}}$ $\Sigma = \{0, 1\}$

$\varepsilon, 1, 00, 010, 0110 \in L_3$

$011 \notin L_3$

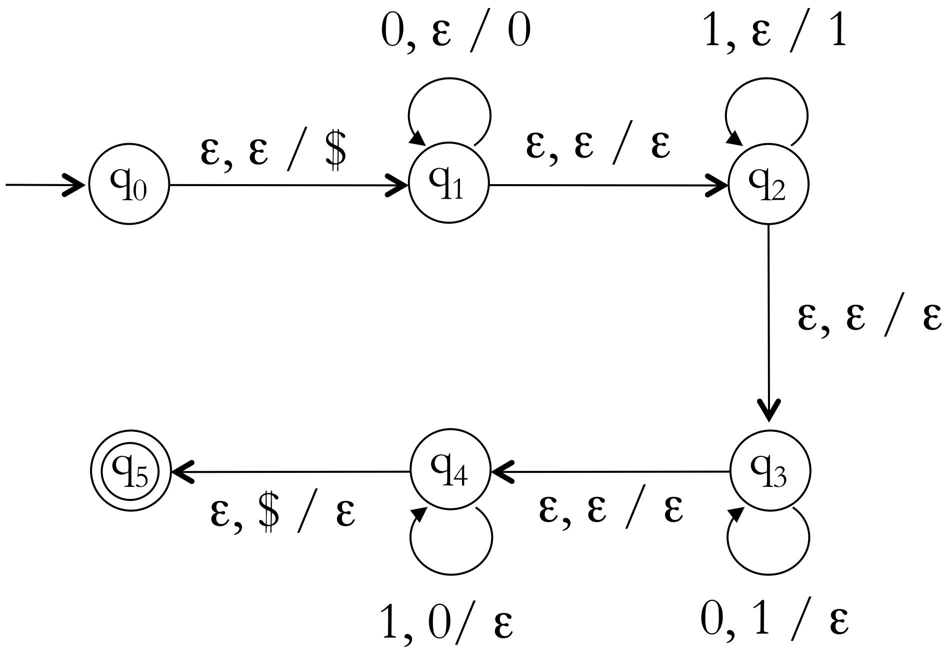
$\underbrace{01101}_{x}\underbrace{10110}_{x^R}$ or $\underbrace{0110}_{x}1\underbrace{0110}_{x^R}$



the middle symbols can be ε, 0, or 1

Example 4

$\boxed{L_4 = \{0^n 1^m 0^m 1^n \mid n \geq 0, m \geq 0\}}$ $\Sigma = \{0, 1\}$



input: $0^n\ 1^m\ 0^m\ 1^n$

stack: $0^n\ 1^m$

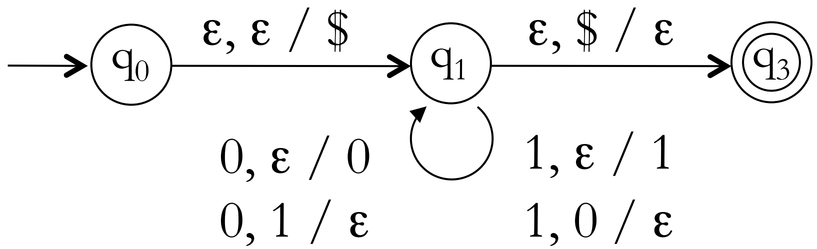
Example 5

$$\boxed{L_5 = \{w\text{: } w \text{ has the same number of 0s and 1s}\}}$$

$\Sigma = \{0, 1\}$

Strategy: Stack keeps track of excess of 0s or 1s

If at the end, stack is empty, the numbers are equal

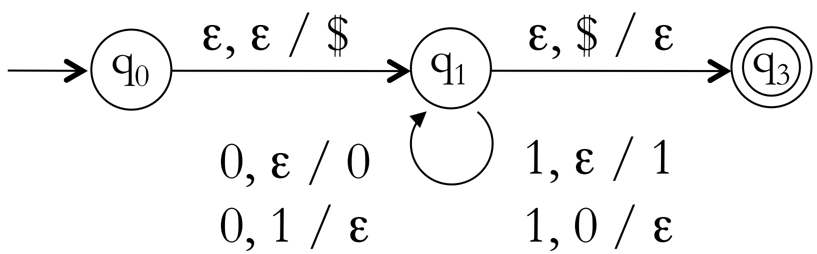


Example 5

$\boxed{L_5 = \{w\text{: } w \text{ has the same number 0s and 1s}\}}$ $\Sigma = \{0, 1\}$

Invariant: In every execution of the PDA:

$\#1 - \#0$ on stack $= \#1 - \#0$ in input so far



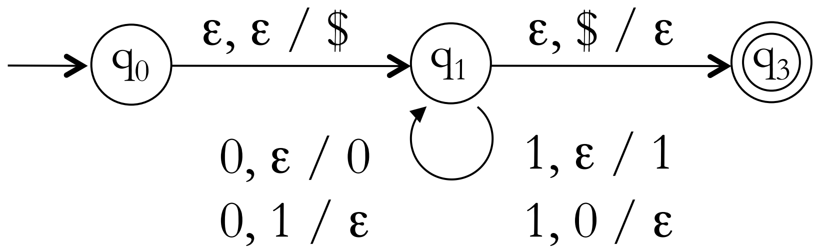
If w is not in L_5, it must be rejected

Example 5

$\boxed{L_5 = \{w\text{: } w \text{ has the same number 0s and 1s}\}}$ $\Sigma = \{0, 1\}$

Property: In some execution of the PDA:

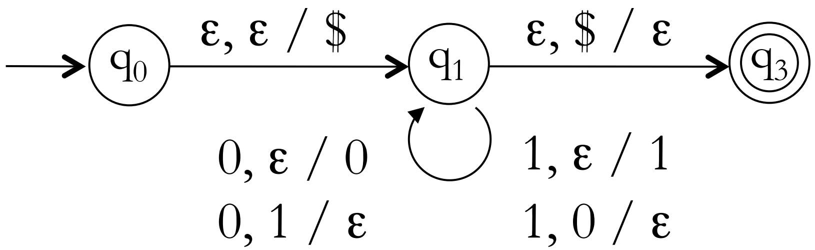
stack consists only of 0s or only of 1s (or ε)



If w is in L_5, some execution will accept

Example 5

$\boxed{L_5 = \{w\text{: } w \text{ has the same number 0s and 1s}\}}$ $\Sigma = \{0, 1\}$



$w = 001110$

read	stack
0	\$0
0	\$00
1	\$0
1	\$
1	\$1
0	\$

Example 6

$L_6 = \{w$: w has two 0-blocks with the same number of 0s

01011, 001011001, 10010101001

allowed

01001000, 01111

not allowed

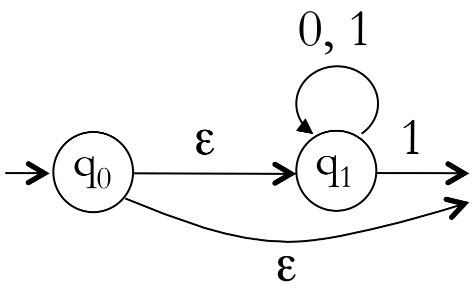
Strategy: Detect start of the first 0-block

Push 0s on the stack

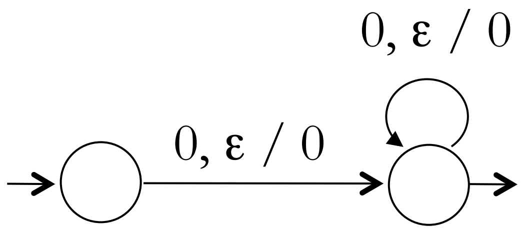
Detect start of the second 0-block

Pop 0s from the stack

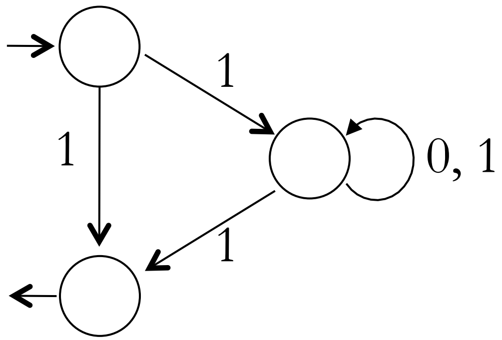
Example 6



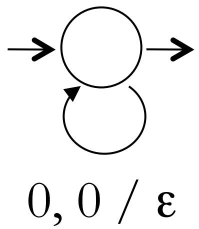
1 Detect start of first 0-block



2 Push 0s on stack



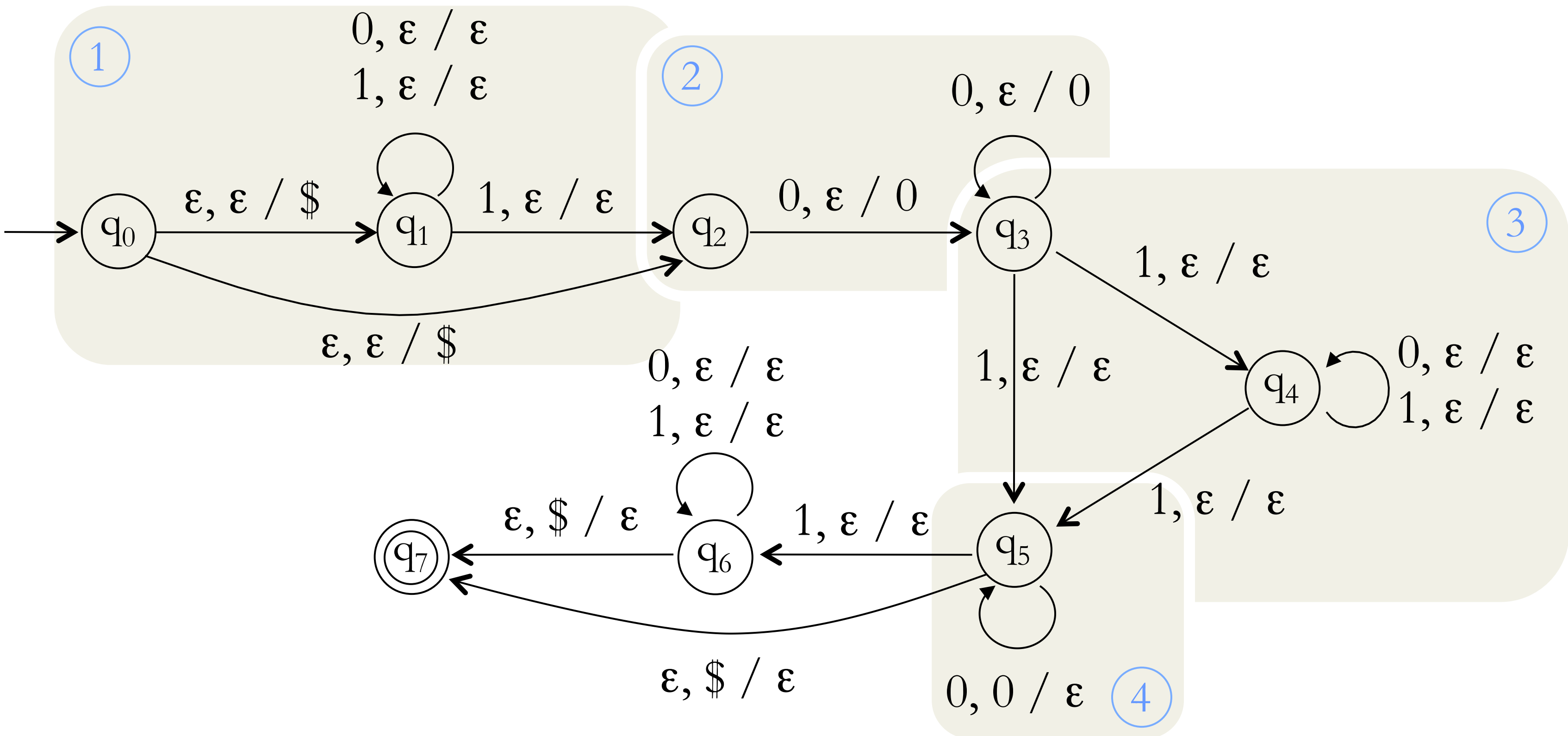
3 Detect start of second 0-block



4 Pop 0s from stack

Example 6

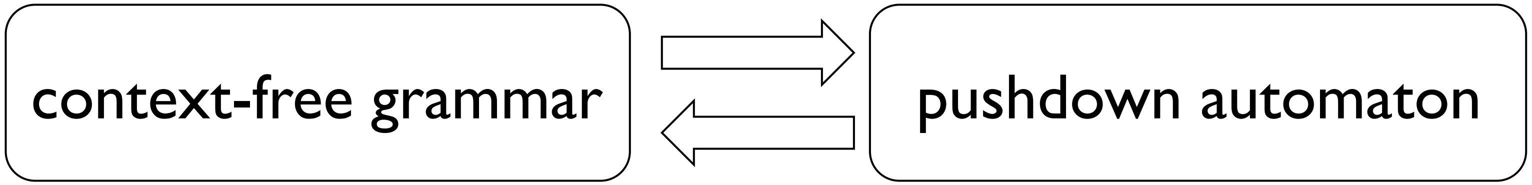
$L_6 = \{w\text{: } w \text{ has two 0-blocks with the same number of 0s}\}$



CFG $\leftrightarrow$ PDA conversions

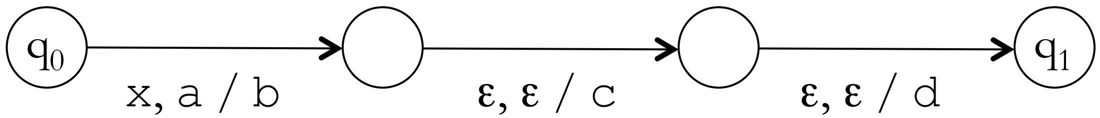
CFGs and PDAs

L has a context-free grammar if and only if it is accepted by some pushdown automaton.



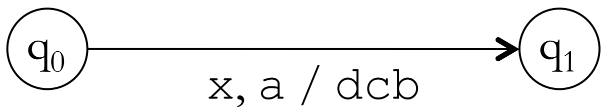
A convention

- When we have a sequence of transitions like:



pop a, then push b, c, and d

- We will abbreviate it like this:



replace a by dcb on the top of the stack
(notice the reverse order: the first symbol
of the word is at the top of the stack)

Converting a CFG to a PDA

- Idea: Use PDA to simulate derivations

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

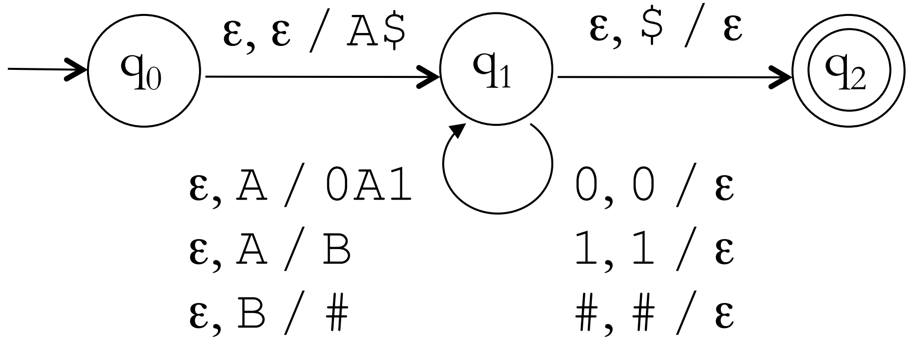
$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00\#11$$

PDA control:	stack:	input:
write start variable	$A	00#11
replace production in reverse	$1A0	00#11
pop terminal and match	$1A	0#11
replace production in reverse	$11A0	0#11
pop terminal and match	$11A	#11
replace production in reverse	$11B	#11

Converting a CFG to a PDA

$A \rightarrow 0A1$
$A \rightarrow B$
$B \rightarrow \#$

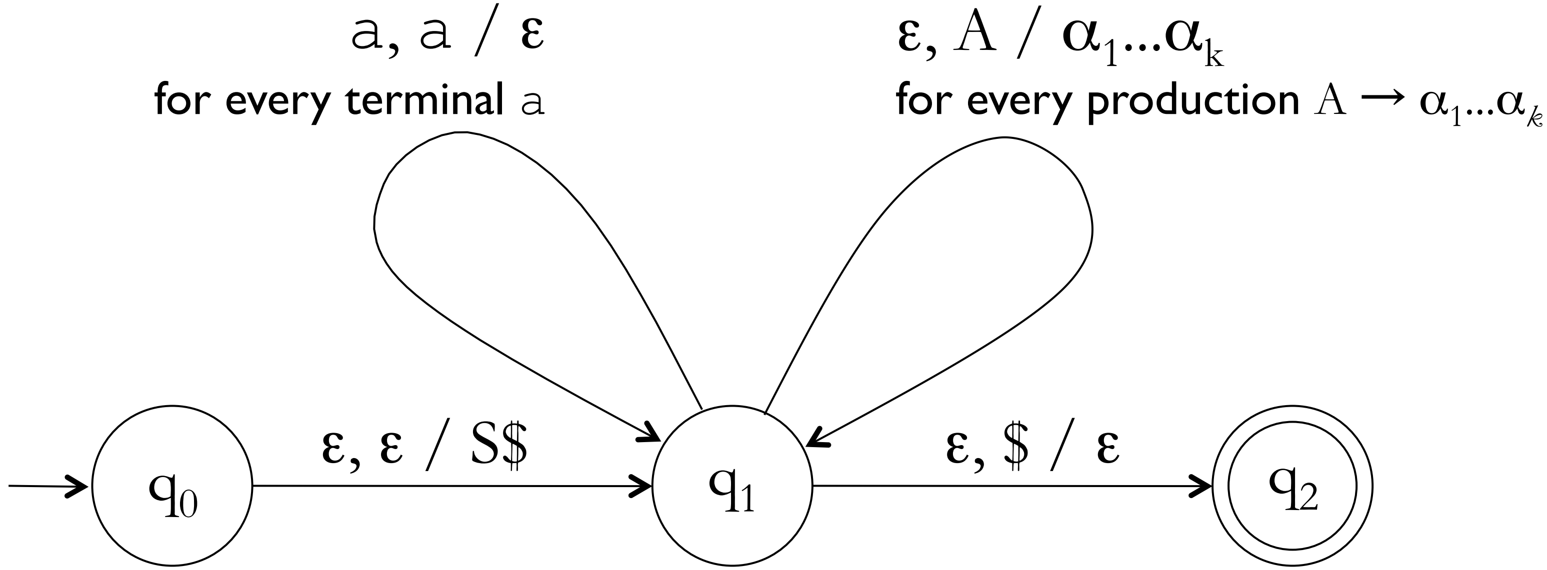
CFG



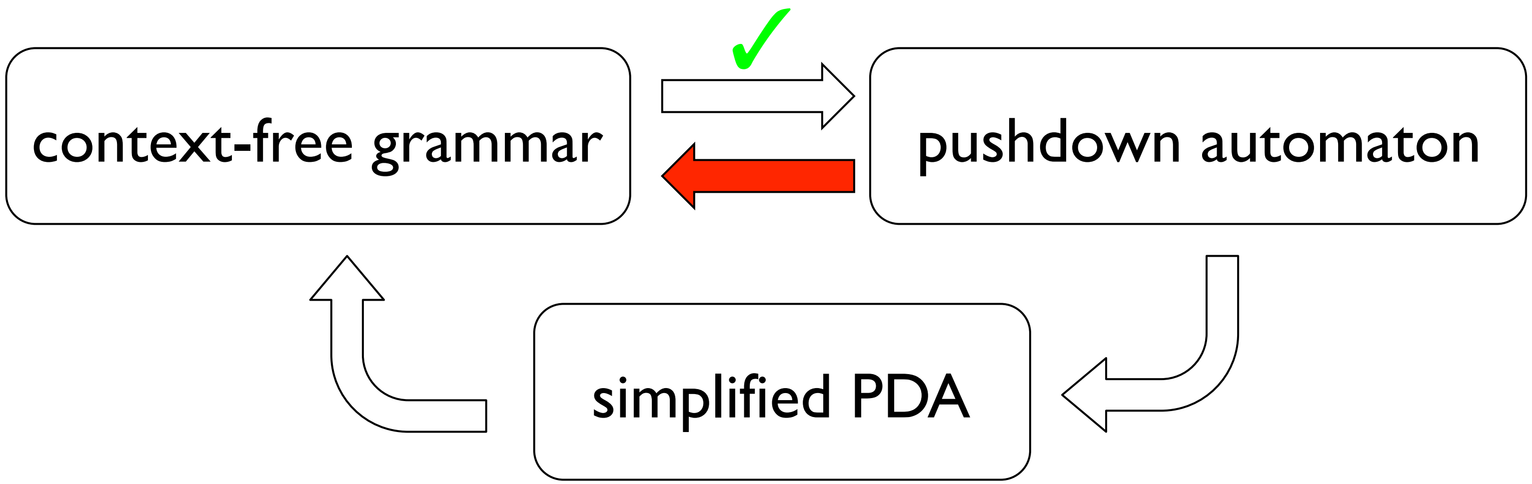
input	00#11	00#11	00#11	00#11	00#11
stack	$A	$1A0	$1A	$11A0	$11A
input	00#11	00#11	00#11	00#11	00#11
stack	$11B	$11#	$11	$1	$

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00\#11$$

General CFG to PDA conversion

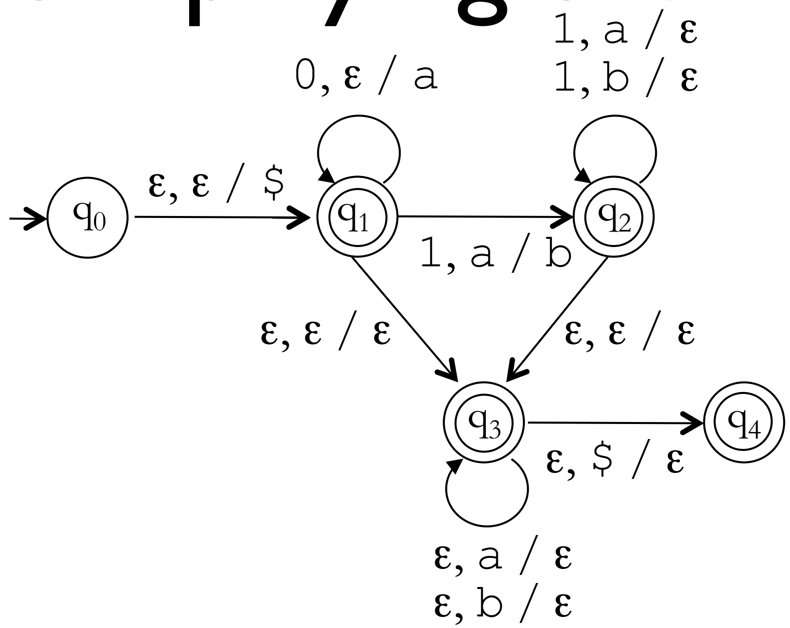


From PDAs to CFGs



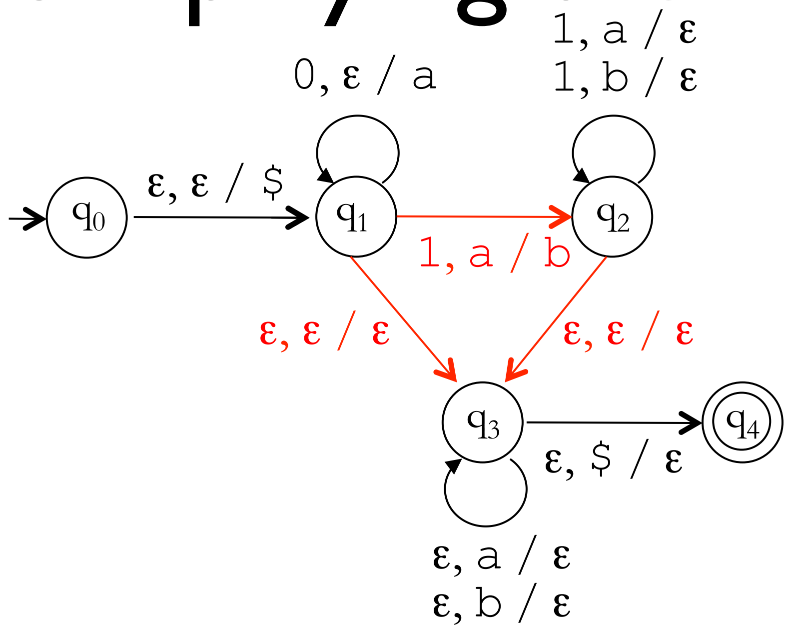
- A simplified PDA:
  - Has a single accept state
  - Empties its stack before accepting
  - Each transition is either a push, or a pop, but not both

Simplifying the PDA



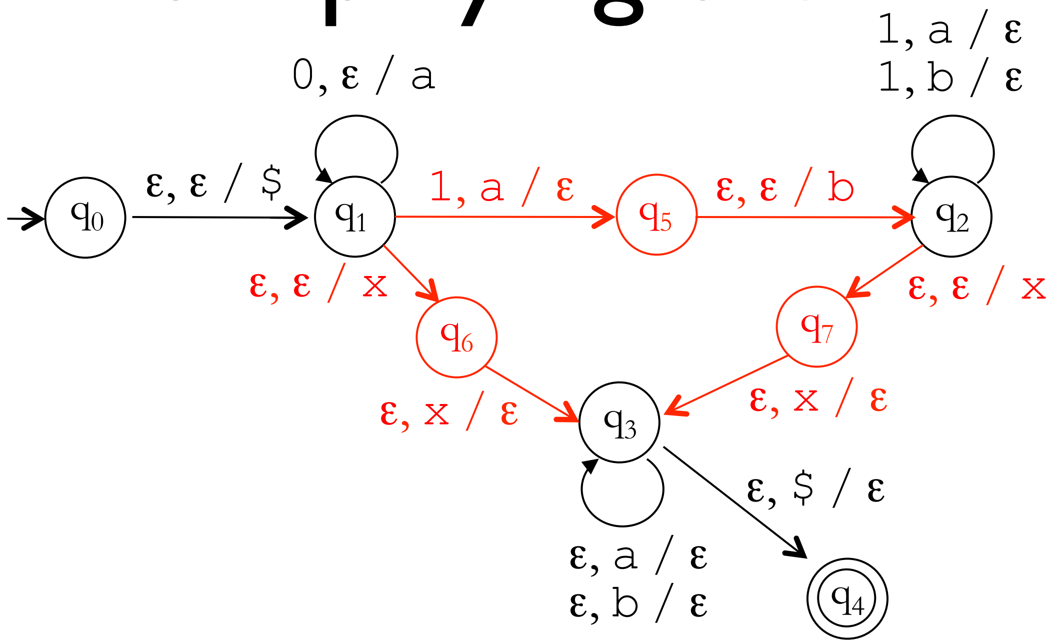
- A simplified PDA:
  - Has a single accept state ✓
  - Empties its stack before accepting ✓
  - Each transition is either a push, or a pop, but not both

Simplifying the PDA



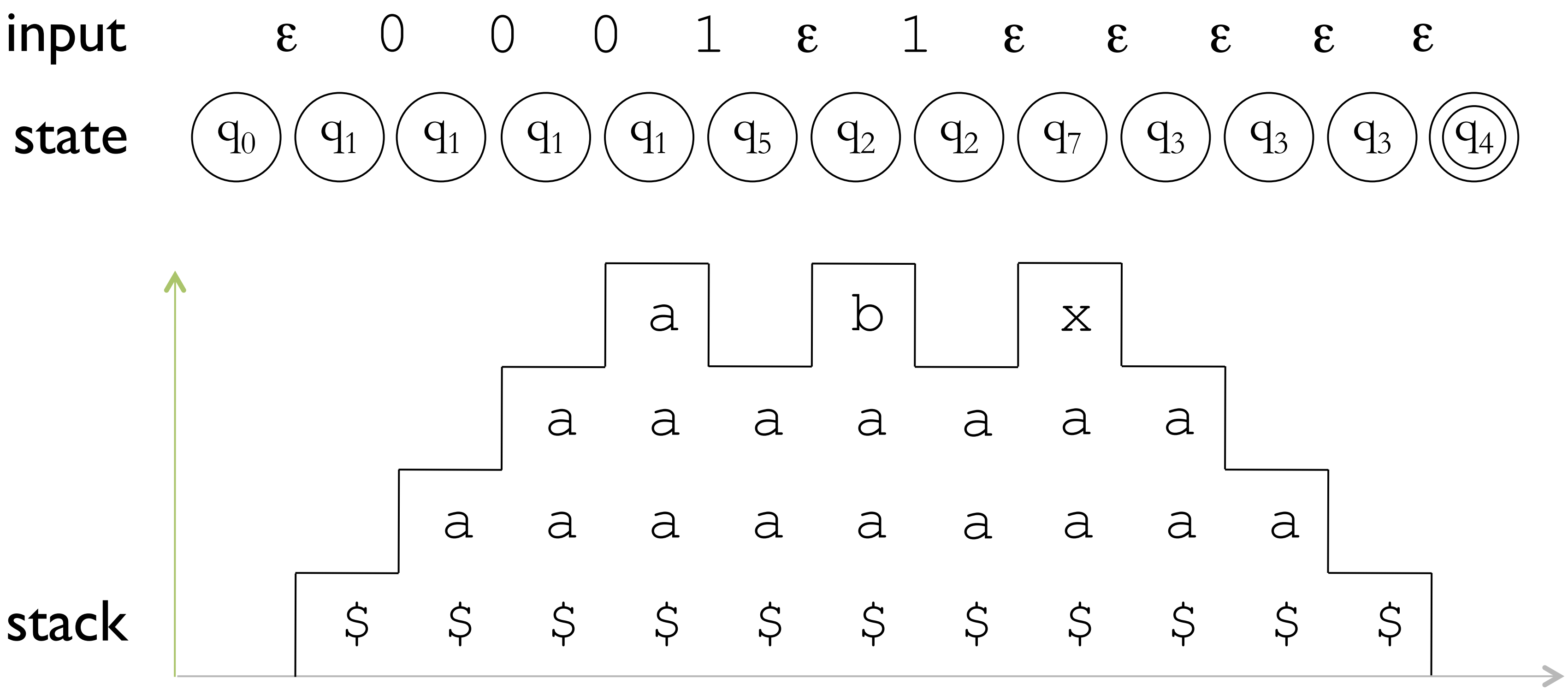
- A simplified PDA:
  - Has a single accept state ✓
  - Empties its stack before accepting ✓
  - Each transition is either a push, or a pop, but not both

Simplifying the PDA



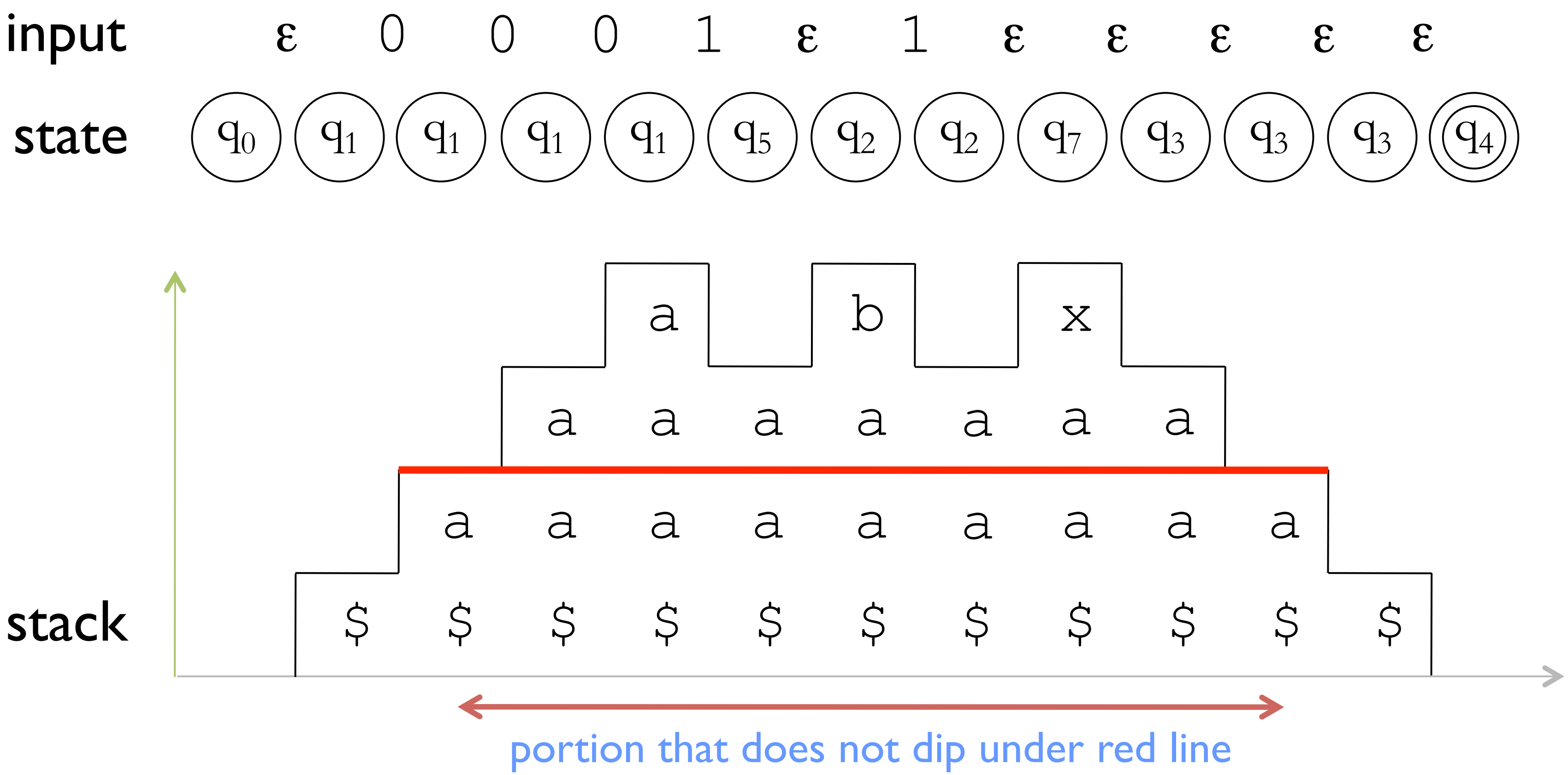
- A simplified PDA:
  - Has a single accept state ✓
  - Empties its stack before accepting ✓
  - Each transition is either a push, or a pop, but not both ✓

Simplified PDA to CFG



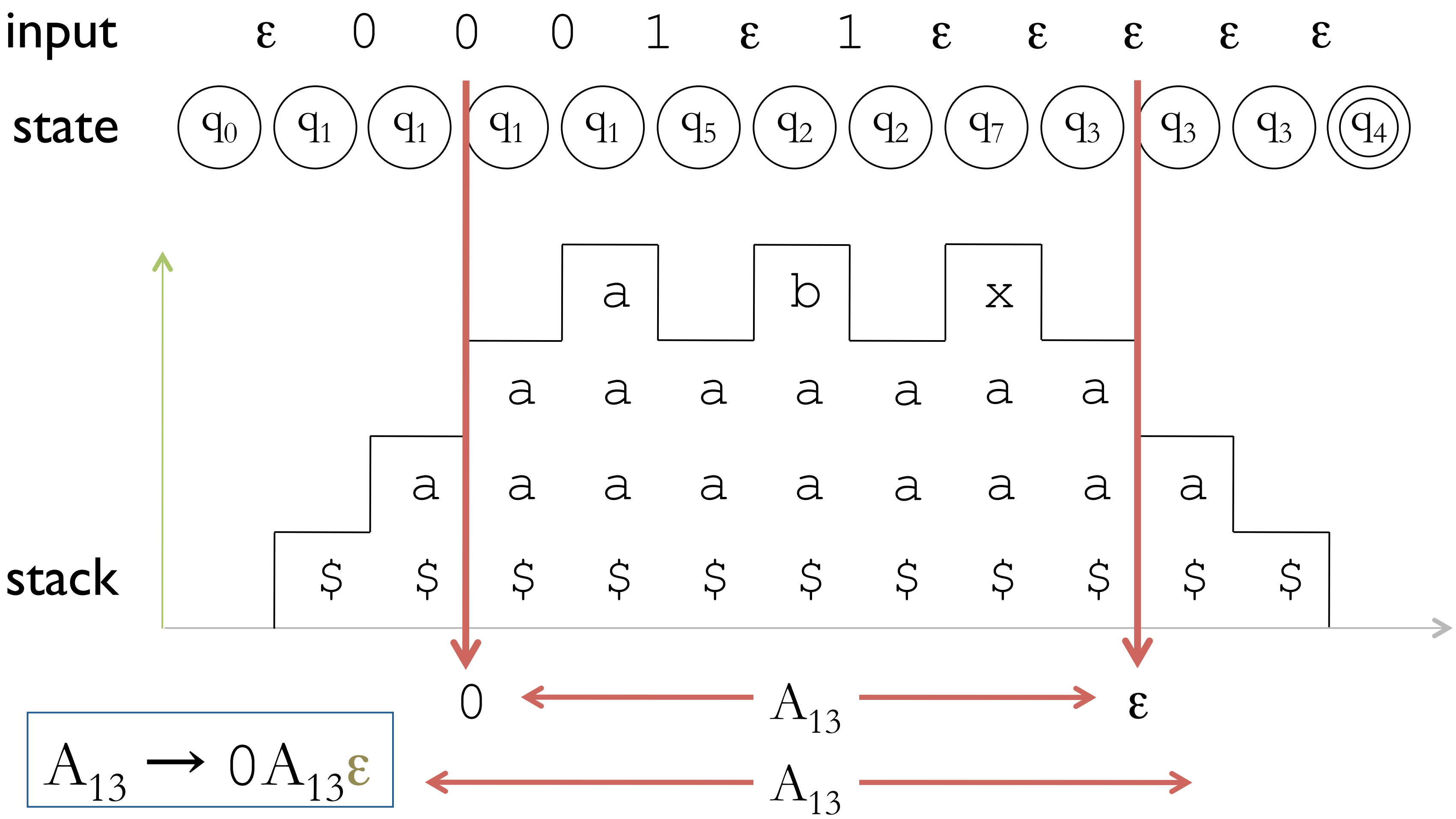
A sample run of the PDA on input `00011`

Simplified PDA to CFG



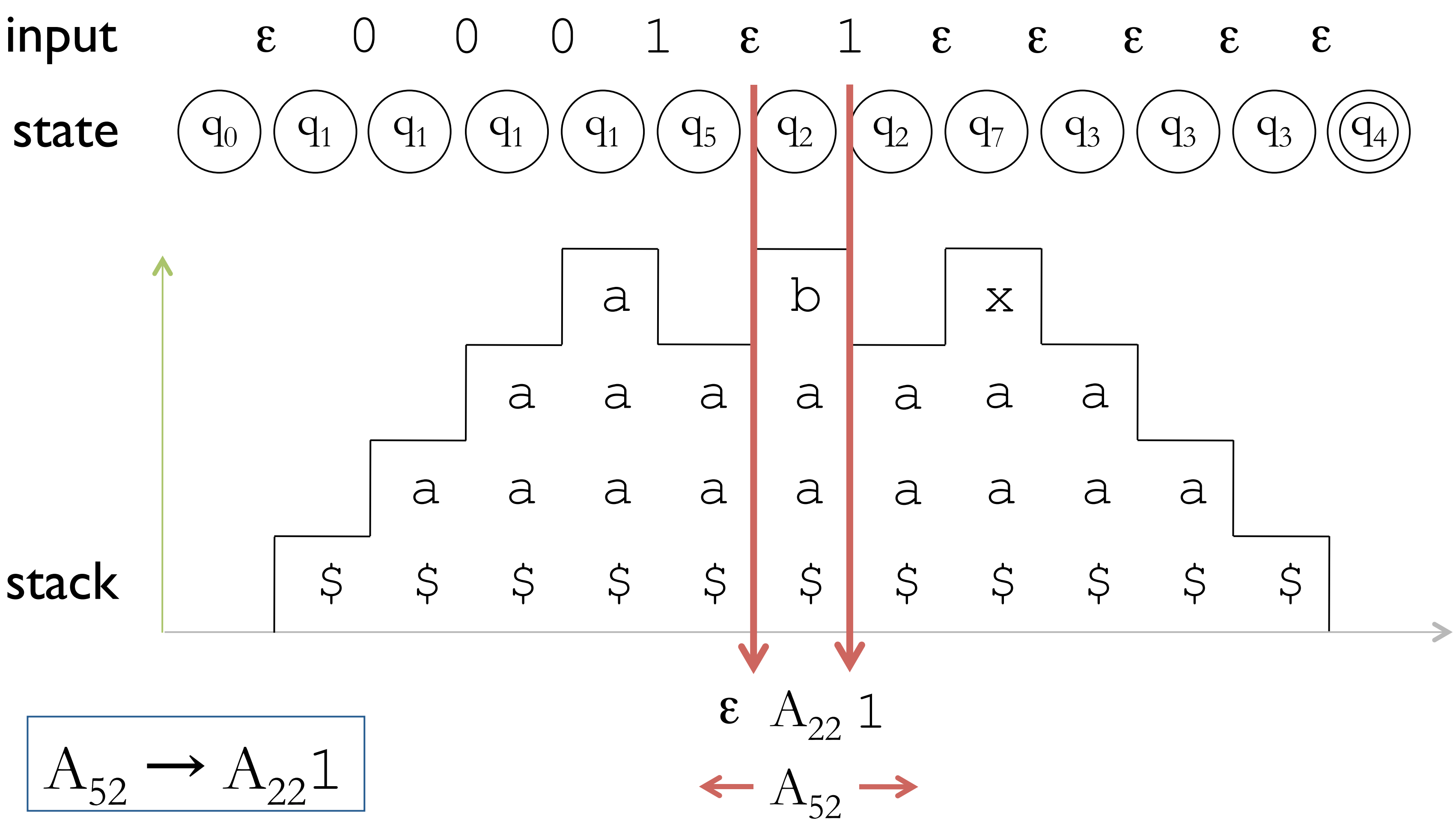
$A_{13} = \{x\text{: } x \text{ leads from } q_1 \text{ to } q_3 \text{ and does not dip under red line}\}$

Simplified PDA to CFG



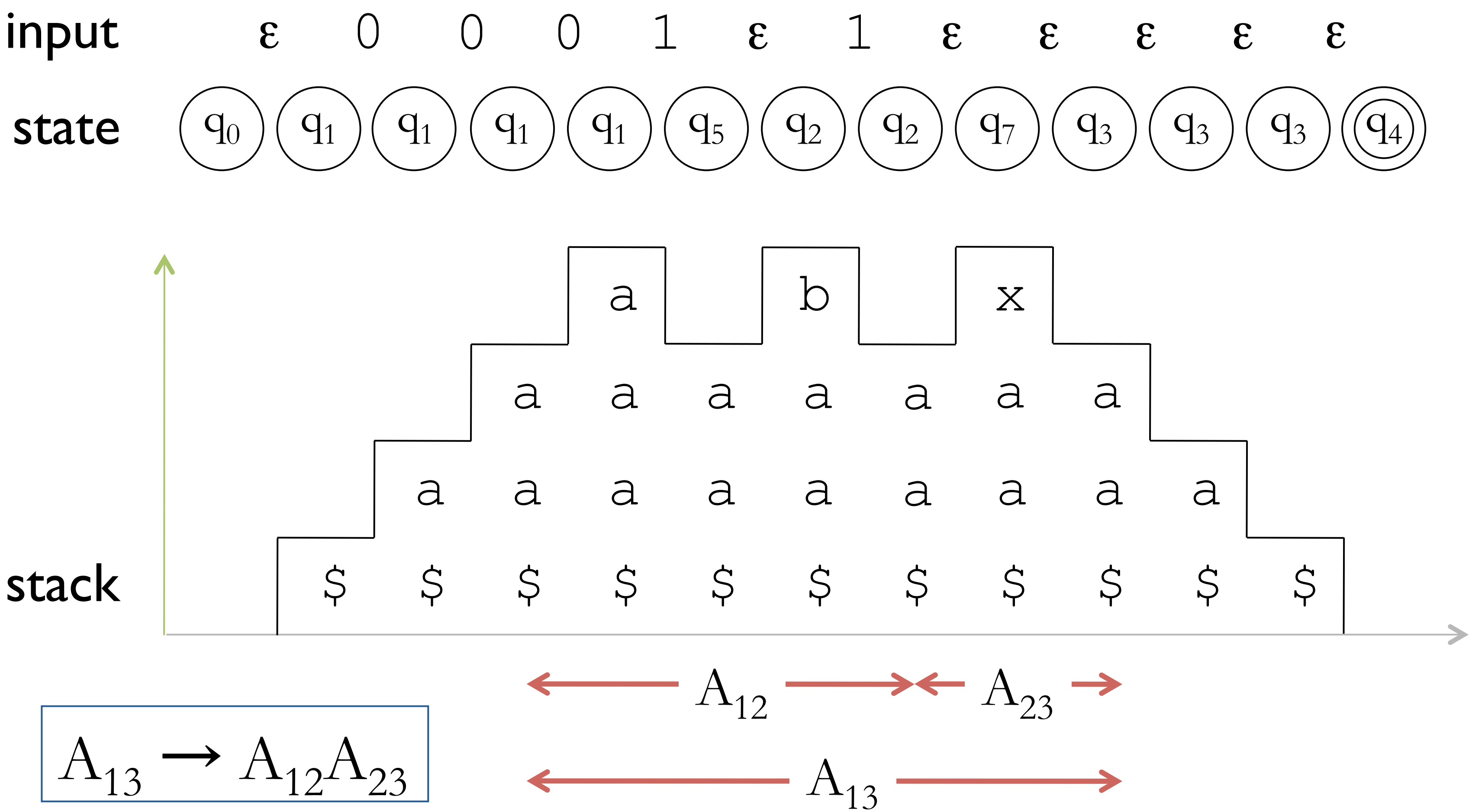
$A_{13} \rightarrow 0A_{13}\varepsilon$

Simplified PDA to CFG



$$A_{52} \rightarrow A_{22}1$$

Simplified PDA to CFG

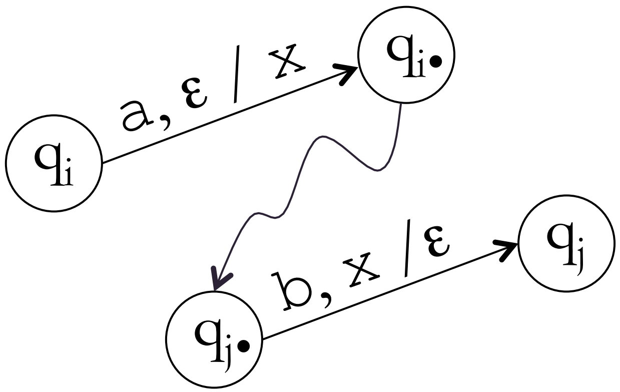


$$\boxed{A_{13} \rightarrow A_{12}A_{23}}$$

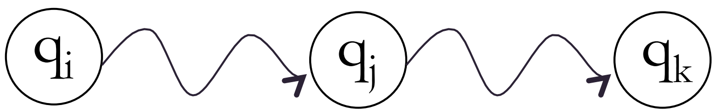
Simplified PDA to CFG

variables: A_{ij}

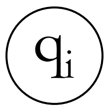
start variable: A_{0f}



$$A_{ij} \rightarrow a A_{i\bullet j\bullet} b$$

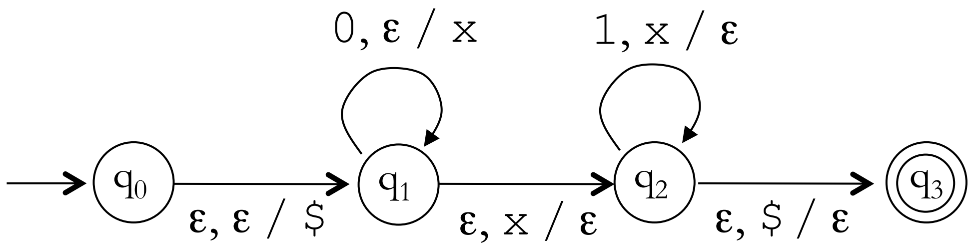


$$A_{ik} \rightarrow A_{ij} A_{jk}$$



$$A_{ii} \rightarrow \varepsilon$$

Example: Simplified PDA to CFG



variables: $A_{01}, A_{02}, A_{03}, A_{11}, A_{12}, A_{13}, A_{22}, A_{23}$

start variable: A_{03}

productions:

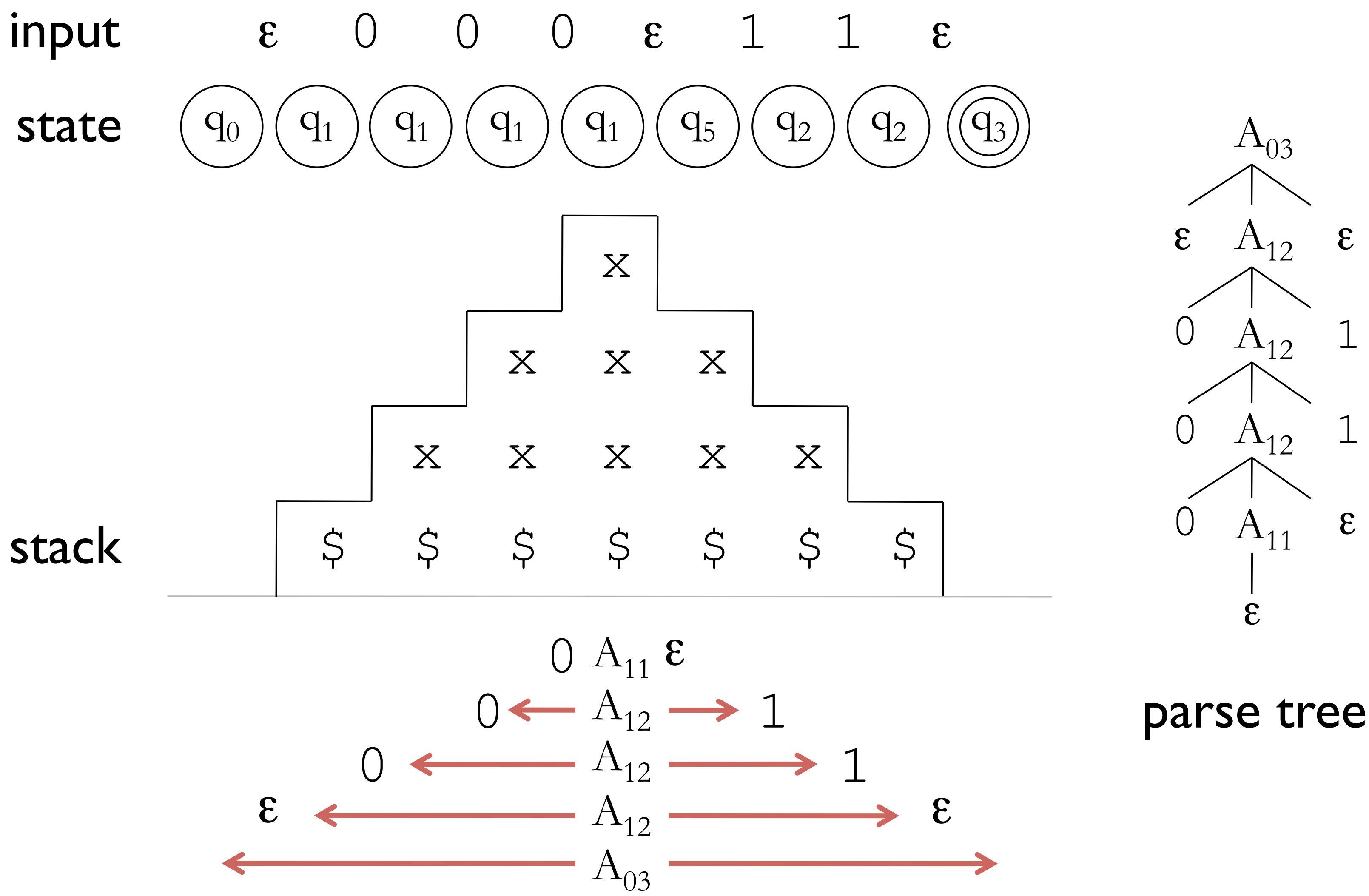
$A_{01} \rightarrow A_{01}A_{11}$
$A_{02} \rightarrow A_{01}A_{12}$
$A_{02} \rightarrow A_{02}A_{22}$
$A_{03} \rightarrow A_{01}A_{13}$
$A_{03} \rightarrow A_{02}A_{23}$
$A_{11} \rightarrow A_{11}A_{11}$

$A_{12} \rightarrow A_{11}A_{12}$
$A_{12} \rightarrow A_{12}A_{22}$
$A_{13} \rightarrow A_{11}A_{13}$
$A_{13} \rightarrow A_{12}A_{23}$
$A_{22} \rightarrow A_{22}A_{22}$
$A_{23} \rightarrow A_{22}A_{23}$

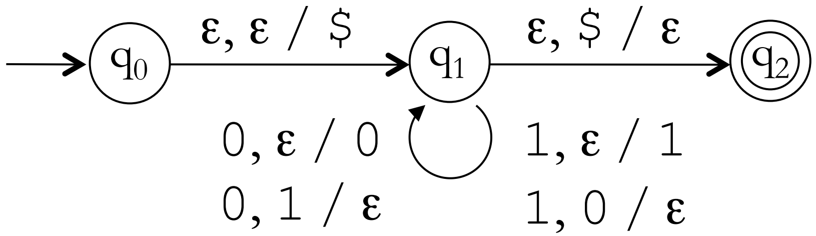
$A_{00} \rightarrow \varepsilon$
$A_{11} \rightarrow \varepsilon$
$A_{22} \rightarrow \varepsilon$
$A_{33} \rightarrow \varepsilon$

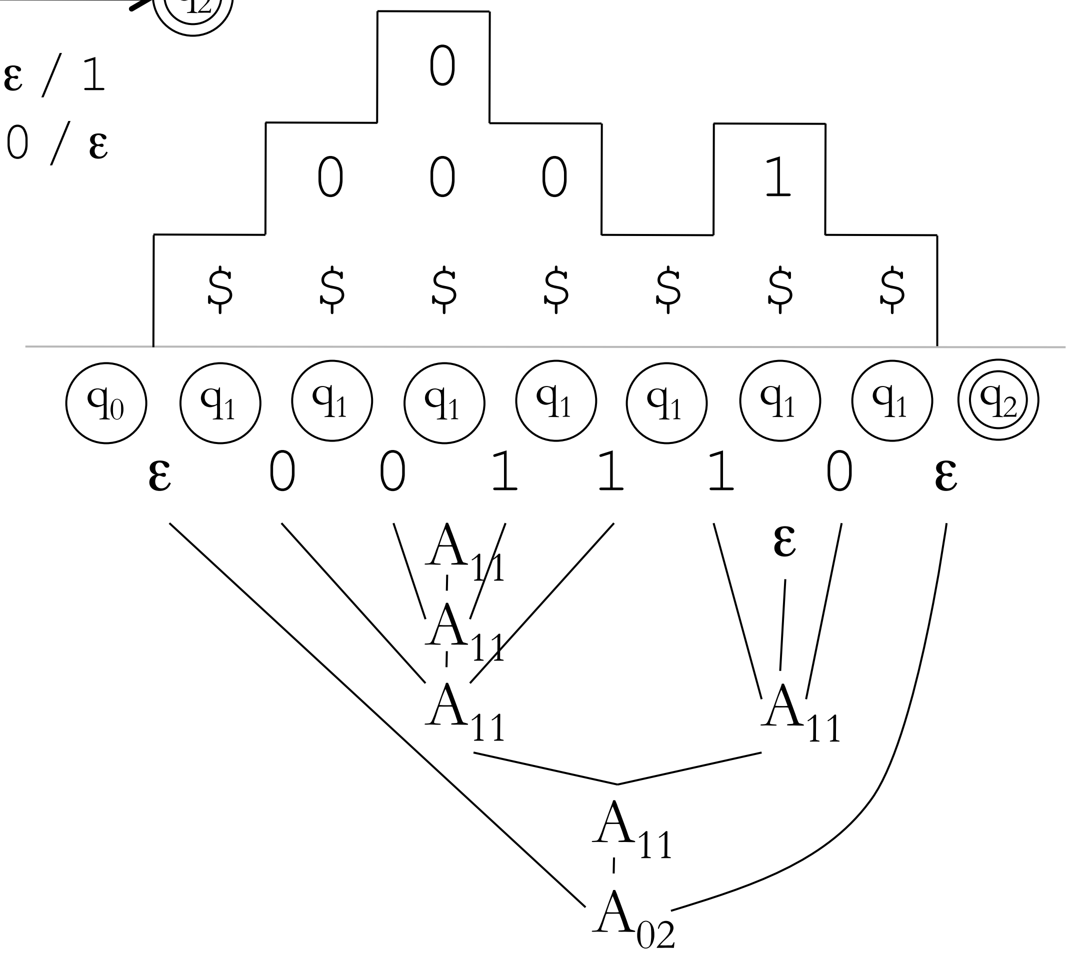
$A_{12} \rightarrow 0A_{12}1$
$A_{12} \rightarrow 0A_{11}$
$A_{03} \rightarrow A_{12}$

Example: Simplified PDA to CFG



Another example





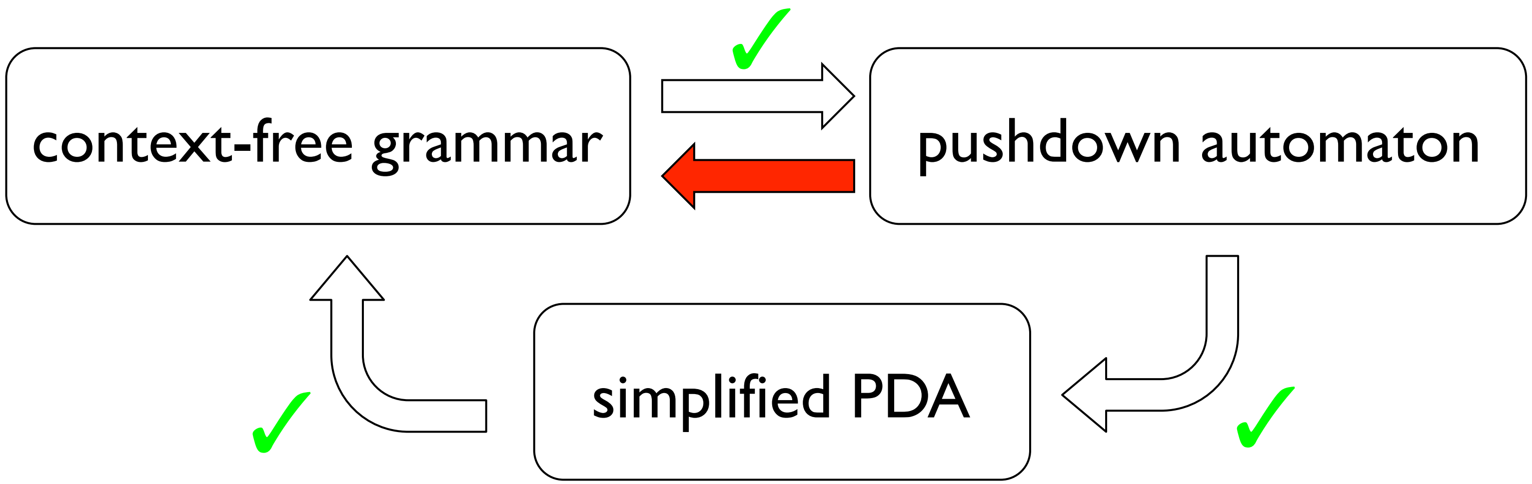
productions:

$A_{01} \rightarrow A_{01}A_{11}$
$A_{02} \rightarrow A_{01}A_{12}$
$A_{12} \rightarrow A_{11}A_{12}$
$A_{11} \rightarrow A_{11}A_{11}$

$A_{11} \rightarrow 0A_{11}1$
$A_{11} \rightarrow 1A_{11}0$
$A_{02} \rightarrow A_{11}$

$A_{00}, A_{11}, A_{22} \rightarrow \varepsilon$

From PDAs to CFGs



- A simplified PDA:
  - Has a single accept state
  - Empties its stack before accepting
  - Each transition is either a push, or a pop, but not both

Limitations of pushdown automata

Non context-free languages

$L_1 = \{\mathtt{a}^n\mathtt{b}^n \colon n \geq 0\}$ ✔

$L_2 = \{s \colon s$ has the same number of $\mathtt{a}$s and $\mathtt{b}$s$\}$ ✔

$L_3 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \colon n \geq 0\}$?

$L_4 = \{ss^R \colon s \in \{\mathtt{a}, \mathtt{b}\}^*\}$

$L_5 = \{ss \colon s \in \{\mathtt{a}, \mathtt{b}\}^*\}$

These are not regular

Are they context-free?

An attempt

$$L_3 = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \colon n \geq 0\}$$

- Let's try to design a CFG or PDA

$S \rightarrow \mathrm{aBc} \mid \varepsilon$
$B \rightarrow$??

read a / push x

read b / pop x

???

What would happen if...

- Suppose we could construct some CFG for L_3, e.g.

$$
\begin{aligned}
S &\rightarrow BC \\
B &\rightarrow CS \mid \mathtt{b} \\
C &\rightarrow SB \mid \mathtt{a} \\
&\;\ldots
\end{aligned}
$$

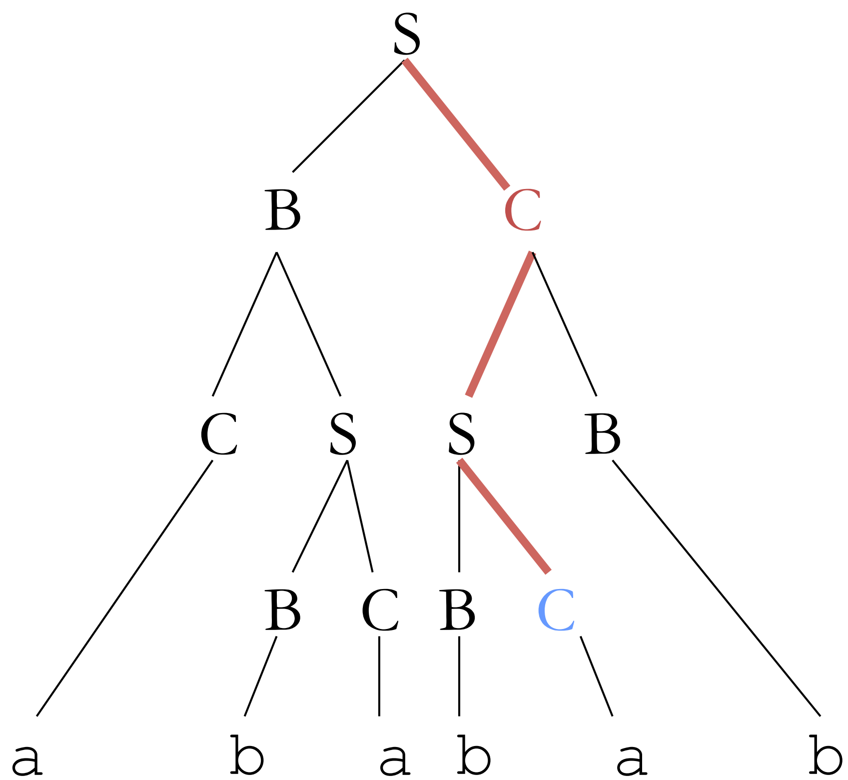
- Let's do some long derivations

$$
\begin{aligned}
S &\Rightarrow BC \\
&\Rightarrow CSC \\
&\Rightarrow \mathtt{a}SC \\
&\Rightarrow \mathtt{a}BCC \\
&\Rightarrow \mathtt{ab}CC \\
&\Rightarrow \mathtt{aba}C \\
&\Rightarrow \mathtt{aba}SB \\
&\Rightarrow \mathtt{aba}BCB \\
&\Rightarrow \mathtt{abab}CB \\
&\Rightarrow \mathtt{ababa}B \\
&\Rightarrow \mathtt{ababab}
\end{aligned}
$$

Repetition in long derivations

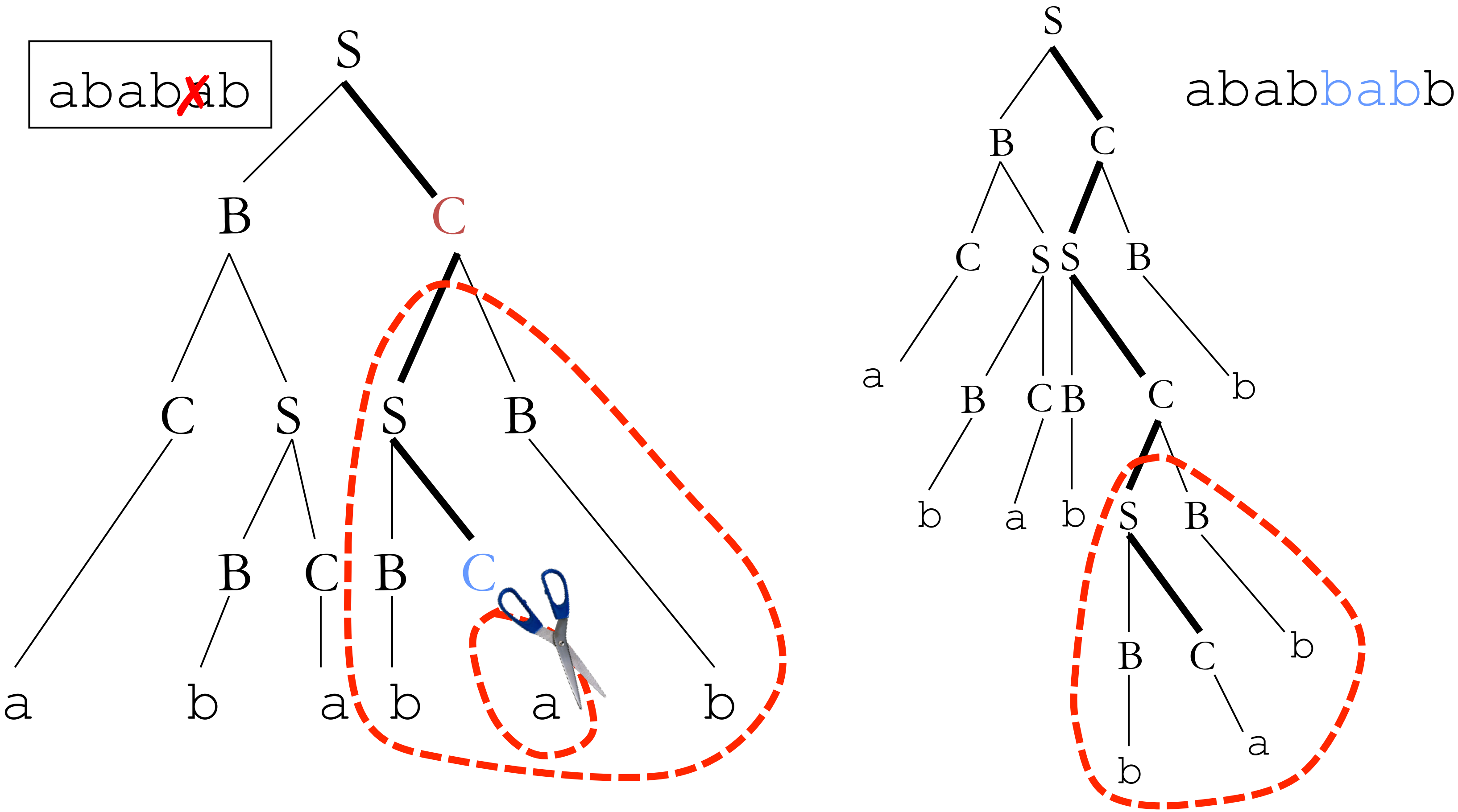
- If a derivation is long enough, some variable must appear twice on the same path in a parse tree

$$
\begin{aligned}
S &\Rightarrow BC \\
&\Rightarrow CSC \\
&\Rightarrow \mathtt{a}SC \\
&\Rightarrow \mathtt{a}BCC \\
&\Rightarrow \mathtt{ab}CC \\
&\Rightarrow \mathtt{aba}C \\
&\Rightarrow \mathtt{aba}SB \\
&\Rightarrow \mathtt{aba}BCB \\
&\Rightarrow \mathtt{abab}CB \\
&\Rightarrow \mathtt{ababa}B \\
&\Rightarrow \mathtt{ababab}
\end{aligned}
$$



Pumping example

- Then we can "cut and paste" part of the parse tree



Pumping example

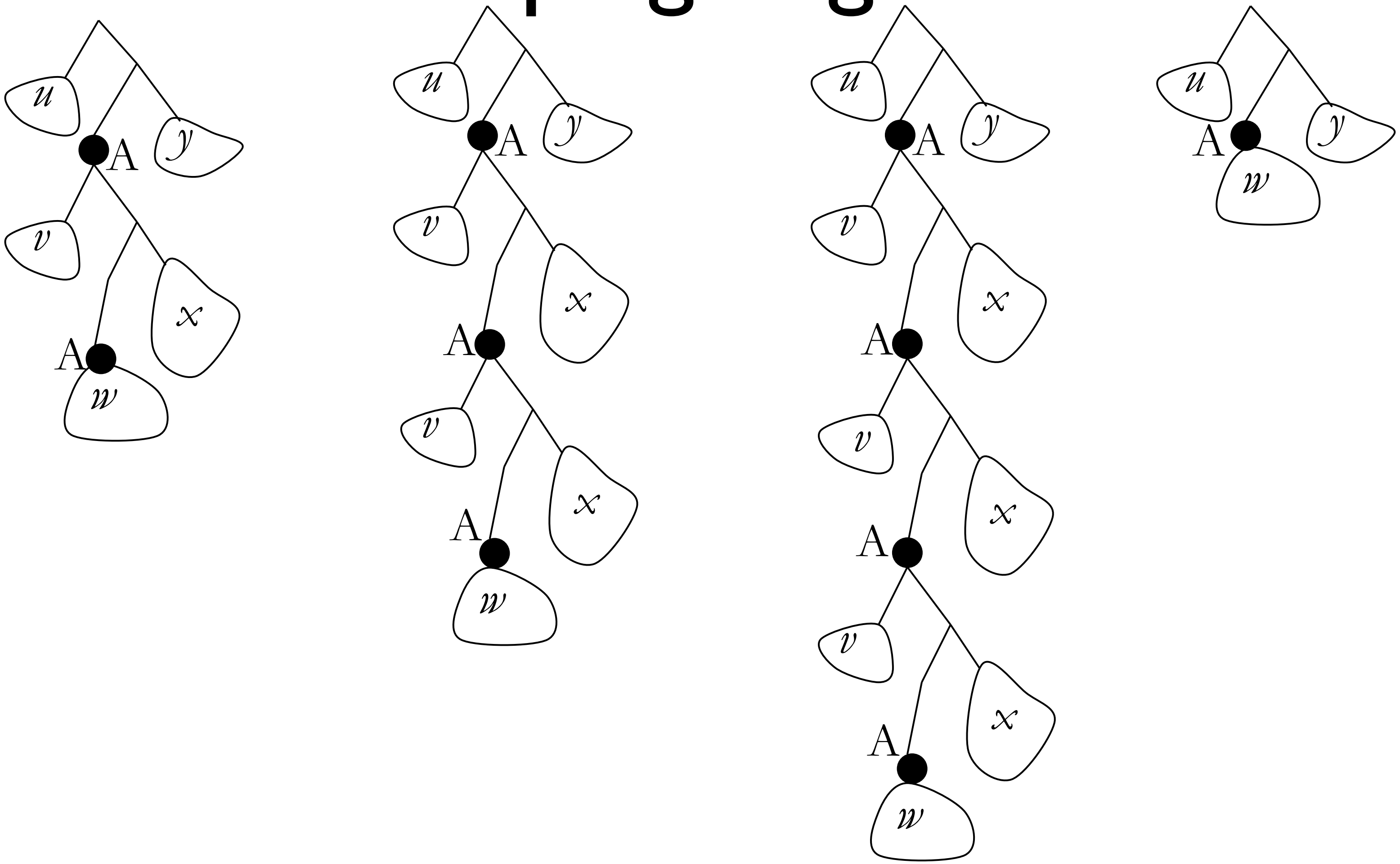
- We can repeat this many times

abab~~a~~b ⇒ ababb~~a~~bb ⇒ ababbbabbbb

⇒ $abab^{n}ab^{n}bb$

- Every sufficiently large derivation will have a middle part that can be repeated indefinitely

Pumping in general



$\boxed{uvwxy}$ $\boxed{uv^2wx^2y}$ $\boxed{uv^3wx^3y}$ $\boxed{uwy}$

Example

$$L_3 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$$

- If L_3 has a context-free grammar G, then

  $\boxed{\text{If } uvwxy \text{ is in } G\text{, so are } uv^2wx^2y,\ uv^3wx^3y,\ uwy, \ldots}$

- What happens for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$?

  $$\underbrace{\mathtt{a\ a\ a}}_{u}\ \underbrace{\ldots\ \mathtt{a\ a\ b\ b\ b}}_{v}\ \underbrace{\ldots\ \mathtt{b}}_{w}\ \underbrace{\mathtt{b\ c\ c\ c}}_{x}\ \underbrace{\ldots\ \mathtt{c\ c}}_{y}$$

- No matter how it is split, $uv^2wx^2y \notin L_3$!

Pumping lemma for context-free languages

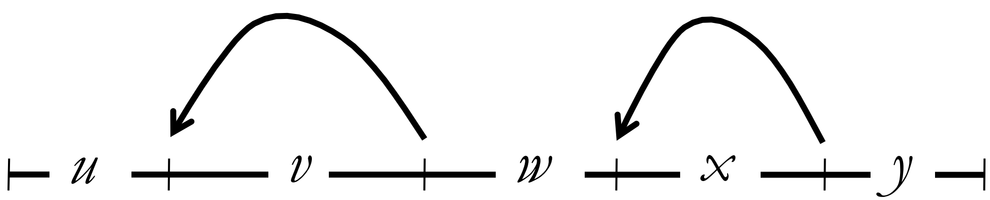
- Pumping lemma: For every context-free language L

There exists a number n such that for every string z in L longer than n, we can write $z = uvwxy$ where

① $|vwx| \leq n$

② $|vx| \geq 1$

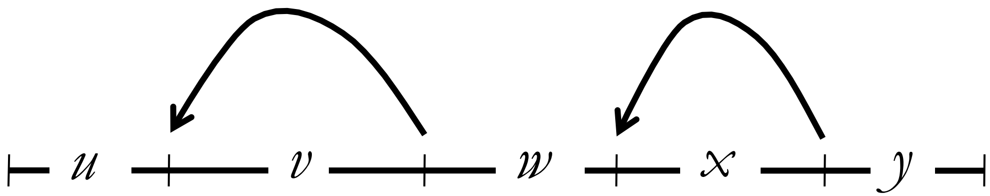
③ For every $i \geq 0$, the string uv^iwx^iy is in L.



Pumping lemma for context-free languages

- So to prove L is not context-free, it is enough that

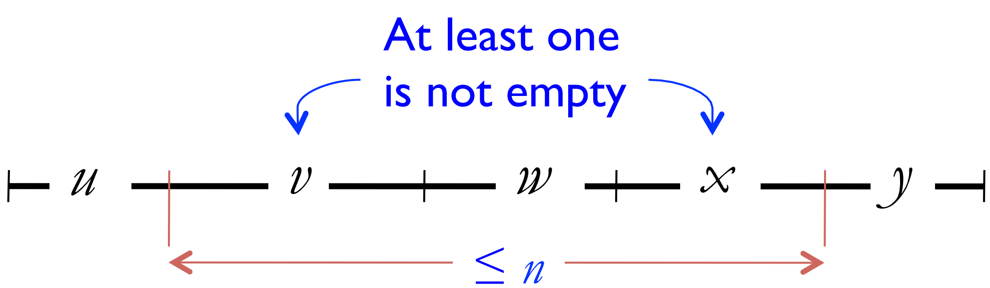
For all n there exists z in L longer than n, such that for all ways of writing $z = uvwxy$ where ① $|vwx| \leq n$ and ② $|vx| \geq 1$, the string uv^iwx^iy is not in L for some $i \geq 0$.



Proving a language is not context-free

- Just like for regular languages, Eve needs a strategy that wins her this game no matter what Adam chooses

	Adam	Eve
1	chooses n	chooses $z \in L$ $(\lvert z \rvert > n)$
2	writes $z = uvwxy$ $(\lvert vwx \rvert \leq n, \lvert vx \rvert \geq 1)$	chooses i
		Eve wins if $uv^iwx^iy \notin L$



Example

	Adam	Eve
1	chooses n	chooses $z \in L$ $(\|z\| > n)$
2	writes $z = uvwxy$ $(\|vwx\| \leq n, \|vx\| \geq 1)$	chooses i
		Eve wins if $uv^iwx^iy \notin L$

$$L_3 = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n\colon n \geq 0\}$$

1	chooses n	$z = \mathrm{a}^n\mathrm{b}^n\mathrm{c}^n$
2	writes $z = uvwxy$	$i = ?$

a a a … a a b b b … b b c c c … c c

$\overleftarrow{\;u\;}\!\!\rightarrow\leftarrow\!\! v \!\!\rightarrow\leftarrow\!\! w \!\!\rightarrow\leftarrow\!\! x \!\!\rightarrow\leftarrow\!\! y \!\!\rightarrow$

Example

- Case 1: v or x contains two kinds of symbols

$$\mathtt{a\ a\ a \dots a\ a\ b\ b\ b \dots b\ b\ c\ c\ c \dots c\ c}$$

$$\longleftarrow v \longrightarrow \qquad \qquad \longleftarrow x \longrightarrow$$

Then uv^2wx^2y not in L_3 because pattern is wrong

- Case 2: v and x both contain one kind of symbol

$$\mathtt{a\ a\ a \dots a\ a\ b\ b\ b \dots b\ b\ c\ c\ c \dots c\ c}$$

$$\longleftarrow v \longrightarrow \qquad \qquad \longleftarrow x \longrightarrow$$

Then uv^2wx^2y does not have same number of as, bs, cs

More examples

$L_1 = \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$ ✔

$L_2 = \{s : s$ has same number of $\mathtt{a}$s and $\mathtt{b}$s$\}$ ✔

$L_3 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$ ✘

$L_4 = \{ss^R : s \in \{\mathtt{a}, \mathtt{b}\}^*\}$ ✔

$L_5 = \{ss : s \in \{\mathtt{a}, \mathtt{b}\}^*\}$

Which is context-free?

Example

$$L_5 = \{ss: s \in \{\texttt{a}, \texttt{b}\}^*\}$$

1	chooses n	$z = \texttt{a}^n\texttt{ba}^n\texttt{b}$
2	writes $z = uvwxy$	$i = ?$

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

$\leftarrow u \rightarrow\leftarrow v \rightarrow\leftarrow w \rightarrow\leftarrow x \rightarrow\leftarrow y \rightarrow$

What if: | a a a a a a a a | a | | a | a a a a a a a a b |

$\leftarrow u \rightarrow$ w $\leftarrow y \rightarrow$

Example

$$L_5 = \{ss\colon s \in \{\mathtt{a}, \mathtt{b}\}^*\}$$

1	chooses n	$z = \mathtt{a}^n\mathtt{b}^n\mathtt{a}^n\mathtt{b}^n$
2	writes $z = uvwxy$	$i = ?$

$$\underbrace{\mathrm{a\,a\,a\,a\,a\,a}\,\mathrm{b}}_{u}\,\underbrace{\mathrm{b}}_{v}\,\underbrace{\mathrm{b\,b}}_{w}\,\underbrace{\mathrm{b}}_{x}\,\underbrace{\mathrm{b\,a\,a\,a\,a\,a\,a\,b\,b\,b\,b\,b\,b}}_{y}$$

Recall that $|vwx| \leq n$

Example

Three cases

Case 1: | a a a a a a b b b b b b | a a a a a a b b b b b b

$\leftarrow v \rightarrow\leftarrow w \rightarrow\leftarrow x \rightarrow$

vwx is in the first half of $\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n\mathtt{b}^n$

Case 2: a a a a a a | b b b b b b a a a a a a | b b b b b b

$\leftarrow v \rightarrow\leftarrow w \rightarrow\leftarrow x \rightarrow$

vwx is in the middle part of $\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n\mathtt{b}^n$

Case 3: a a a a a a b b b b b b | a a a a a a b b b b b b |

$\leftarrow v \rightarrow\leftarrow w \rightarrow\leftarrow x \rightarrow$

vwx is in the second half of $\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n\mathtt{b}^n$

Example

Apply pumping with $i = 0$

Case 1: $\big|\,\mathtt{a\ a\ a\ a\ a\ a\ b\ b\ b\ b\ b\ b}\,\big|\,\mathtt{a\ a\ a\ a\ a\ a\ b\ b\ b\ b\ b\ b}$

$\leftarrow v \rightarrow\leftarrow w \rightarrow\leftarrow x \rightarrow$

uwy looks like $\mathtt{a}^j\mathtt{b}^k\mathtt{a}^n\mathtt{b}^n$, where $j < n$ or $k < n$

Case 2: $\mathtt{a\ a\ a\ a\ a\ a}\,\big|\,\mathtt{b\ b\ b\ b\ b\ b\ a\ a\ a\ a\ a\ a}\,\big|\,\mathtt{b\ b\ b\ b\ b\ b}$

$\leftarrow v \rightarrow\leftarrow w \rightarrow\leftarrow x \rightarrow$

uwy looks like $\mathtt{a}^n\mathtt{b}^j\mathtt{a}^k\mathtt{b}^n$, where $j < n$ or $k < n$

Case 3: $\mathtt{a\ a\ a\ a\ a\ a\ b\ b\ b\ b\ b\ b}\,\big|\,\mathtt{a\ a\ a\ a\ a\ a\ b\ b\ b\ b\ b\ b}\,\big|$

$\leftarrow v \rightarrow\leftarrow w \rightarrow\leftarrow x \rightarrow$

uwy looks like $\mathtt{a}^n\mathtt{b}^n\mathtt{a}^j\mathtt{b}^k$, where $j < n$ or $k < n$

Example

Apply pumping with $i = 0$

$\boxed{L_5 = \{ss\text{: } s \in \{\mathtt{a}, \mathtt{b}\}^*\}}$

Case 1: uv^0wx^0y looks like $\mathtt{a}^j\mathtt{b}^k\mathtt{a}^n\mathtt{b}^n$, where $j < n$ or $k < n$

Not of the form ss

Case 2: uv^0wx^0y looks like $\mathtt{a}^n\mathtt{b}^j\mathtt{a}^k\mathtt{b}^n$, where $j < n$ or $k < n$

Not of the form ss

Case 3: uv^0wx^0y looks like $\mathtt{a}^n\mathtt{b}^n\mathtt{a}^j\mathtt{b}^k$, where $j < n$ or $k < n$

Not of the form ss

This covers all the cases, so L_5 is not context-free.

Which language is context-free?

$L_1 = \{\mathtt{a}^n\mathtt{b}^n\text{: } n \geq 0\}$ ✔

$L_2 = \{s\text{: } s \text{ has same number of } \mathtt{a}\text{s and } \mathtt{b}\text{s}\}$ ✔

$L_3 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\text{: } n \geq 0\}$ ✘

$L_4 = \{ss^R\text{: } s \in \{\mathtt{a}, \mathtt{b}\}^*\}$ ✔

$L_5 = \{ss\text{: } s \in \{\mathtt{a}, \mathtt{b}\}^*\}$ ✘

Properties
of
Context-Free languages

Union

Context-free languages

are closed under: **Union**

$$\left.\begin{array}{l} L_1 \text{ is context free} \\ \\ L_2 \text{ is context free} \end{array}\right\} \Rightarrow \begin{array}{c} L_1 \cup L_2 \\ \text{is context-free} \end{array}$$

Example

Language

Grammar

$$L_1 = \{a^n b^n\}$$

$$S_1 \to aS_1b \mid \varepsilon$$

$$L_2 = \{ww^R\}$$

$$S_2 \to aS_2a \mid bS_2b \mid \varepsilon$$

Union

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

$$S \to S_1 \mid S_2$$

In general:

For context-free languages $L_1, \ L_2$

with context-free grammars $G_1, \ G_2$

and start variables $S_1, \ S_2$

The grammar of the **union** $L_1 \cup L_2$

has new start variable S

and additional production $S \to S_1 \mid S_2$

Concatenation

Context-free languages
are closed under: **Concatenation**

L_1 is context free

L_2 is context free

$\Rightarrow$ $L_1 L_2$ is context-free

Example

Language	Grammar
$L_1 = \{a^n b^n\}$	$S_1 \to aS_1b \mid \varepsilon$
$L_2 = \{ww^R\}$	$S_2 \to aS_2a \mid bS_2b \mid \varepsilon$

Concatenation

$L = \{a^n b^n\}\{ww^R\}$ $\qquad$ $S \to S_1 S_2$

In general:

For context-free languages $L_1, \ L_2$

with context-free grammars $G_1, \ G_2$

and start variables $S_1, \ S_2$

The grammar of the **concatenation** L_1L_2

has new start variable S

and additional production $S \to S_1S_2$

Star Operation

Context-free languages are closed under: **Star-operation**

L is context free $\Rightarrow$ L^* is context-free

Example

Language

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

Grammar

$$S \rightarrow aSb \mid \varepsilon$$

Star Operation

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

$$S_1 \rightarrow SS_1 \mid \varepsilon$$

In general:

For context-free language L

with context-free grammar G

and start variable S

The grammar of the **star operation** L^*

has new start variable S_1

and additional production $S_1 \to SS_1 \mid \varepsilon$

Negative Properties
of
Context-Free Languages

Intersection

Context-free languages are **not** closed under: **intersection**

L_1 is context free

L_2 is context free

$\Rightarrow$ $L_1 \cap L_2$ **not** necessarily context-free

Example

$$L_1 = \{a^n b^n c^*\}$$

Context-free:

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid \varepsilon$$

$$C \rightarrow cC \mid \varepsilon$$

$$L_2 = \{a^* b^m c^m\}$$

Context-free:

$$S \rightarrow AB$$

$$A \rightarrow aA \mid \varepsilon$$

$$B \rightarrow bBc \mid \varepsilon$$

Intersection

$L_1 \cap L_2 = \{a^n b^n c^n\}$ **NOT** context-free

Complement

Context-free languages
are **not** closed under: **complement**

L is context free $\Rightarrow$ $\overline{L}$ **not** necessarily context-free

Example

$$L_1 = \{a^n b^n c^m\}$$

Context-free:

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid \varepsilon$$

$$C \rightarrow cC \mid \varepsilon$$

$$L_2 = \{a^n b^m c^m\}$$

Context-free:

$$S \rightarrow AB$$

$$A \rightarrow aA \mid \varepsilon$$

$$B \rightarrow bBc \mid \varepsilon$$

Complement

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2 = \{a^n b^n c^n\}$$

NOT context-free

Intersection
of
Context-free languages
and
Regular Languages

The intersection of

a context-free language and

a regular language

is a context-free language

$$\left.\begin{array}{ll} L_1 & \text{context free} \\ \\ L_2 & \text{regular} \end{array}\right\} \Rightarrow \begin{array}{c} L_1 \cap L_2 \\ \text{context-free} \end{array}$$

Machine M_1

PDA for L_1

context-free

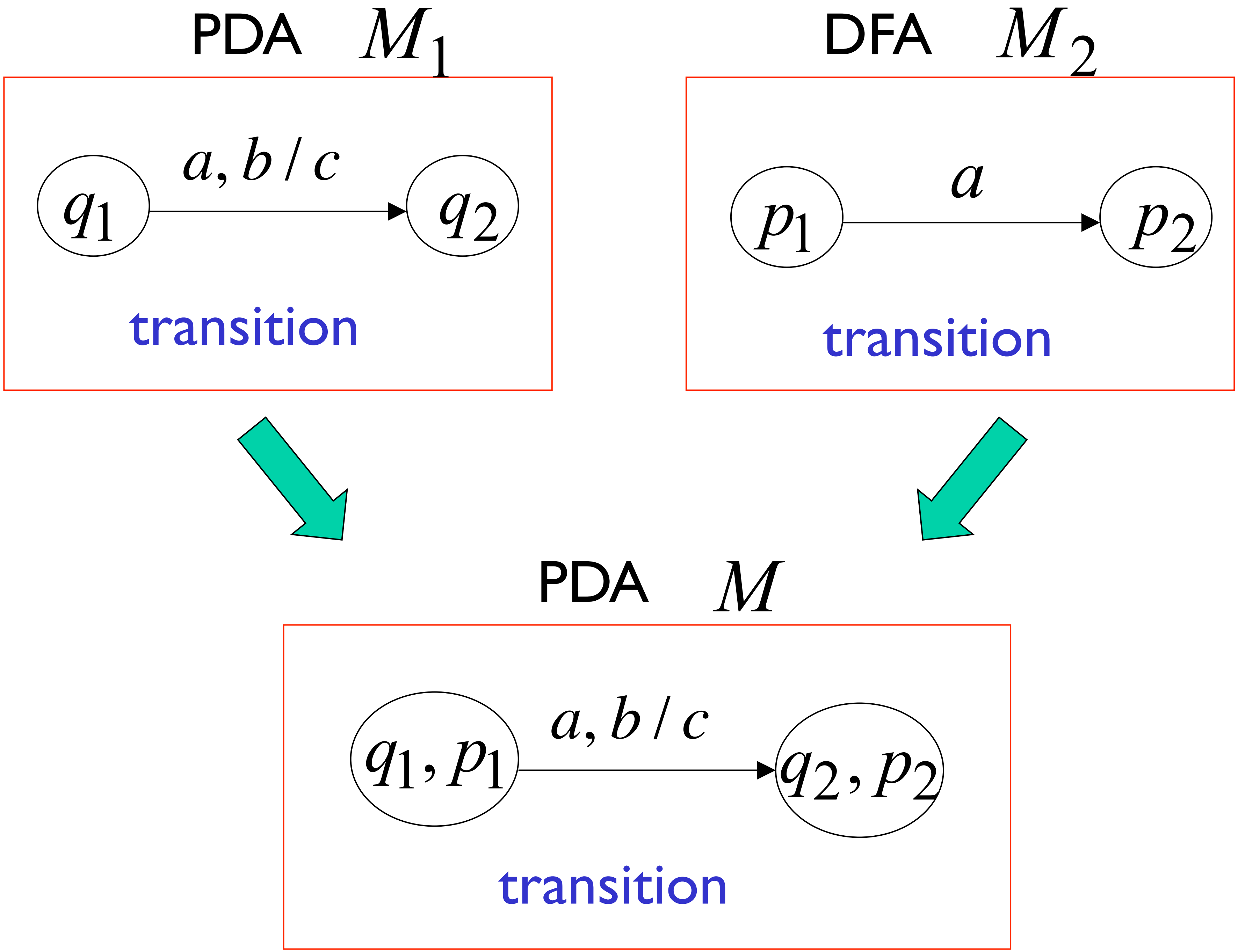
Machine M_2

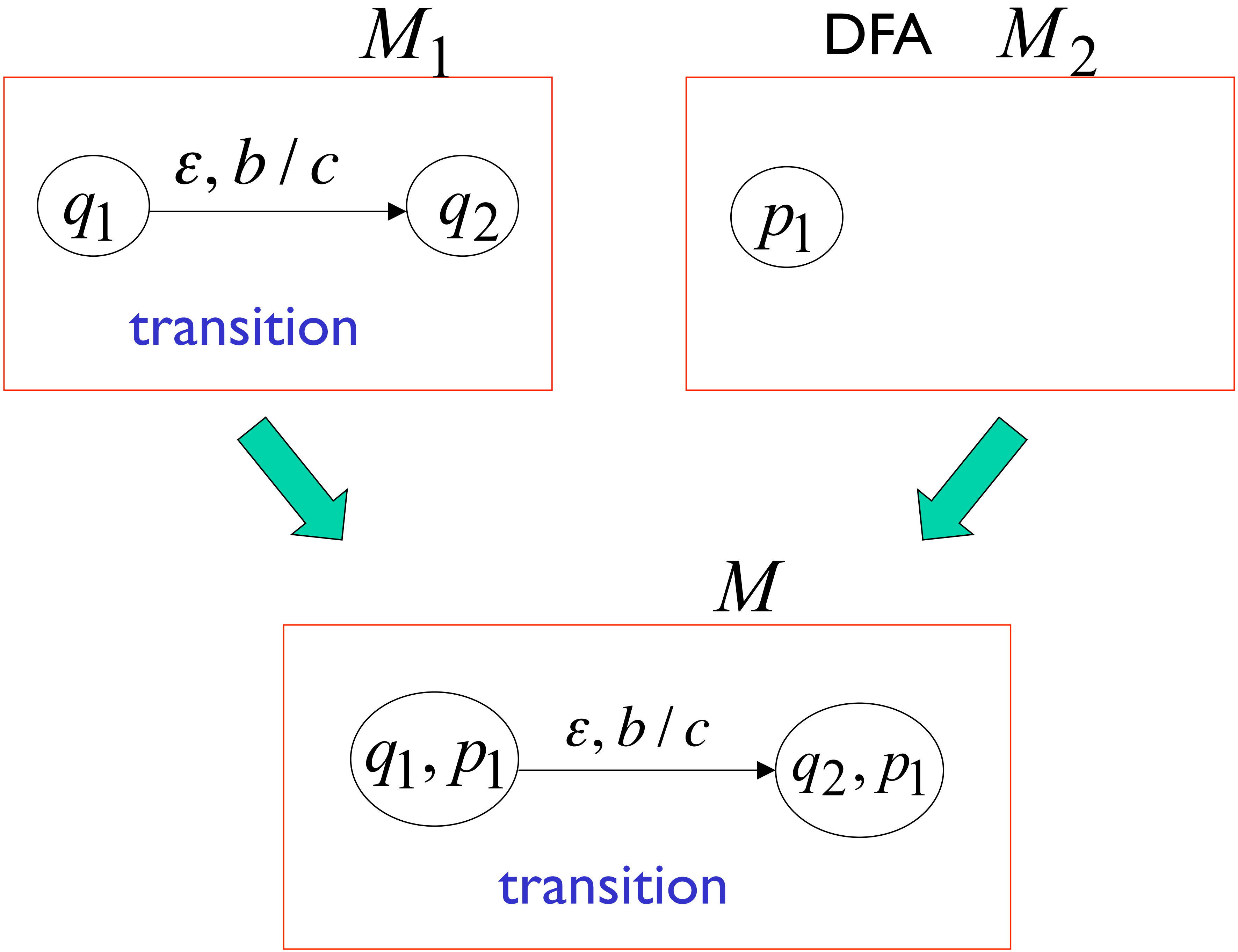
DFA for L_2

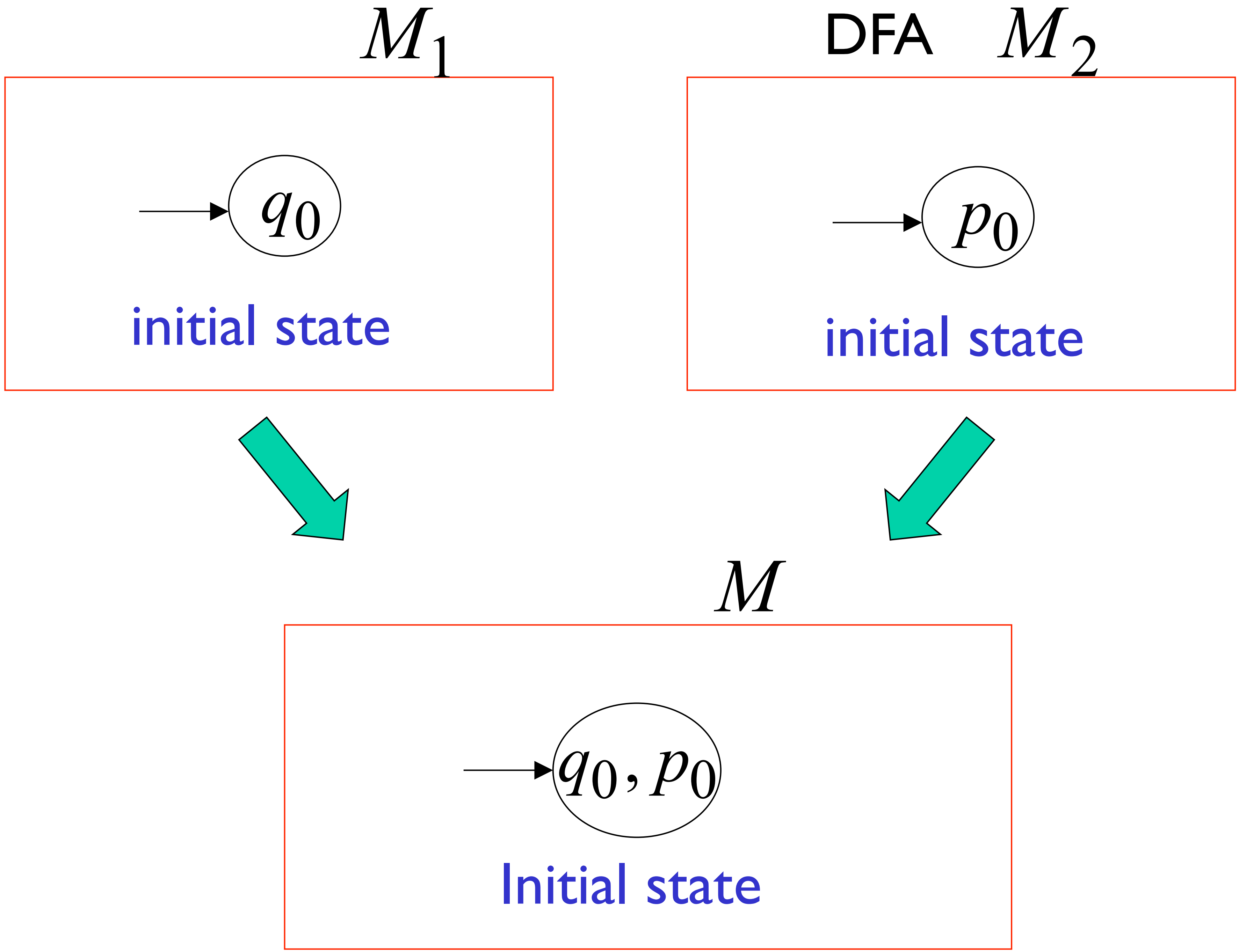
regular

Construct a new PDA M
that accepts $L_1 \cap L_2$

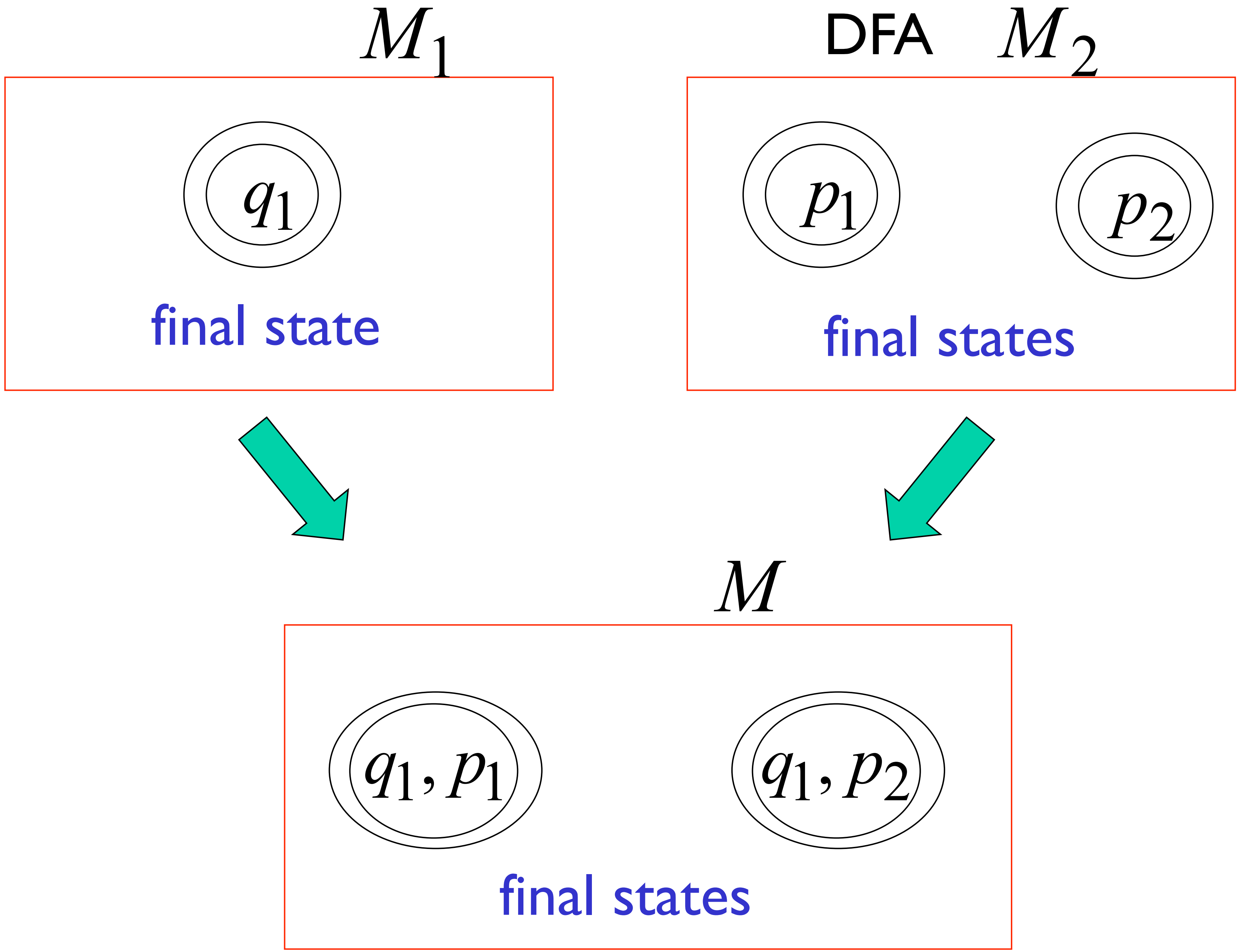
M will simulate in parallel M_1 and M_2


PDA M_1
$a, b / c$
q_1
q_2
transition
DFA M_2
a
p_1
p_2
transition
PDA M
$a, b / c$
q_1, p_1
q_2, p_2
transition


M_1
DFA M_2
$\varepsilon, b / c$
q_1
q_2
transition
p_1
M
q_1, p_1
$\varepsilon, b / c$
q_2, p_1
transition



M_1
DFA M_2
q_0
initial state
p_0
initial state
M
q_0, p_0
Initial state



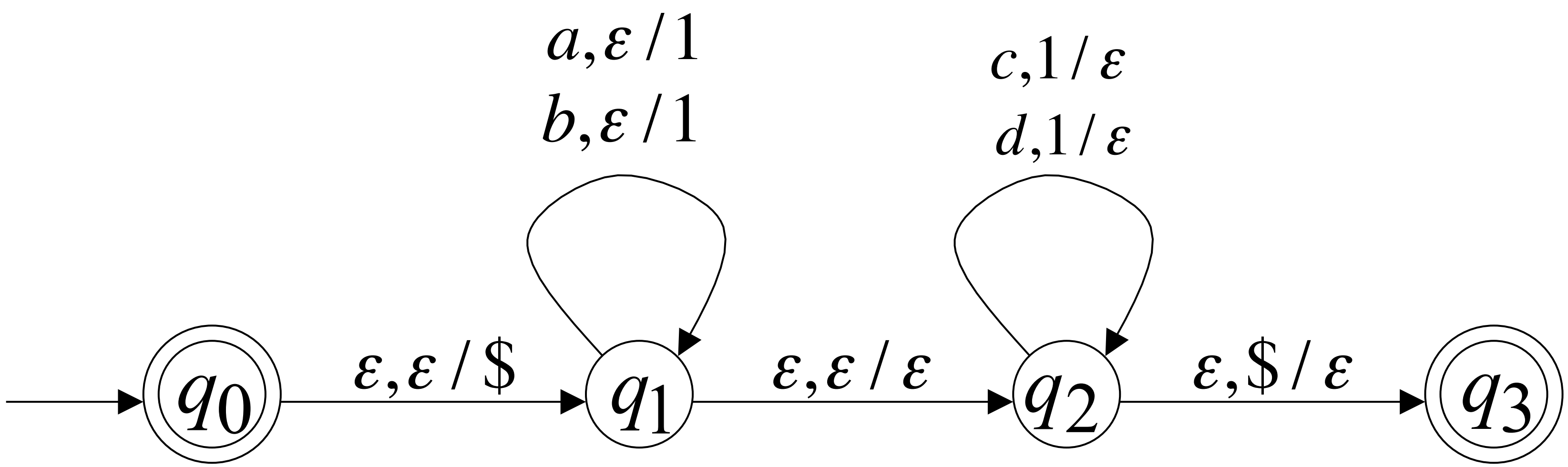
M_1
q_1
final state
DFA M_2
p_1
p_2
final states
M
q_1, p_1
q_1, p_2
final states

Example:

context-free

$$L_1 = \{w_1 w_2 : |w_1| = |w_2|, w_1 \in \{a,b\}^*, w_2 \in \{c,d\}^*\}$$

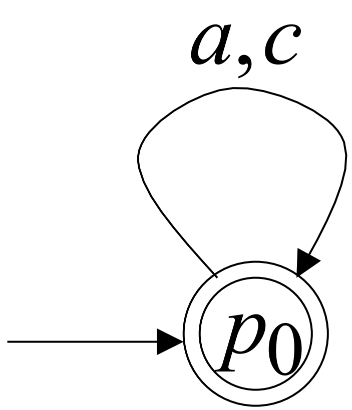
NPDA M_1



regular

$$L_2 = \{a, c\}^*$$

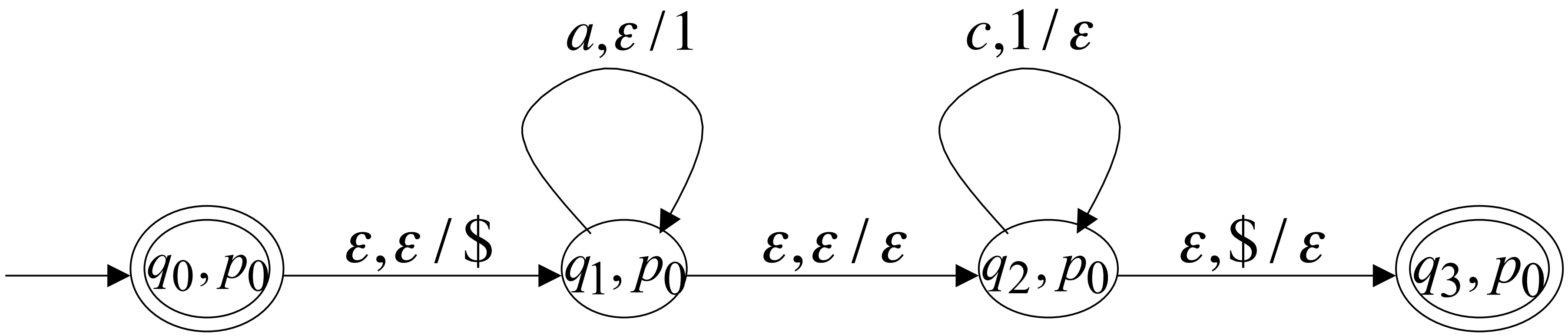
DFA M_2



context-free

Automaton for: $L_1 \cap L_2 = \{a^n c^n : n \geq 0\}$

PDA M



In General:

M simulates in parallel M_1 and M_2

M accepts string w if and only if

M_1 accepts string w and

M_2 accepts string w , i.e.

$$L(M) = L_1 \cap L_2$$

As M is a PDA, then

$L(M) = L_1 \cap L_2$ is context-free.

Applications
of
Regular Closure

The intersection of

a context-free language and

a regular language

is a context-free language

Regular Closure

L_1 context free

L_2 regular

$\Rightarrow$ $L_1 \cap L_2$

context-free

An Application of Regular Closure

Prove that: $L = \{a^n b^n : \ n \neq 100, n \geq 0\}$

is context-free

We know:

$\{a^n b^n : n \geq 0\}$ is context-free

We also know:

$$L_1 = \{a^{100}b^{100}\}$$ is regular

$$\overline{L_1} = \{(a+b)^*\} - \{a^{100}b^{100}\}$$ is regular

$$\{a^n b^n\}$$

context-free

$$\overline{L_1} = \{(a+b)^*\} - \{a^{100} b^{100}\}$$

regular

(regular closure) $\{a^n b^n\} \cap \overline{L_1}$ context-free

$$\{a^n b^n\} \cap \overline{L_1} = \{a^n b^n : \quad n \neq 100, n \geq 0\} = L$$

is context-free

Another Application of Regular Closure

Prove that: $L = \{w: \ \ n_a = n_b = n_c\}$

is **not** context-free

If $L = \{w: \ n_a = n_b = n_c\}$ is context-free

(regular closure)

Then $L \cap \{a^*b^*c^*\} = \{a^n b^n c^n\}$

context-free   regular   context-free

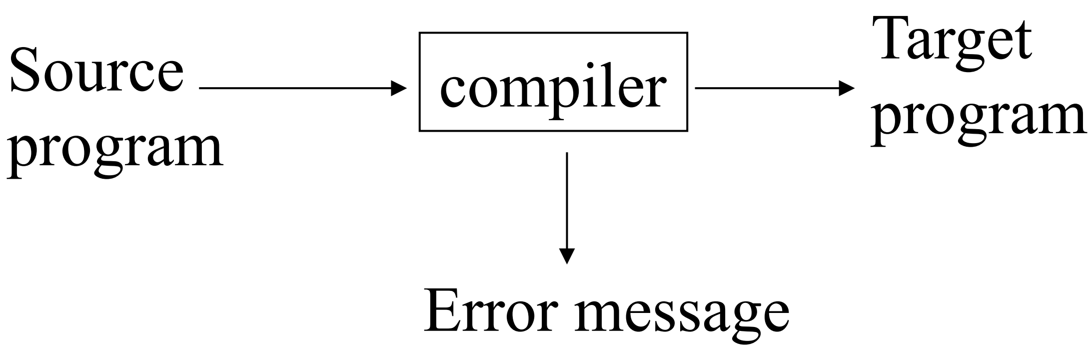
Impossible!!!

Therefore, L is **not** context free

LR(0) grammars

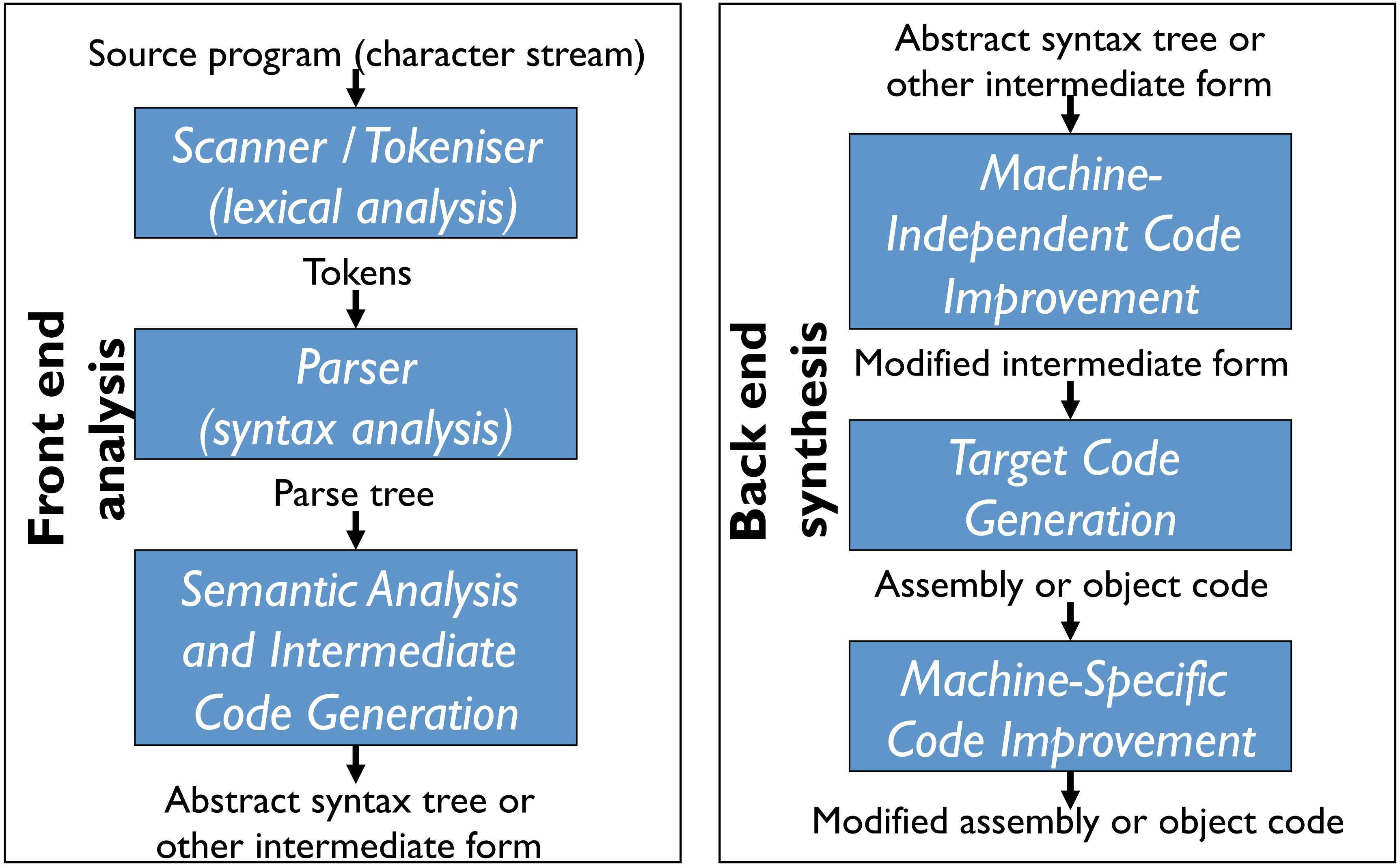
Compiler

A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language).



Typically the source language is a high-level language and the target language is a low-level language (machine code).

Compiler Front-end and Back-end



Tokenising computer programs

```
if (n == 0) { return x; }
```

- First the `javac` compiler does a lexical analysis:

```
if (ID == INT_LIT) { return ID; }
```

`ID` = identifier (name of variable, procedure, class, ...)

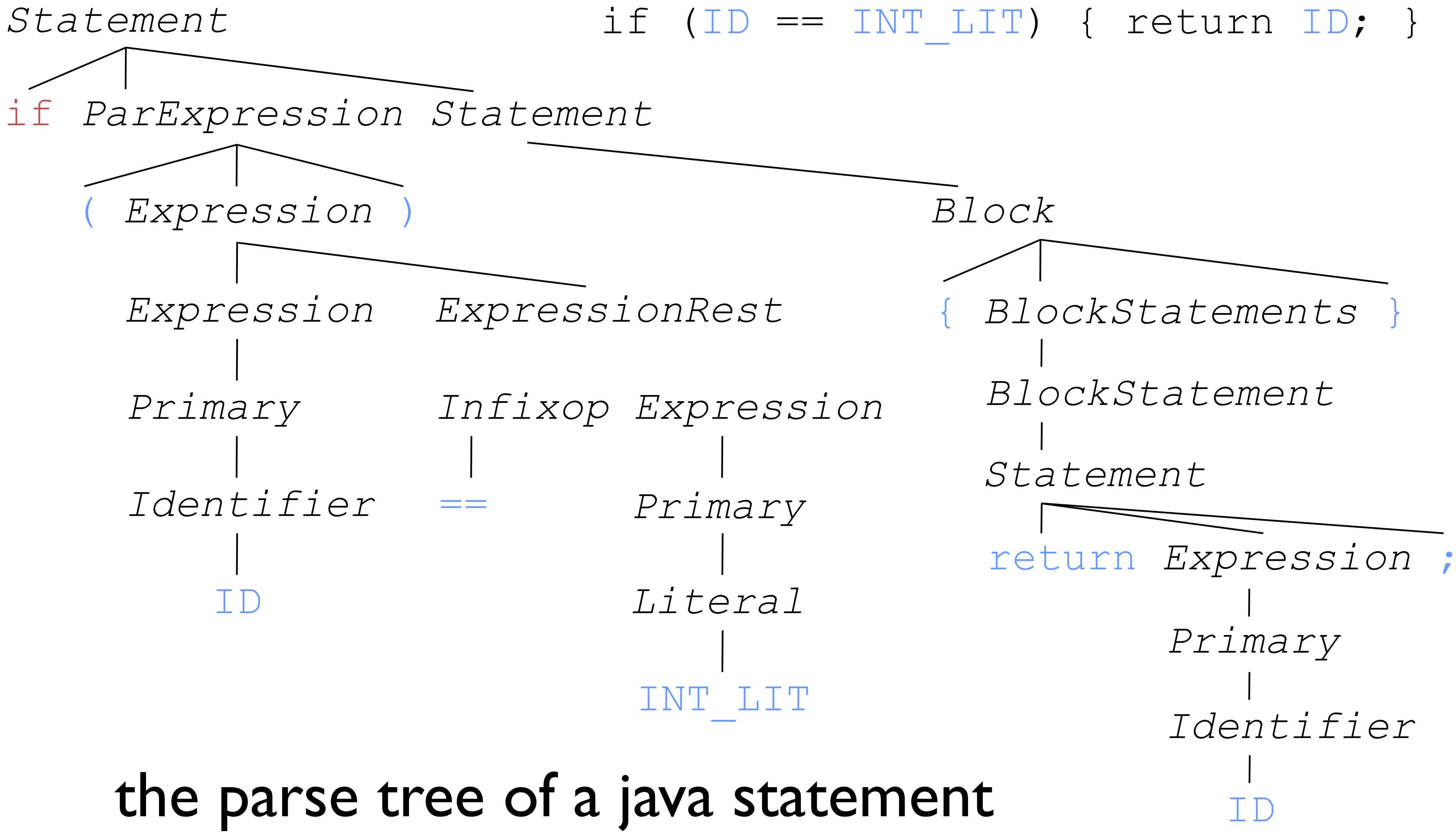
`INT_LIT` = integer literal (value)

- The alphabet of java CFG consists of symbols like:

Σ = {`if`, `return`, `(`, `)` `{`, `}`, `;`, `==`, `ID`, `INT_LIT`, ...}

Parsing computer programs

```
if (n == 0) { return x; }
if (ID == INT_LIT) { return ID; }
```



the parse tree of a java statement

CFG of the java programming language

```
Identifier:
            ID

QualifiedIdentifier:
            Identifier { . Identifier }

Literal:
            IntegerLiteral
            FloatingPointLiteral
            CharacterLiteral
            StringLiteral
            BooleanLiteral
            NullLiteral

Expression:
            Expression1 [AssignmentOperator Expression1]]

AssignmentOperator:
            =
            +=
            -=
            *=
            /=
            &=
            |=
```

...

from `http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html#52996`

Parsing java programs

```
class Point2d {
    /* The X and Y coordinates of the point--instance variables */
    private double x;
    private double y;
    private boolean debug;                      // A trick to help with debugging

    public Point2d (double px, double py) {     // Constructor
            x = px;
            y = py;

            debug = false;                       // turn off debugging
    }

    public Point2d () {             // Default constructor
            this (0.0, 0.0);                    // Invokes 2 parameter Point2D constructor
    }
    // Note that a this() invocation must be the BEGINNING of
    // statement body of constructor

    public Point2d (Point2d pt) {               // Another consructor
            x = pt.getX();
            y = pt.getY();
    }
    ...
}
```

Simple java program: about 500 symbols

Parsing algorithms

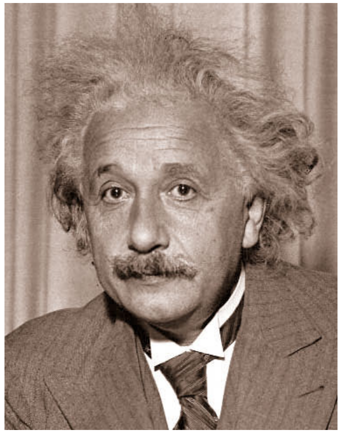
- How long would it take to parse this program?

try all parse trees	about 10^{80} years
CYK algorithm	about 1 week!

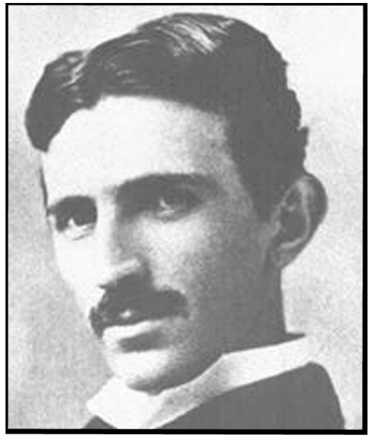
- Can we parse faster?
- No! CYK $O(|G|n^3)$ is essentially the fastest known general-purpose parsing algorithm for CFGs

Remark: Coppersmith–Winograd algorithm for multiplying matrices gives an asymptotic worst-case running time $O(n^{2.38} \cdot |G|)$. However, the hidden constant is so high that it is non-practical for present-day computers (Knuth 1997).

Another way of thinking

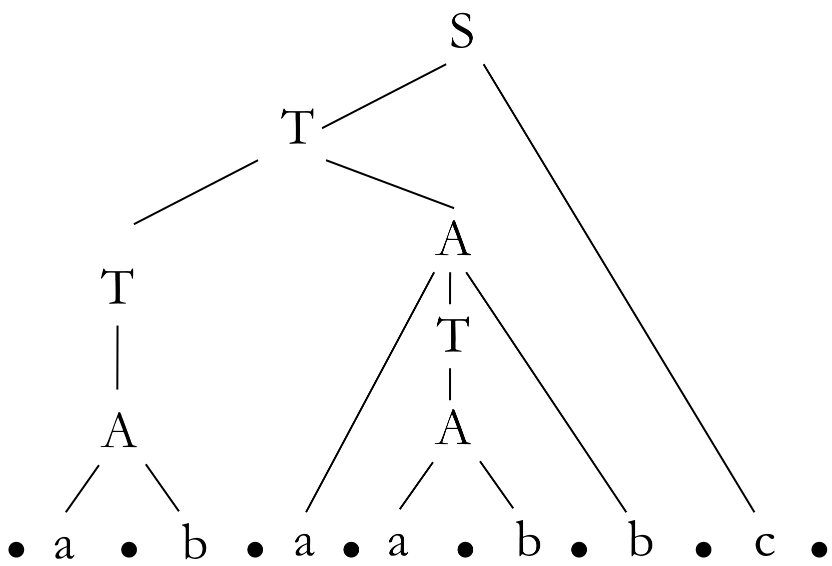


Scientist:
Find an algorithm that
can parse any CFG



Engineer:
Design your CFG
so it can be parsed
very quickly

Left-to-right parsing



$S \rightarrow Tc^{(1)}$
$T \rightarrow TA^{(2)} \mid A^{(3)}$
$A \rightarrow aTb^{(4)} \mid ab^{(5)}$

input: abaabbc

Try to match to the left of •

Items

$S \rightarrow Tc^{(1)}$	$T \rightarrow TA^{(2)}$	$T \rightarrow A^{(3)}$	$A \rightarrow aTb^{(4)}$	$A \rightarrow ab^{(5)}$
$S \rightarrow \bullet Tc$	$T \rightarrow \bullet TA$	$T \rightarrow \bullet A$	$A \rightarrow \bullet aTb$	$A \rightarrow \bullet ab$
$S \rightarrow T \bullet c$	$T \rightarrow T \bullet A$	$T \rightarrow A \bullet$	$A \rightarrow a \bullet Tb$	$A \rightarrow a \bullet b$
$S \rightarrow Tc \bullet$	$T \rightarrow TA \bullet$		$A \rightarrow aT \bullet b$	$A \rightarrow ab \bullet$
			$A \rightarrow aTb \bullet$	

- An item is a production augmented with a •
- The item is complete if the • is the last symbol

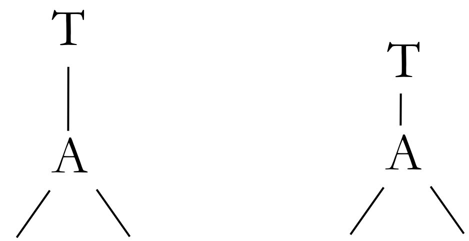
Meaning of items

$$\begin{aligned} S &\rightarrow Tc^{(1)} \\ T &\rightarrow TA^{(2)} \mid A^{(3)} \\ A &\rightarrow aTb^{(4)} \mid ab^{(5)} \end{aligned}$$

$A \rightarrow a \bullet Tb$

$A \rightarrow a \bullet b$

a • b a a b b c



$A \rightarrow aT \bullet b$

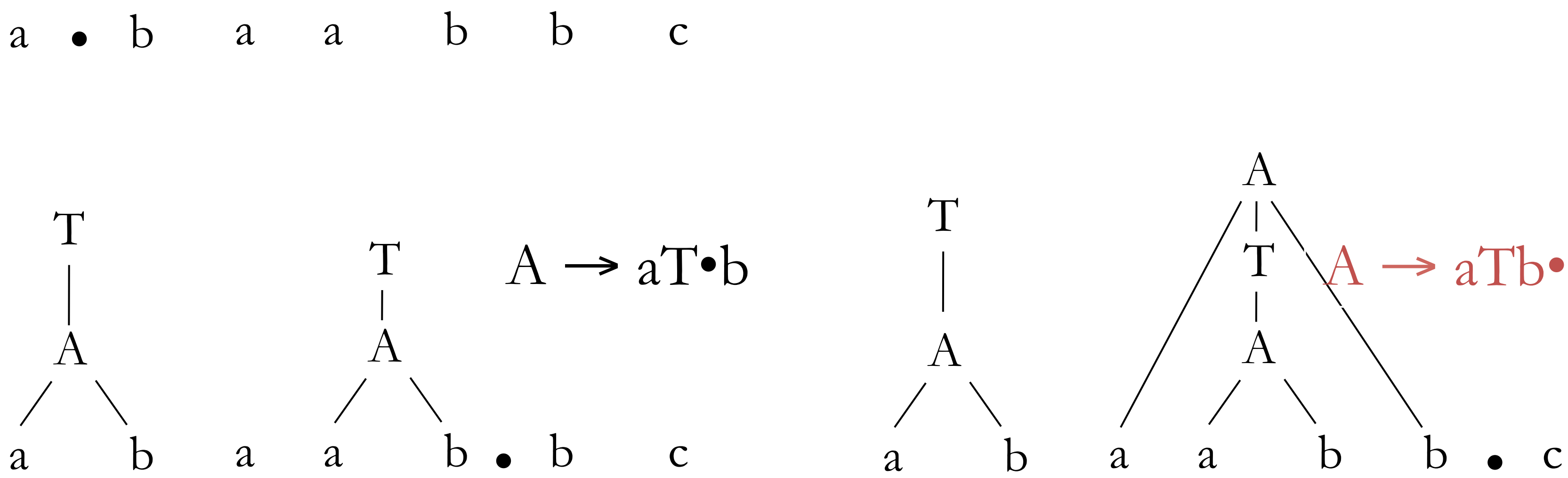
a b a a b • b c

Items represent possibilities
at various stages of the parsing process

Meaning of items

$S \rightarrow Tc^{(1)}$
$T \rightarrow TA^{(2)} \mid A^{(3)}$
$A \rightarrow aTb^{(4)} \mid ab^{(5)}$

$A \rightarrow a\bullet Tb$
$A \rightarrow a\bullet b$



When a complete item occurs,
a part of the parse tree is discovered

LR(0) parsing

Move from left to right

Keep track of all possible valid items

Prune the invalid items

When a complete item occurs, build part of parse tree

$A \rightarrow aAb \mid ab$

valid items registry
$A \rightarrow \bullet aAb$ $\quad A \rightarrow \bullet ab$

$\bullet\, a \quad a \quad b \quad b$

LR(0) parsing

$$A \rightarrow aAb \mid ab$$

Move from left to right

Keep track of all possible valid items

Prune the invalid items

When a complete item occurs, build part of parse tree

valid items registry	
$A \rightarrow a\bullet Ab$	$A \rightarrow a\bullet b$
$A \rightarrow \bullet aAb$	$A \rightarrow \bullet ab$

$$a \bullet a \quad b \quad b$$

LR(0) parsing

$$A \rightarrow aAb \mid ab$$

Move from left to right

Keep track of all possible valid items

Prune the invalid items

When a complete item occurs, build part of parse tree

valid items registry	
$\cancel{A \rightarrow a\bullet Ab}$	$\cancel{A \rightarrow a\bullet b}$
$A \rightarrow a\bullet Ab$	$A \rightarrow a\bullet b$

$A \rightarrow a\bullet Ab$

$a \quad a \bullet b \quad b$

LR(0) parsing

$A \to aAb \mid ab$

Move from left to right

Keep track of all possible valid items

Prune the invalid items

When a complete item occurs, build part of parse tree

valid items registry	
$A \to a\bullet Ab$	$A \to a\bullet b$
$A \to \bullet aAb$	$A \to \bullet ab$

$A \to a\bullet Ab$

$a \quad a \;\bullet\; b \quad b$

LR(0) parsing

$A \longrightarrow aAb \mid ab$

Move from left to right

Keep track of all possible valid items

Prune the invalid items

When a complete item occurs, build part of parse tree

valid items registry

$\cancel{A \longrightarrow a\bullet Ab}$ $\quad A \longrightarrow ab\bullet$

$\cancel{A \longrightarrow \bullet aAb}$ $\quad \cancel{A \longrightarrow \bullet ab}$

$A \longrightarrow a\bullet Ab$

$a \quad a \quad b \bullet b$

LR(0) parsing

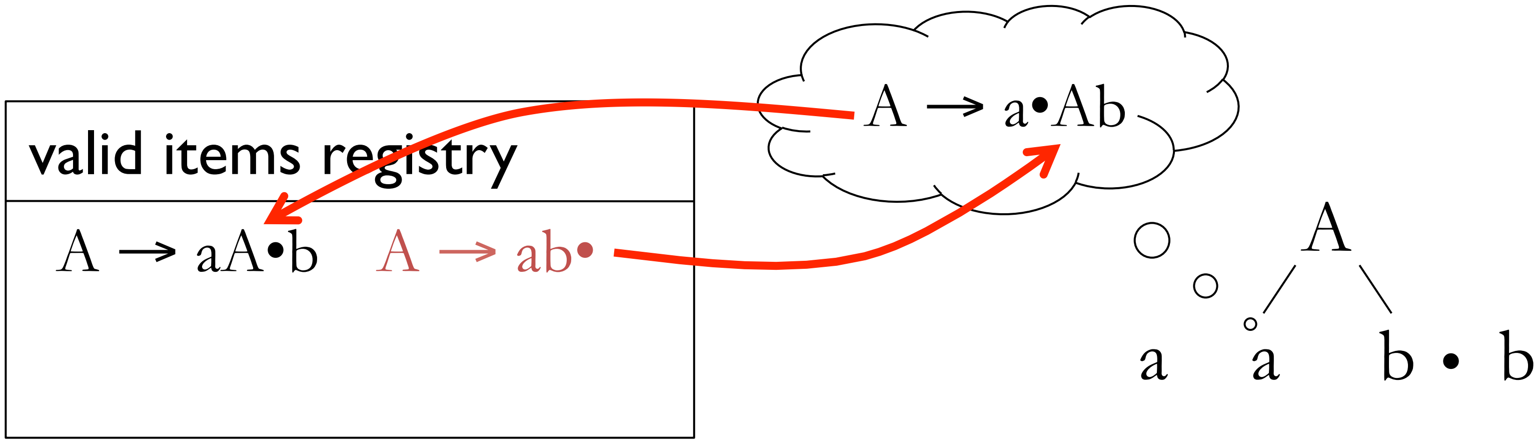
$A \to aAb \mid ab$

Move from left to right

Keep track of all possible valid items

Prune the invalid items

When a complete item occurs, build part of parse tree



LR(0) parsing

Move from left to right

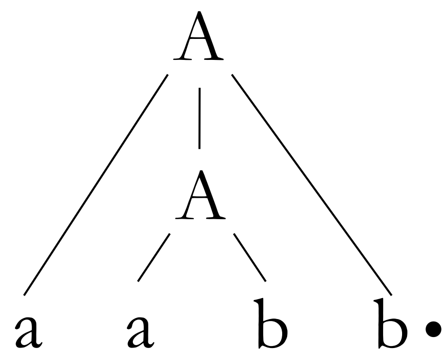
$A \to aAb \mid ab$

Keep track of all possible valid items

Prune the invalid items

When a complete item occurs, build part of parse tree

valid items registry
$A \to aAb\bullet$



two kinds of actions

valid items registry	
A → a•Ab	A → a•b
A → •aAb	A → •ab

a • a • b b

no complete item
in registry

shift

valid items registry
A → ab•

A

a a b • b

exactly one
complete item

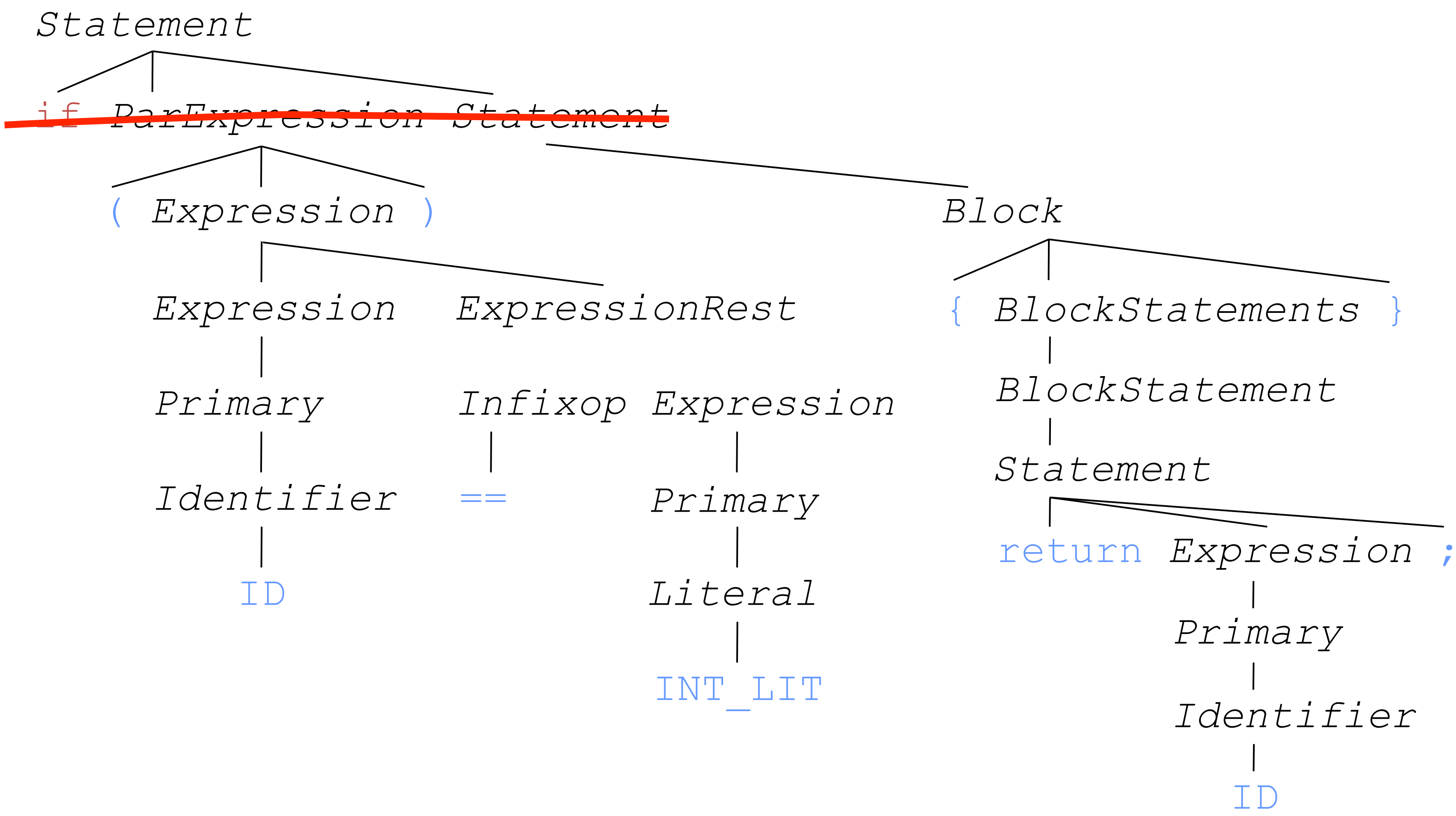
reduce

LR(0) grammars and deterministic PDAs

- The PDA for LR(0) parsing is deterministic
- Some context-free languages require non-deterministic PDAs, e.g. $L = \{ww^R : w \in \{\mathrm{a}, \mathrm{b}\}^*\}$
- The class of deterministic CFGs/PDAs is less expressive than CFGs/PDAs
- What can go wrong with LR(0) parsing?

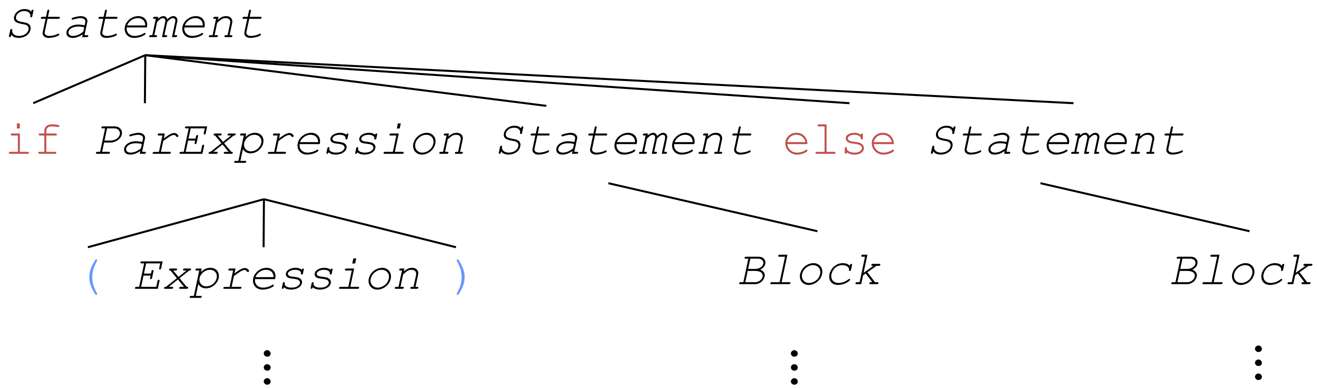
Parsing computer programs

```
if (n == 0) { return x; }
else { return x + 1; }
```



Parsing computer programs

```
if (n == 0) { return x; }
else { return x + 1; }
```



LR(0) parsers cannot tell apart

`if` … `then` **from** `if` … `then` … `else`

When you can't LR(0) parse

- LR(0) parser can perform two actions:

no complete item is valid → **shift (S)**

there is one valid item, and it is complete → **reduce (R)**

- What if:

some valid items complete, some not → **S / R conflict**

more than one valid complete item → **R / R conflict**

Dangling else

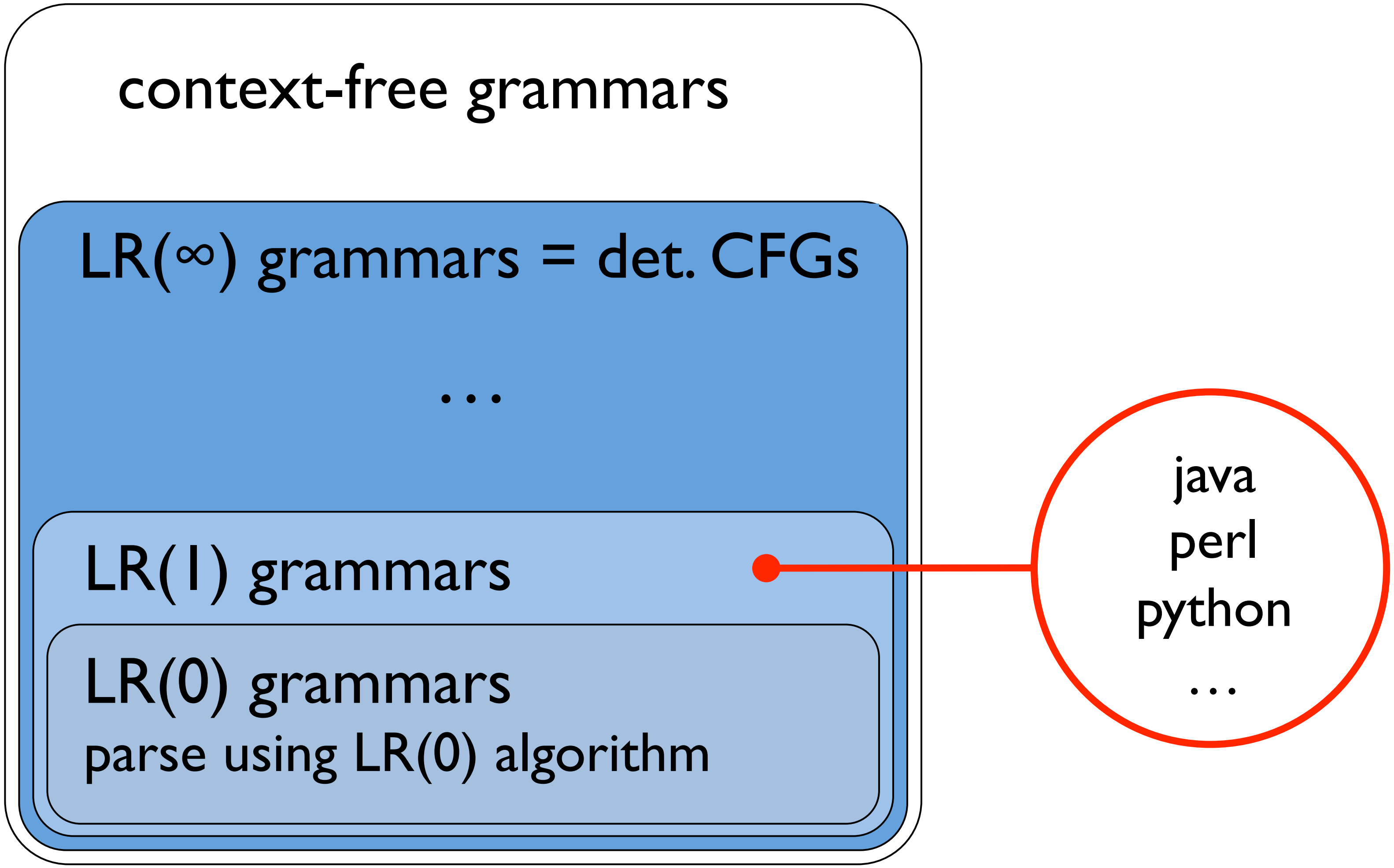
if a **then if** b **then** s **else** s2

Can be interpreted as

1. **if** a **then** (**if** b **then** s) **else** s2
2. **if** a **then** (**if** b **then** s **else** s2)

shift-reduce conflict

Hierarchy of context-free grammars



LR(k) = grammars that can be parsed like LR(0) but the decision whether to shift or reduce can be based on the next k symbols