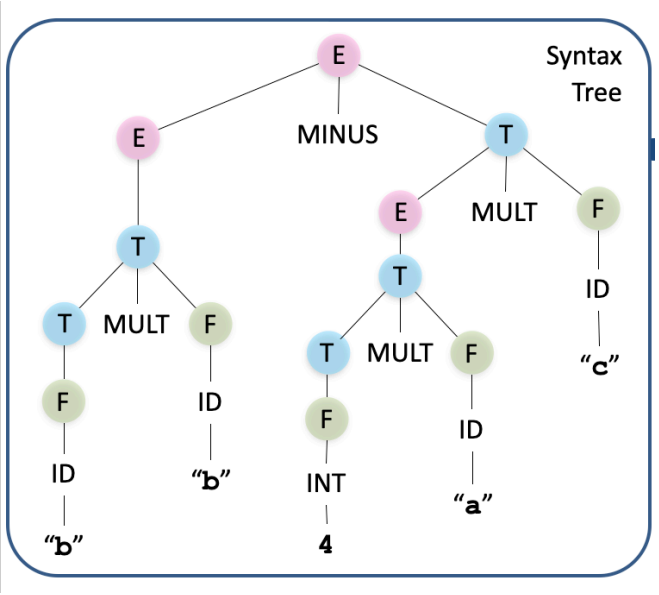
Compilation

0368-3133

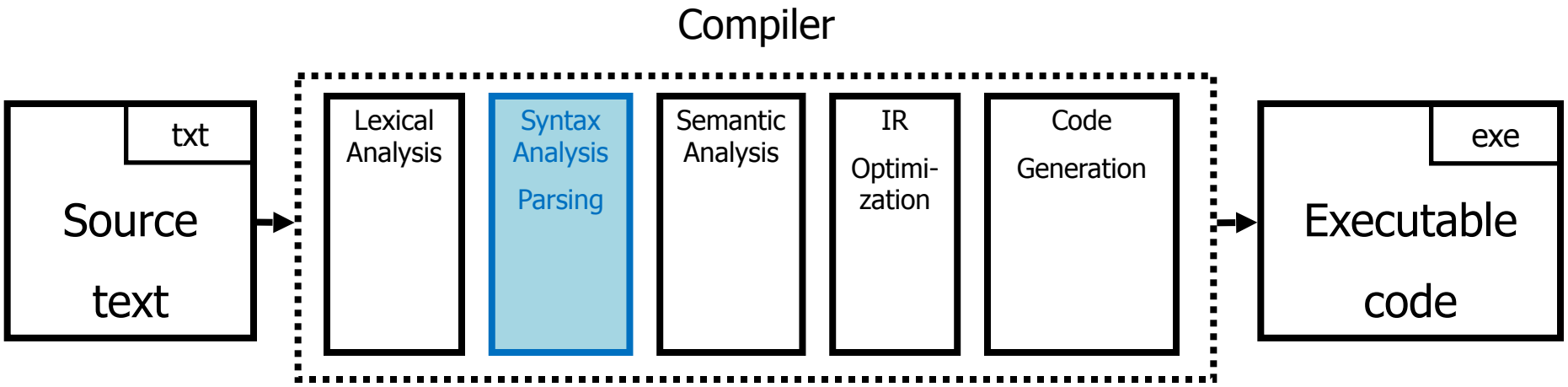
Lecture 2a:

Syntax Analysis: Foundations

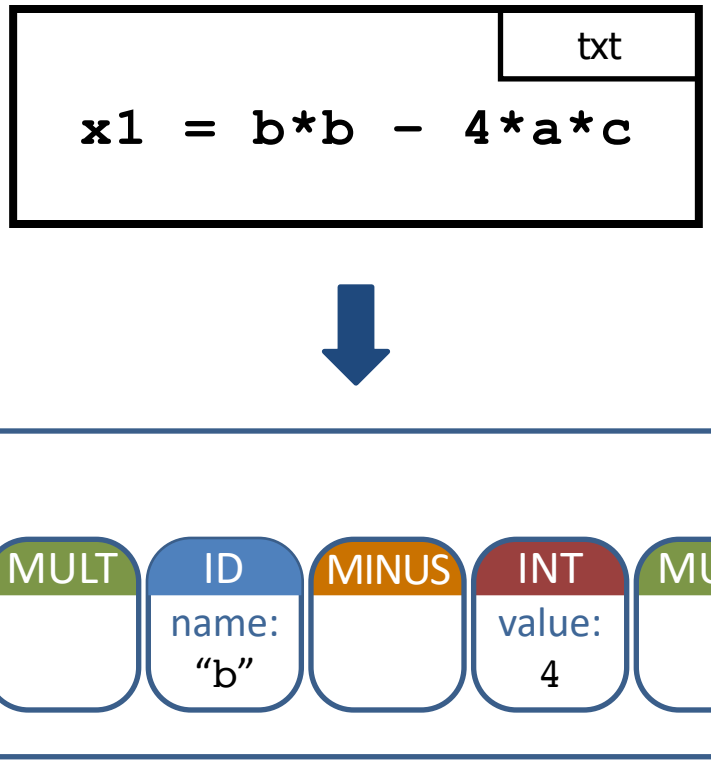


Slides credit: Shachar Itzhaky, Eran Yahav, Mooly Sagiv, Iftach Haitner

You are here

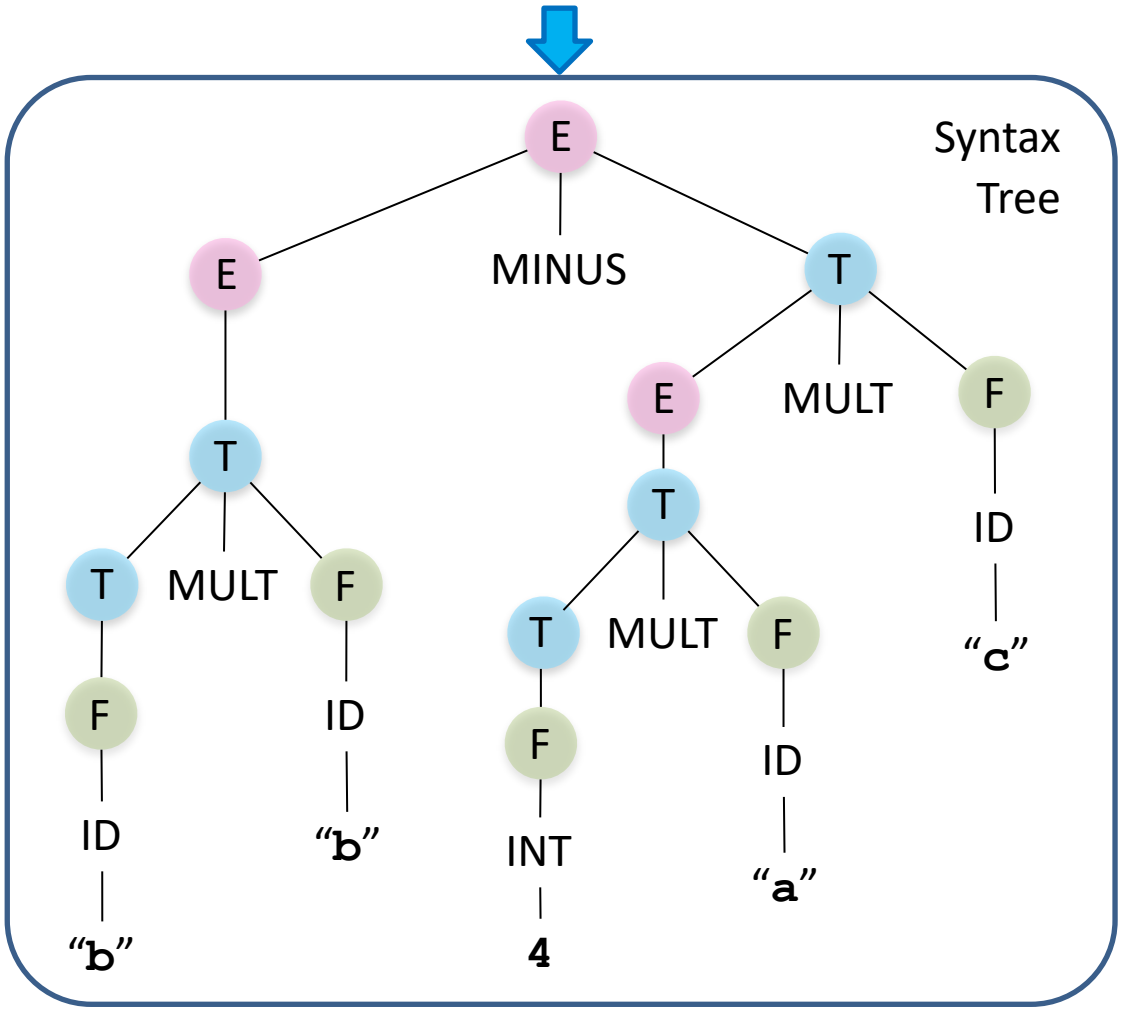

Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code
Generation
exe
Executable code

Previously: from Characters to Tokens



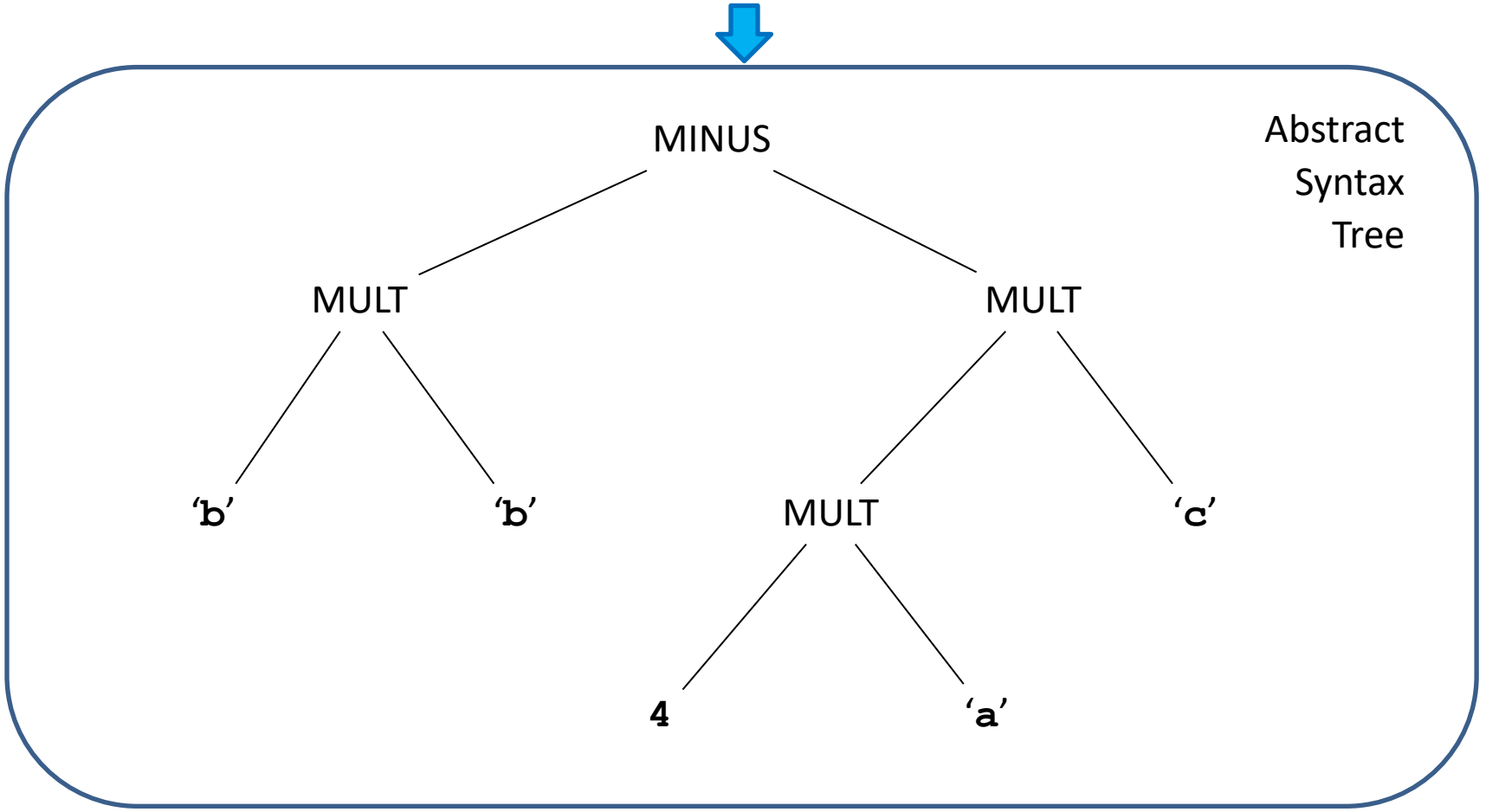
Now: from Tokens to Syntax Tree

⟨ID,"b"⟩ ⟨MULT⟩ ⟨ID,"b"⟩ ⟨MINUS⟩ ⟨INT,4⟩ ⟨MULT⟩ ⟨ID,"a"⟩ ⟨MULT⟩ ⟨ID,"c"⟩



Later On: from Tokens to AST

⟨ID,"**b**"⟩ ⟨MULT⟩ ⟨ID,"**b**"⟩ ⟨MINUS⟩ ⟨INT,**4**⟩ ⟨MULT⟩ ⟨ID,"**a**"⟩ ⟨MULT⟩ ⟨ID,"**c**"⟩



Parsing

- Goals
  - Decide whether the sequence of tokens a valid program in the source language
  - Construct a structured representation of the input
    - Parse tree ≈ “Diagramming” the input
  - Error detection and reporting
- Challenges
  - How do you describe the programming language?
  - How do you check validity of an input?
  - Where do you report an error?

Regular Languages to the Rescue?

- Regular expressions (RE)

Generate

~ language-defining power

- Nondeterministic finite state automata (NFA)

Recognize

~ language-defining power

- Deterministic finite state automata (DFA)

Algorithmically Recognize

Well, …

```
x = 0;
x = 1;
y = x + z;
…
```

But …

```
while (cond) {
   while (cond) {
      …
   }
}
```

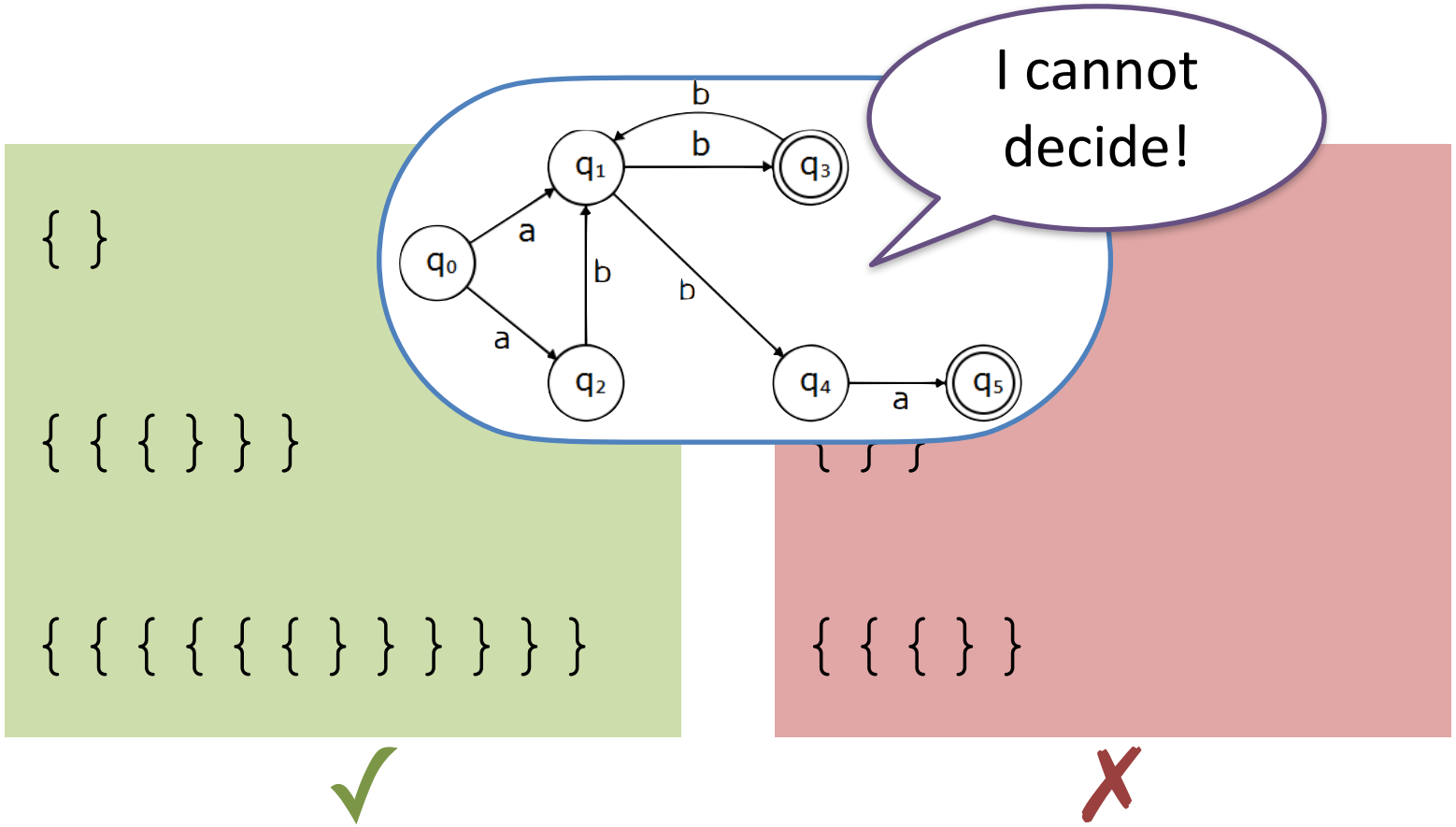
```
x = ((a + 3) * (…)) / 2
```

{{{{{{ … }}}}}}

((()(()())))

Context-Free Languages ⊃ Regular Languages

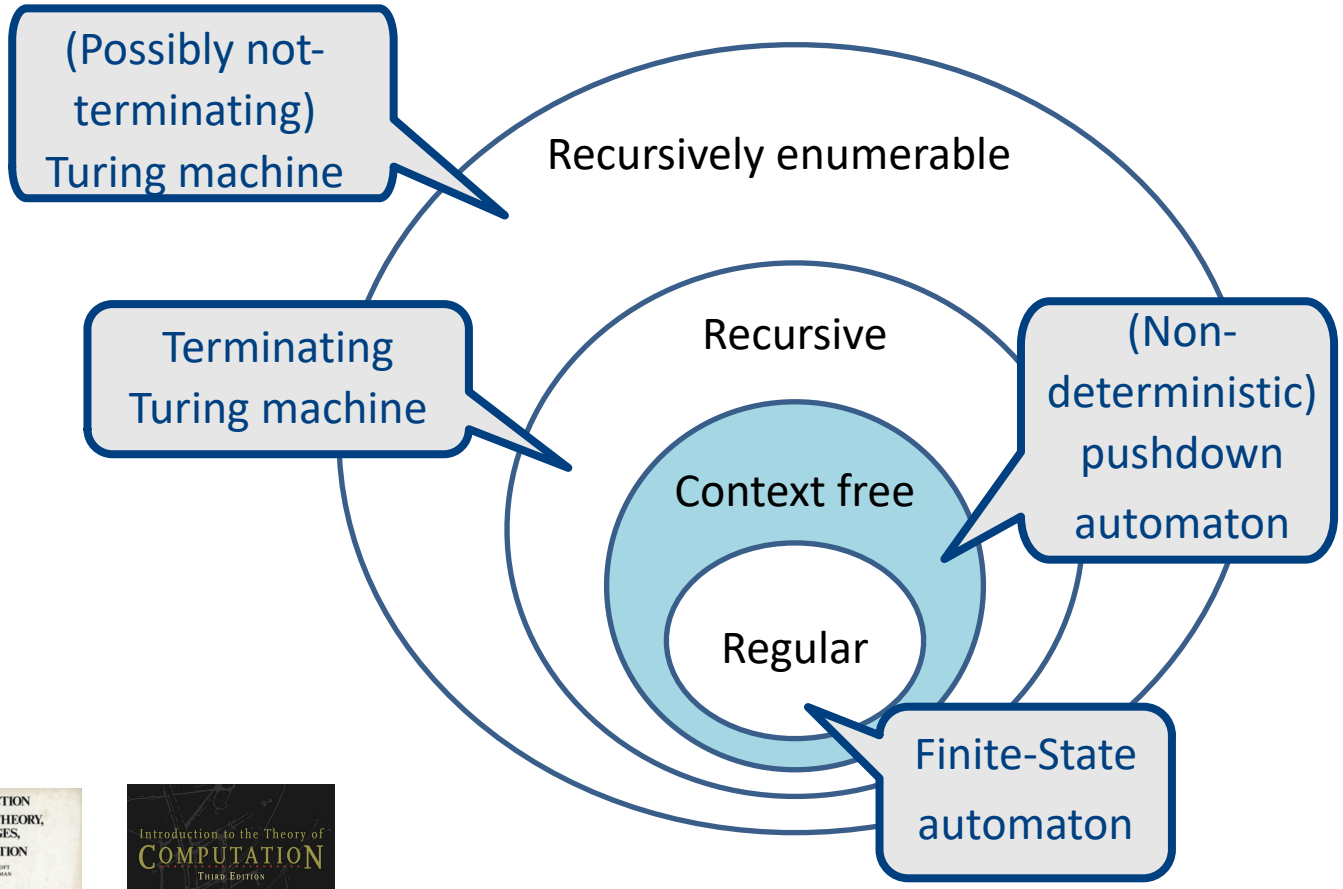
$L_{\{\}}$= language of nested curly brackets over Σ = {'{', '}'}

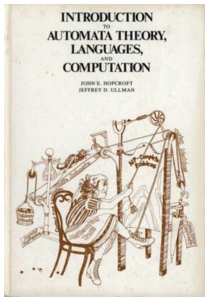


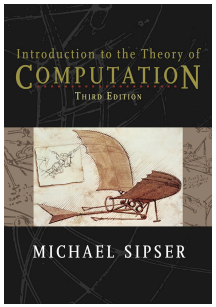
Context-Free Languages $\supset$ Regular Languages

- $L_{01} = \{ 0^n1^n \mid n \geq 0 \}$
  - E.g., 000111
- $L_{\text{even-palindrome}} = \{pp' \mid p \in \{a,b\}^*, p'=\text{reverse}(p)\}$
  - E.g., abbbba
- $L_{\text{palindrome\#}} = \{p\#p' \mid p \in \{a,b\}^*, p'=\text{reverse}(p)\}$
  - E.g., abbb#bbba
- $L_{\text{ballanced}} = \{pq, (p) \mid p,q \in \{ \varepsilon \} \cup L_{\text{ballanced}} \}$
  - E.g., ()(())
- $L_{\text{eqnum}} = \{p \in \{a, b\}^* \mid \#_a(p) = \#_b(p) \}$
  - E.g., abbaab

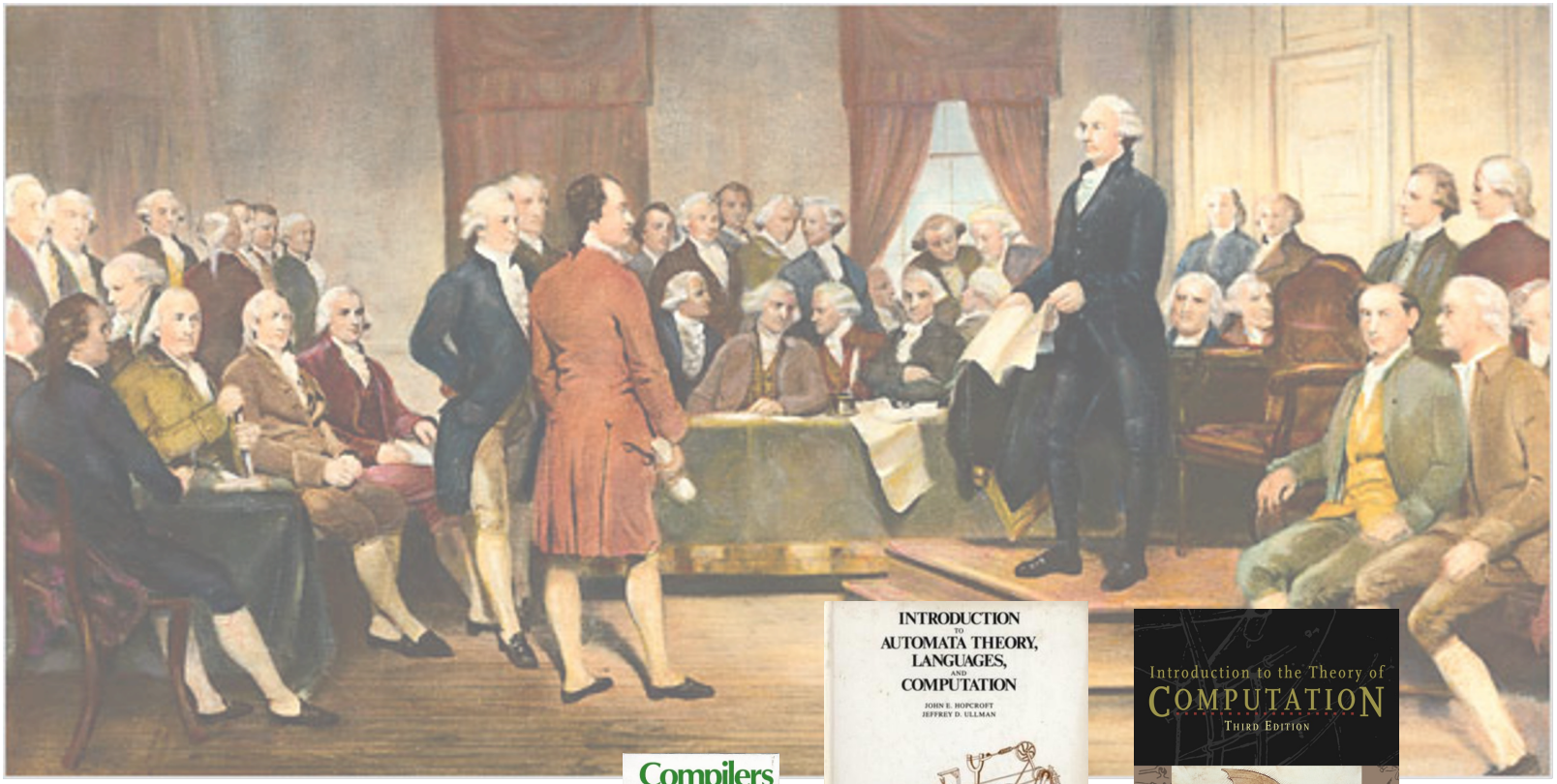
Formal Languages Theory

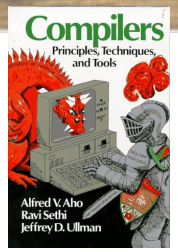


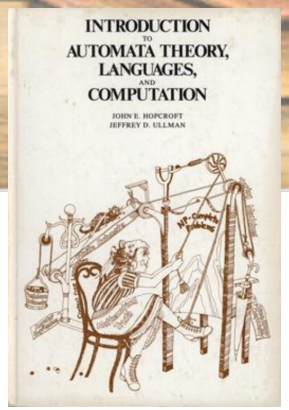


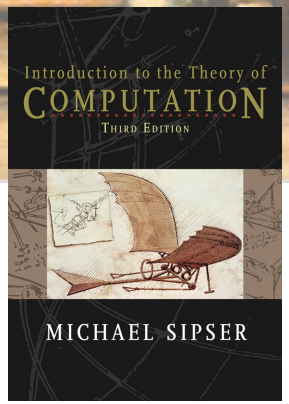


Context Free Languages: Foundations

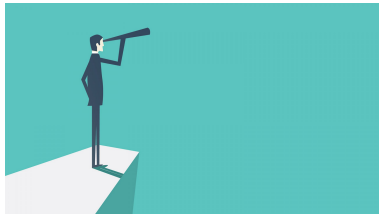








An Overview of the Land



Regular Languages

- Regular Expressions
- Finite Automata
  - Deterministic
  - Nondeterministic
- Pumping Lemma

Context Free Languages

- Context-Free Grammars
- Pushdown Automata
  - Deterministic
  - Nondeterministic
- Pumping Lemma

Context Free Grammars

Context Free Grammars

$$G = \langle V, T, P, S \rangle$$

$V \cap T = \varnothing$

Symbols

- V – non-terminals (syntactic variables)
- T – terminals
- P – derivation rules
- S – start symbol

Variables can be expanded regardless of their context

Each rule of the form $A \rightarrow \underbrace{(T \cup V)^*}_{\alpha, \beta, \mu}$, where $A \in V$

ab^+c

S → aKc

K → b

K → bK

Same rules for K, regardless of context

S ⇒ aKc

⇒ abKc

⇒ abbKc

⇒ abbbc

Example

$$G = \langle V, T, P, S \rangle$$

$S \rightarrow \varepsilon$

$S \rightarrow (S)$

$V = \{ S \}$

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

What's the language?
What's "non regular" here?

Example

$$G = \langle V, T, P, S \rangle$$

$S \longrightarrow S \texttt{;} S$

$S \longrightarrow \texttt{id} \texttt{:=} E$

$E \longrightarrow \texttt{id}$

$E \longrightarrow \texttt{num}$

$E \longrightarrow E \texttt{+} E$

$E \longrightarrow E \texttt{*} E$

$E \longrightarrow \texttt{(} E \texttt{)}$

$V = \{ S, E \}$

$T = \{ \texttt{id}, \texttt{num}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{:=}, \texttt{;} \}$

What's "non regular" here?

Notation: Useful Shorthands

The start symbol is the variable in the LHS of the first rule

$$S \rightarrow S \texttt{ ; } S$$

$$S \rightarrow \texttt{id := } E$$

$$E \rightarrow \texttt{id} \mid \texttt{num} \mid E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)}$$

$V = \{ S, E \}$

$T = \{ \texttt{id}, \texttt{num}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{:=}, \texttt{;} \}$

Notation: Useful Shorthands

The start symbol is the variable in the LHS of the first rule

$$S \rightarrow S\ ;\ S$$

$$S \rightarrow \texttt{id} := E$$

$$E \rightarrow \texttt{id} \mid \texttt{num}$$

$$\mid E + E \mid E * E \mid (E)$$

(We will usually not write T and V explicitly)

Terminology

- Let $G = \langle V, T, P, S \rangle$ be a CFG

- A sequence of symbols $\alpha A \beta$ **yields** $\alpha\gamma\beta$,

  denoted by $\alpha A \beta \Longrightarrow \alpha\gamma\beta$,

  if there is a rule $A \longrightarrow \gamma \in P$

- A sequence of symbols α **derives** β,

  denoted by $\alpha \Longrightarrow^{*} \beta$,

  if $\alpha = \beta$ or there is a sequence $\alpha \Longrightarrow \alpha_1 \Longrightarrow \ldots \Longrightarrow \beta$


```
S → S ; S
S → id := E
…
```

Derivation from S

- Let $G = \langle V, T, P, S \rangle$ be a CFG

- **Language of a Grammar** — the set of strings of terminals ("sentences") derivable from the start symbol

$$L(G) = \{ \omega \in T^* \mid S \Longrightarrow^* \omega \}$$

- **Sentential form** — the result of a partial derivation starting at S, in which there may be non-terminals

$$S \Rightarrow S\ \texttt{;}\ S \Rightarrow \texttt{id :=}\ E\ \texttt{;}\ S \Rightarrow \cdots$$

S → S ; S
S → id := E
…

* We sometimes talk about the language of a non terminal in a grammar

Example: Derivation

```
id := id ; id := id + id
```

lex

input

```
x := z;
y := x + z
```

grammar

S → S ; S

S → id := E

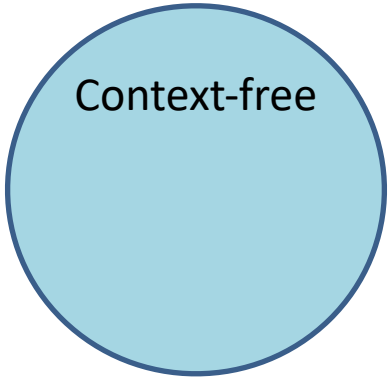
E → id | E + E | E * E | (E)

Compilation:
Terminal = Token Kind

S	S → S ; S
S ; S	S → id := E
id := E ; S	S → id := E
id := E ; id := E	E → id
id := id ; id := E	E → E + E
id := id ; id := E + E	E → id
id := id ; id := E + id	E → id
id := id ; id := id + id	

Context-Free Languages (CFLs)

- A Language L is **context-free** iff $L=L(G)$ for some context-free grammar G
  - *i.e., can be generated by some grammar*



$$L(G) = \{ \omega \in T^* \mid S \Longrightarrow^* \omega \}$$

Example

- $L_{\text{even-palindrome}} = \{pp' \mid p \in \{a,b\}^*, p'=\text{reverse}(p)\}$
  - E.g., ε, abbbba
- ?
- **Claim** $L_{\text{even-palindrome}} = L(G)$
- Proof
  - $\subseteq$ by induction on the length of the word
  - $\supseteq$ by induction on the number of derivation steps

Context Insensetivity

- Let G = ⟨V, T, P, S⟩ be a CFG
- Theorem: If $A \Longrightarrow^* \beta$ then $\gamma A \delta \Longrightarrow^* \gamma \beta \delta$
- Proof: By induction on length of derivation

Non-Determinism

- To show that $\omega \in L(G)$ we need a derivation
  - Start with the start non-terminal
  - Repeatedly replace one of the non-terminals by a right-hand side of a production
  - Stop when the sentential form is ω

- Derivation is inherently **non-deterministic**
  - Which non-terminal to develop
  - Which rule of the chosen non-terminal to use

Derivation Strategies

- **Question**: Does the derivation strategy matter?
- E.g., can different derivation orders define different languages?

Derivation Strategies

- **Question**: Does the derivation strategy matter?
- E.g., can different derivation orders define different languages?

- Important orders
  - Leftmost derivation (develop leftmost nonterminal)
  - Rightmost derivation (develop rightmost nonterminal)

Leftmost Derivation

```
x := z;
y := x + z
```

S → S;S | id := E

E → id | num | E + E

S

S ; S

id := E ; S

id := id; S

id := id; id := E

id := id; id := E + E

id := id; id := id + E

id := id; id := id + id

S → S;S

S → id := E

E → id

S → id := E

E → E + E

E → id

E → id

<id,"x"> ASS <id,"z"> ; <id,"y"> ASS <id,"x"> PLUS <id,"z">

Rightmost Derivation

```
x := z;
y := x + z
```

$S \rightarrow S;S \mid id := E$

$E \rightarrow id \mid num \mid E + E$

$$
\begin{aligned}
&S \\
&S\ ;\ S \\
&S\ ;\ id := E \\
&S\ ;\ id := E + E \\
&S\ ;\ id := E + id \\
&S\ ;\ id := id + id \\
&id := E\ ;\ id := id + id \\
&id := id\ ;\ id := id + id
\end{aligned}
$$

$$
\begin{aligned}
S &\rightarrow S;S \\
S &\rightarrow id := E \\
E &\rightarrow E + E \\
E &\rightarrow id \\
E &\rightarrow id \\
S &\rightarrow id := E \\
E &\rightarrow id
\end{aligned}
$$

<id,"x"> ASS <id,"z"> ; <id,"y"> ASS <id,"x"> PLUS <id,"z">

Derivation Strategy is Immaterial wrt. Showing Membership

Lemma: Let $G=\langle V, T, P, S\rangle$ be a context-free grammar. For every sentence $\omega \in T^*$, for every derivation $S \Rightarrow^* \omega$ there is

(a) a leftmost derivation $S \Rightarrow^*_{lm} \omega$

(b) a rightmost derivation $S \Rightarrow^*_{rm} \omega$

Claim: For every sentence $\omega \in T^*$, for every sequence of symbols α, if $\alpha \Rightarrow^n \omega$ then there is a leftmost/rightmost derivation $\alpha \Rightarrow^n_{lm} \omega$ / $\alpha \Rightarrow^n_{rm} \omega$

Proof: By induction on n

Ambiguity

- Leftmost/rightmost derivations dictate which nonterminal to develop but not which rule to use!
- Some grammars allow for different leftmost/ rightmost derivations for some sentences
- We refer to such grammars as **ambiguous**

Example: Ambiguous Grammar

- $E \longrightarrow 1 \mid E - E$
- $\omega = 1 - 1 - 1$

$E \Rightarrow E - E$

$\Rightarrow 1 - E$

$\Rightarrow 1 - E - E$

$\Rightarrow 1 - 1 - E$

$\Rightarrow 1 - 1 - 1$

$E \Rightarrow E - E$

$\Rightarrow E - E - E$

$\Rightarrow 1 - E - E$

$\Rightarrow 1 - 1 - E$

$\Rightarrow 1 - 1 - 1$

Is Ambiguity a Problem?

- Yes
  - The intended meaning of the sentence is unclear

Example: Ambiguous Grammar

- $E \longrightarrow 1 \mid E - E$
- $\omega = 1 - 1 - 1$

$E \Rightarrow E - E$

$\Rightarrow 1 - E$

$\Rightarrow 1 - E - E$

$\Rightarrow 1 - 1 - E$

$\Rightarrow 1 - 1 - 1$

1

$E \Rightarrow E - E$

$\Rightarrow E - E - E$

$\Rightarrow 1 - E - E$

$\Rightarrow 1 - 1 - E$

$\Rightarrow 1 - 1 - 1$

-1

Is Ambiguity a Problem?

- Yes
  - The intended meaning of the sentence is unclear
  - Checking membership is more complicated for ambiguous grammars
  - *We'll get back to this later on*

Parse Trees

Aka: syntax trees, derivation trees

Parse Trees: Traces of Derivations

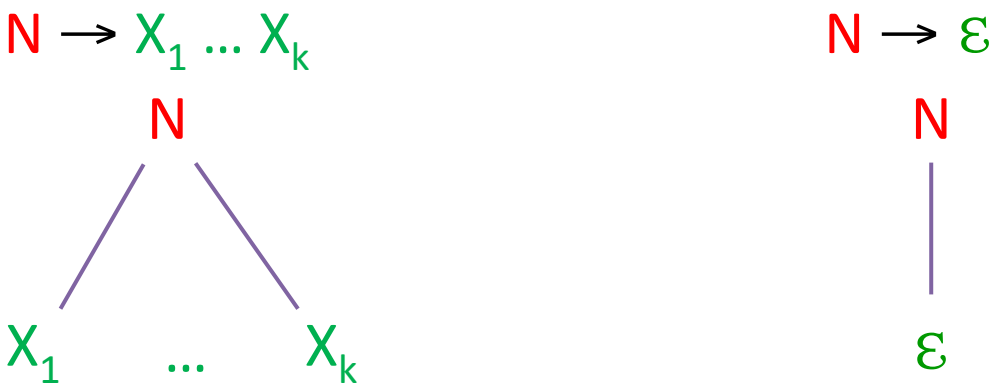
Not: abstract syntax trees

- A parse tree is a graphical representation of a derivation
  - Filters out the order in which productions are applied to replace nonterminals

- Parse tree "diagram" the sentence
  - Exposes (some of its) intended meaning

Parse trees: Traces of Derivations

- Tree nodes are labeled with symbols, children ordered left-to-right
- Each internal node is non-terminal
  - Children correspond to one of its productions



- Root is the start non-terminal
- Leaves are terminals, non-terminals, ε

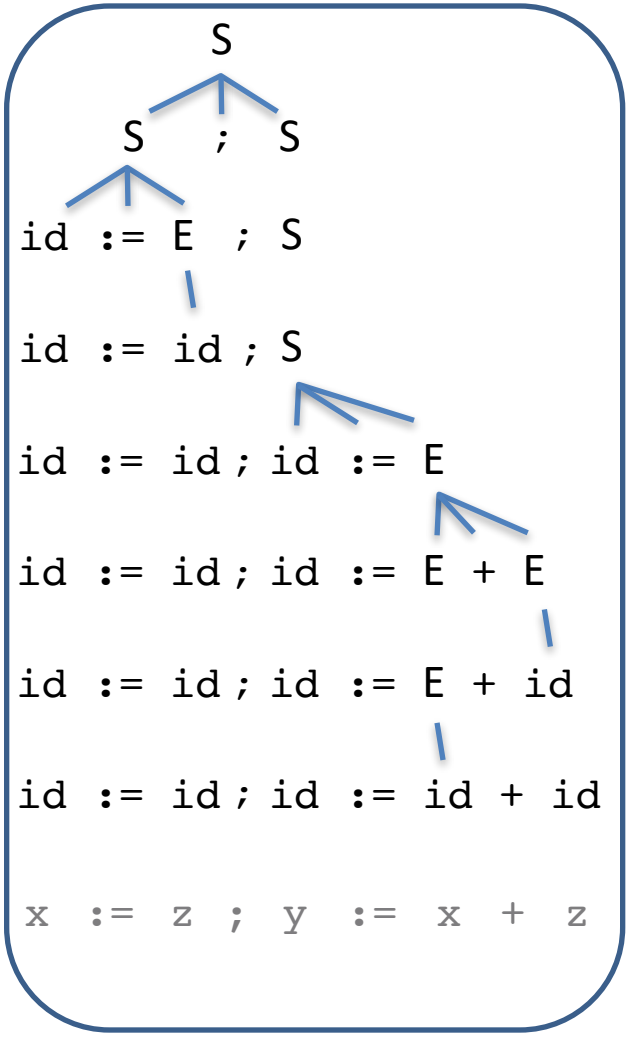
- ***Yield*** of parse tree: left-to-right walk over leaves

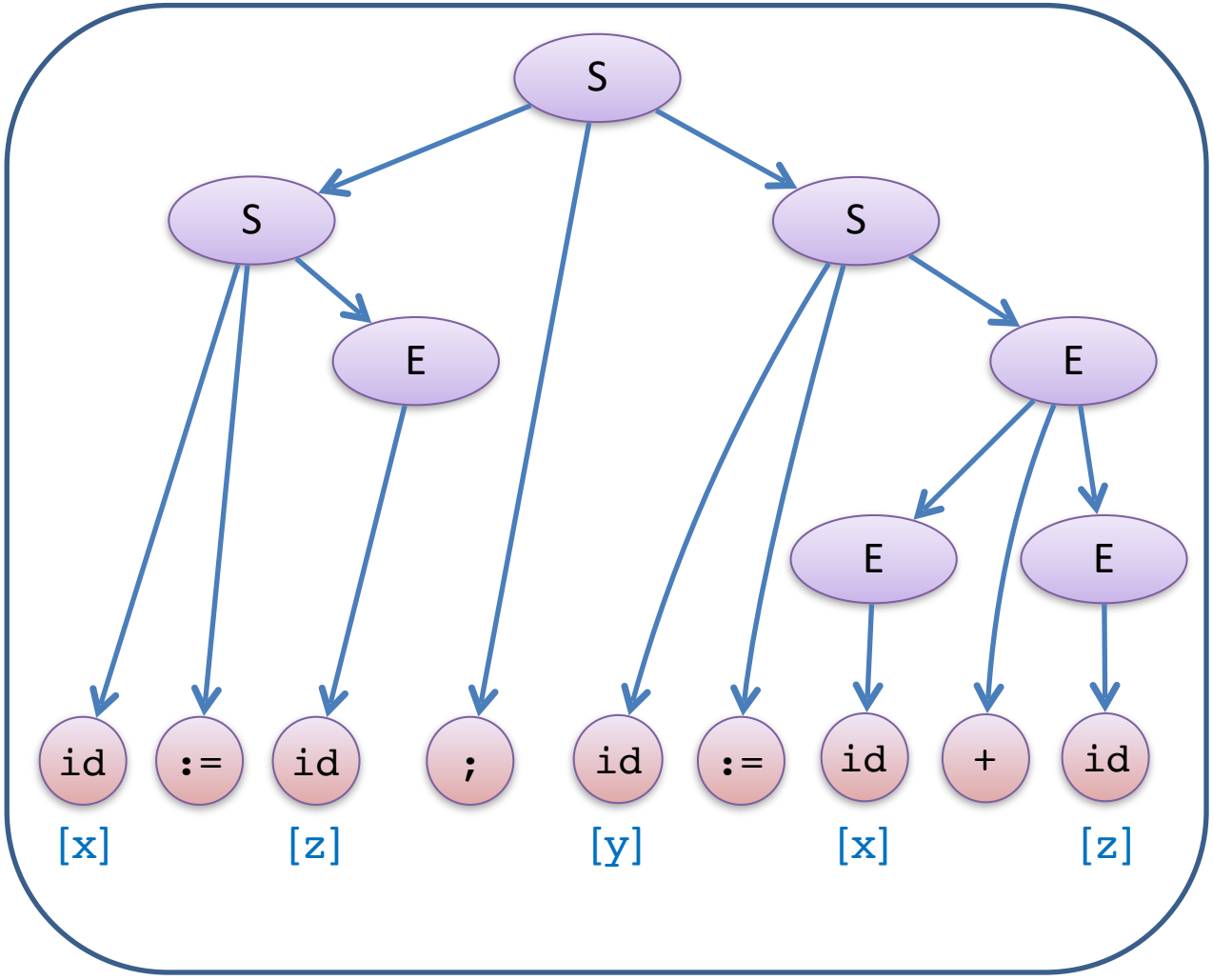
```
x := z;
y := x + z
```

Parse Tree

```
S → S;S | id := E
E → id | E + E
```

also: Syntax Tree, Derivation Tree





```
x := z;
y := x + z
```

Parse Tree

```
S → S;S | id := E
E → id | E + E
```

also: Syntax Tree, Derivation Tree

```
                                S
                          S ; S
                          S ; id := E
                          S ; id := E + E
                          S ; id := E + id
                          S ; id := id + id
                  id := E ; id := id + id
                  id := id ; id := id + id
<id,"x"> ASS <id,"z"> ; <id,"y"> ASS  <id,"x"> PLUS <id,"z">
```

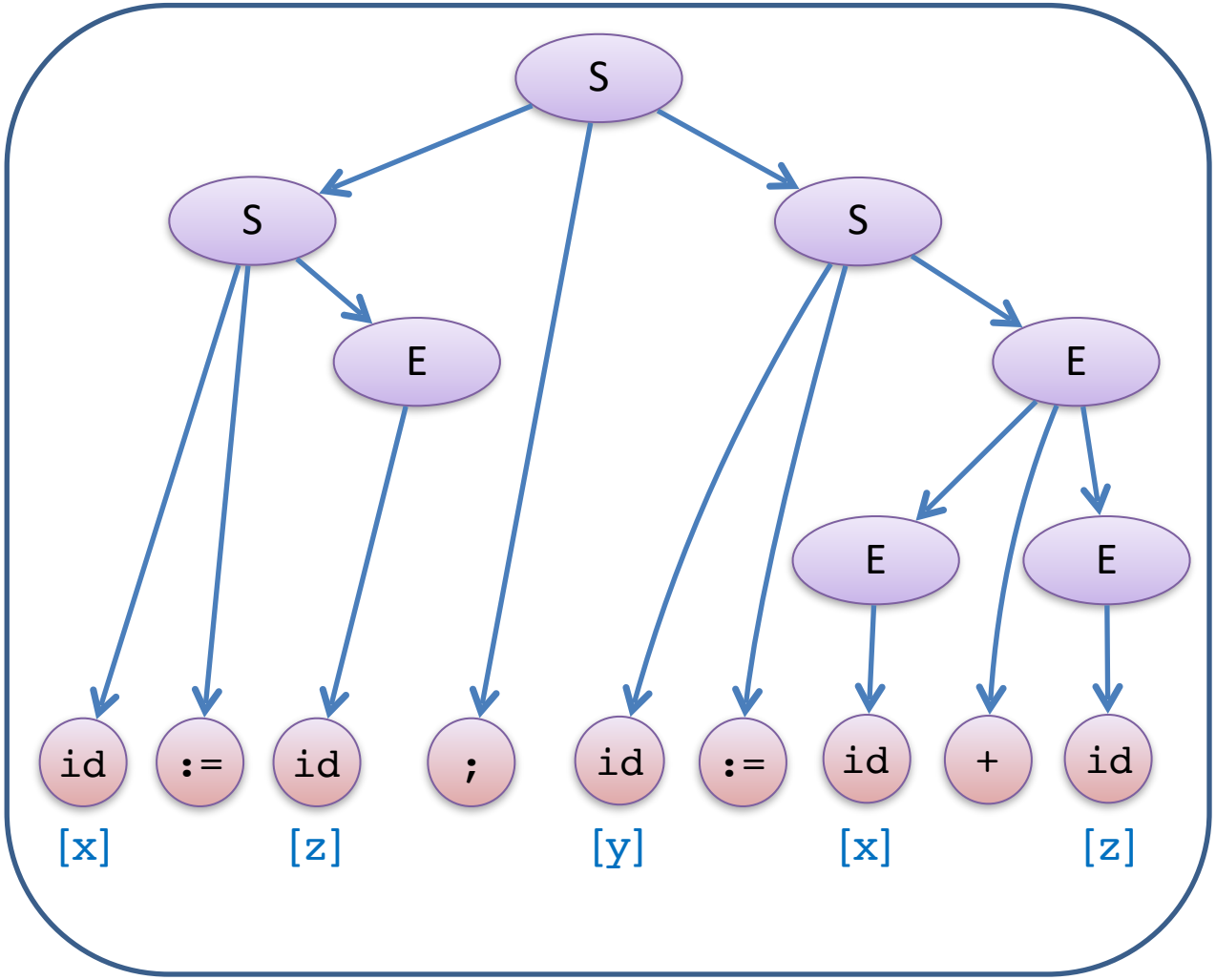
```
S → S;S
S → id := E
E → E + E
E → id
E → id
S → id := E
E → id
```

Rightmost derivation

```
                                S
                          S ; S
                  id := E ; S
                  id := id; S
                  id := id; id := E
                  id := id; id := E + E
                  id := id; id := id + E
                  id := id; id := id + id
<id,"x"> ASS <id,"z"> ; <id,"y"> ASS  <id,"x"> PLUS <id,"z">
```

```
S → S;S
S → id := E
E → id
S → id := E
E → E + E
E → id
E → id
```

Leftmost derivation



Parse Trees vs. Yield Relations

- Every derivation is traced by a single parse tree
- Every parse tree traces (possibly multiple) derivation(s)

Theorem: Let $G=\langle V, T, P, S\rangle$ be a context-free grammar and $\alpha \in (V \cup T)^*$.

$S \Rightarrow^* \alpha$ *iff* G has a parse tree that yields α

Note: Different parse trees can yield the same sentence

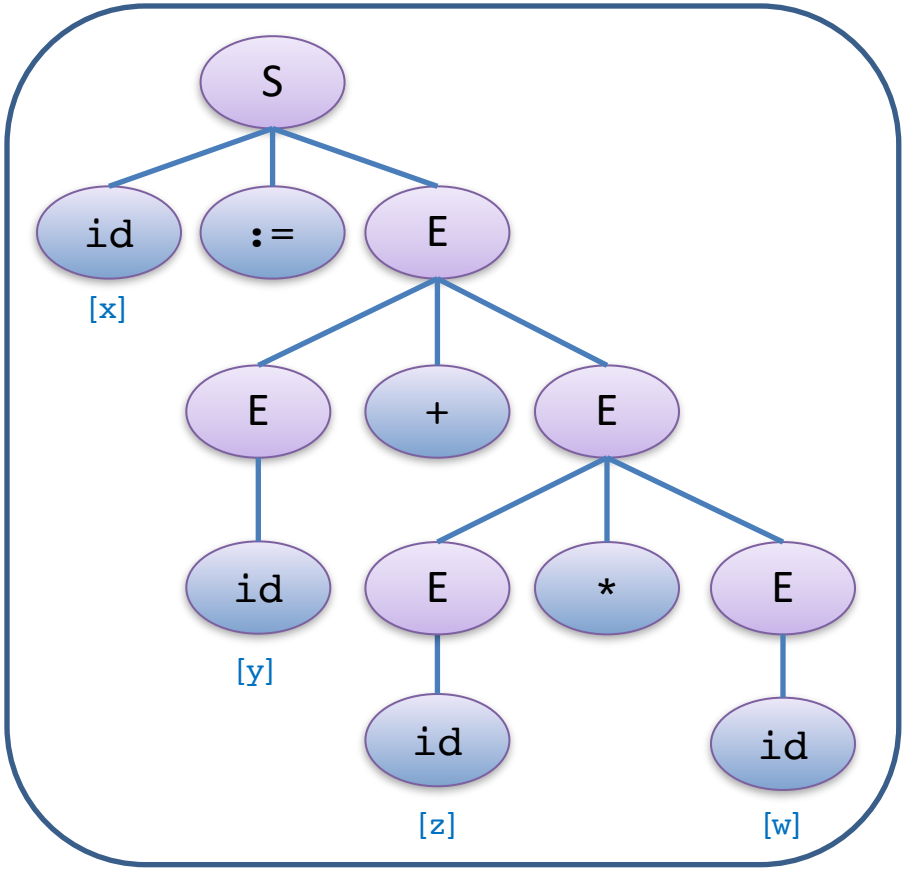
Are multiple parse trees a problem?

- Yes
  - The intended meaning of the sentence is unclear
  - Checking membership is more complicated for ambiguous grammars

Ambiguity

```
x := y+z*w
```

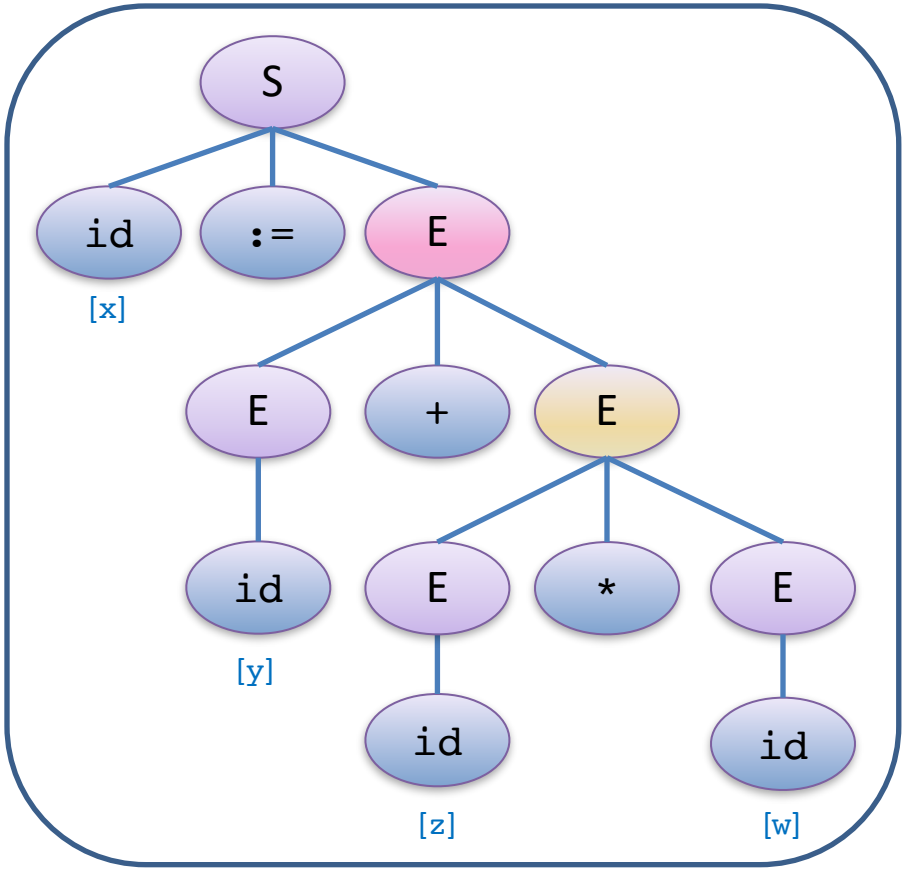
```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```

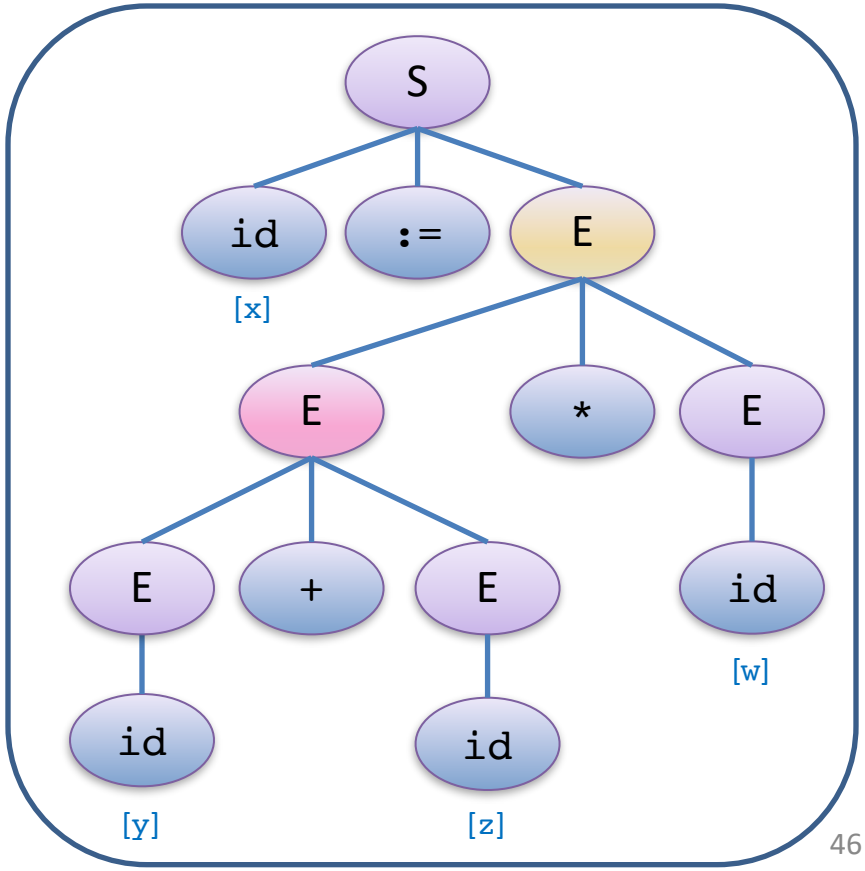


Ambiguity

```
x := y+z*w
```

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```





Ambiguity is a Problem!

- In compilation, we diagram the program using the parse tree* in order to extract the intended meaning of the problem

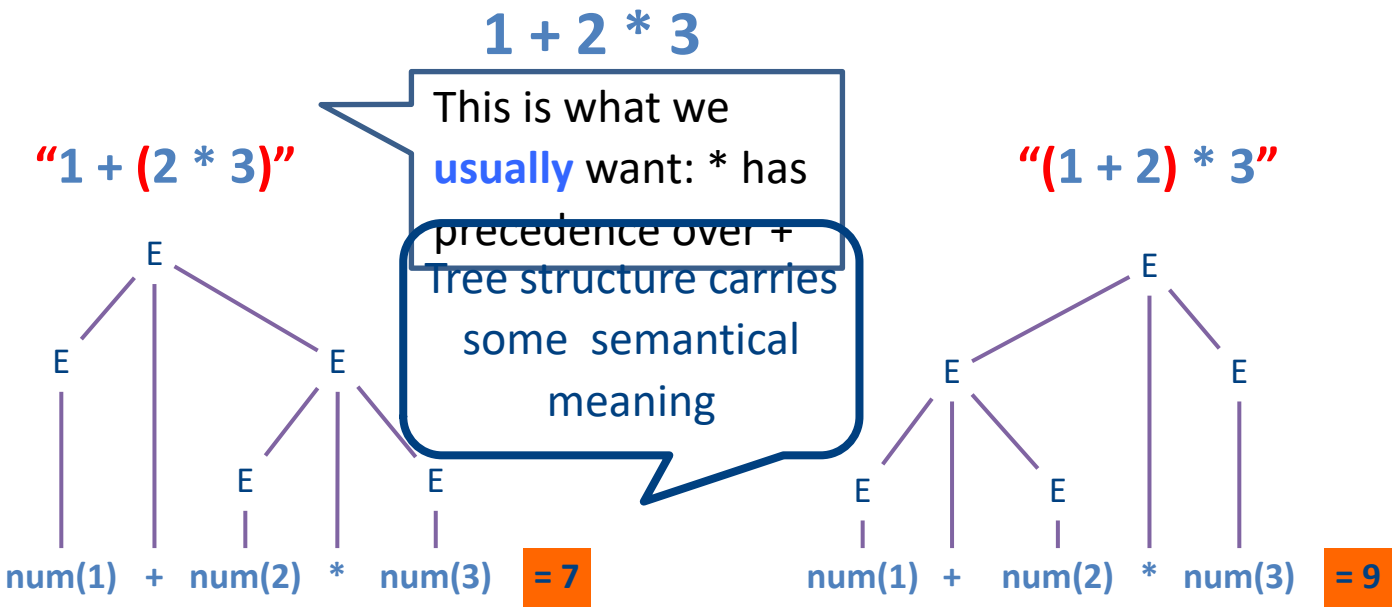
➡ Two parse tree = two possible interpretations of the of the program

*But constructs the abstract syntax tree, which is an abstraction of the parse tree

Problematic Ambiguity Example

Arithmetic expressions:

E → id
E → num
E → E + E
E → E * E
E → (E)



Leftmost derivation

E
E + E
num + E
num + E * E
num + num * E
num + num * num

Rightmost derivation

E
E * E
E * num
E + E * num
E + num * num
num + num * num

Problematic Ambiguity Example

Arithmetic expressions:

E → id

E → num

E → E + E

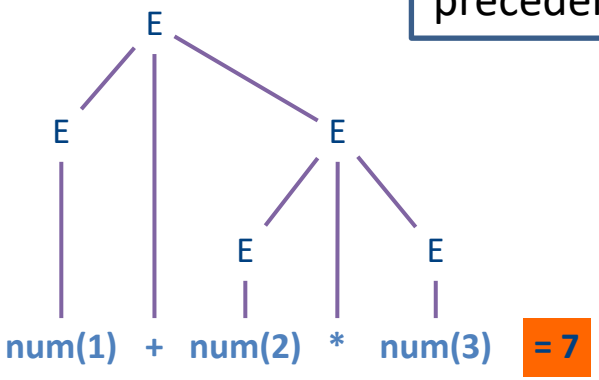
E → E * E

E → (E)

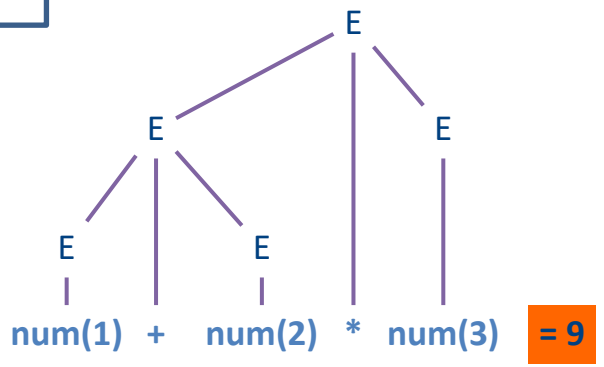
1 + 2 * 3

This is what we **usually** want: * has precedence over +

"1 + (2 * 3)"



"(1 + 2) * 3"



Leftmost derivation

E

E + E

num + E

num + E * E

num + num * E

num + num * num

Leftmost derivation'

E

E * E

E + E * E

num + E * E

num + num * E

num + num * num

What are the Parse Trees?

- $E \longrightarrow 1 \mid E - E$
- $\omega = 1 - 1 - 1$

$E \Rightarrow E - E$

$\Rightarrow 1 - E$

$\Rightarrow 1 - E - E$

$\Rightarrow 1 - 1 - E$

$\Rightarrow 1 - 1 - 1$

1

$E \Rightarrow E - E$

$\Rightarrow E - E - E$

$\Rightarrow 1 - E - E$

$\Rightarrow 1 - 1 - E$

$\Rightarrow 1 - 1 - 1$

-1

Ambiguous grammars

- A grammar is *ambiguous* if there exists a sentence for which there are
  - Two different parse trees, or
  - Two different leftmost derivations, or
  - Two different rightmost derivations

- Property of *grammars*, not *languages*
  - *e.g., language of arithmetical expressions*

- Some languages are *inherently ambiguous*: No unambiguous grammars exist
  - $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$
  - Claim: L is a CFL (easy). L is inherently ambiguous (see Hopcroft)

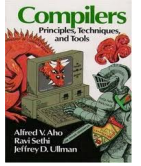
! No algorithm to detect whether arbitrary grammar is ambiguous

Drawbacks of ambiguous grammars

- Ambiguous semantics
  - 1 + 2 * 3 = 7 or 9
- Parsing complexity

- Solutions
  - Transform grammar into non-ambiguous
  - Handle as part of parsing method
    - Using special form of “precedence”

Transforming ambiguous grammars to non-ambiguous by layering



p. 212

Ambiguous grammar

E → E + E

E → E * E

E → id

E → num

E → (E)

Unambiguous grammar

E → E + T — **Layer 1**

E → T

T → T * F — **Layer 2**

T → F

F → id — **Layer 3**

F → num

F → (E)

Each layer takes care of one way of composing sub-strings to form a string:
1: by +
2: by *
3: atoms

Problems:

- Defining precedence
- Defining associativity

Transformed grammar: * precedes +

Ambiguous grammar

E → E + E

E → E * E

E → id

E → num

E → (E)

Let's derive 1 + 2 * 3

Unambiguous grammar

E → E + T

E → T

T → T * F

T → F

F → id

F → num

F → (E)

Derivation

E

=> E + T

=> T + T

=> F + T

=> 1 + T

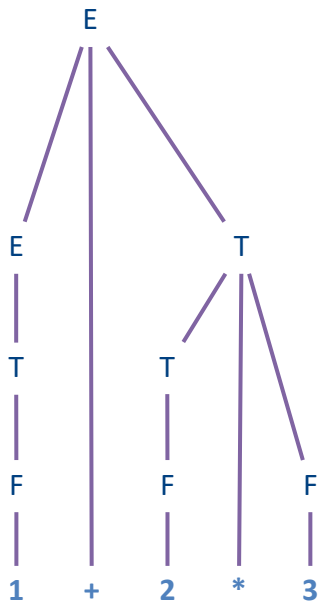
=> 1 + T * F

=> 1 + F * F

=> 1 + 2 * F

=> 1 + 2 * 3

Parse tree



Transformed grammar: + precedes *

Ambiguous grammar

E → E + E
E → E * E
E → id
E → num
E → (E)

Let's derive 1 + 2 * 3

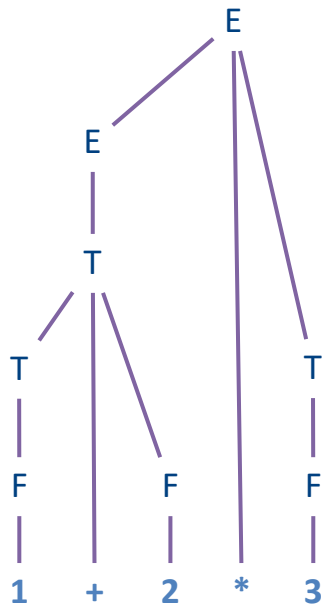
Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

Derivation

E
=> E * T
=> T * T
=> T + F * T
=> F + F * T
=> 1 + F * T
=> 1 + 2 * T
=> 1 + 2 * F
=> 1 + 2 * 3

Parse tree



Transformed grammar: * left-associative

Ambiguous grammar

E → E + E
E → E * E
E → id
E → num
E → (E)

Let's derive 1 * 2 * 3

Unambiguous grammar

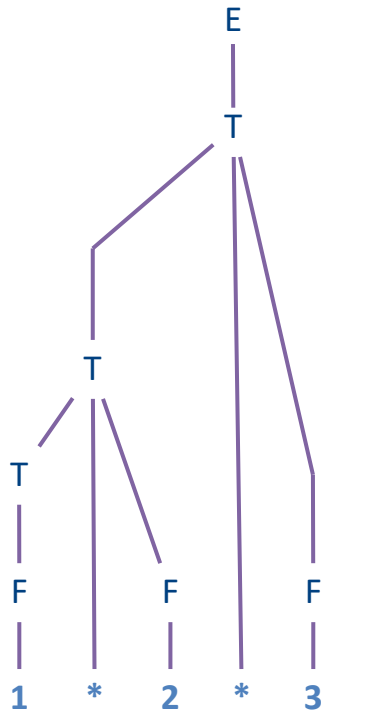
E → E + T
E → T
T → T * F
T → F
F → id
F → num
F → (E)

T * F defines left associativity

Derivation

E
=> T
=> T * F
=> T * F * F
=> F * F * F
=> 1 * F * F
=> 1 * 2 * F
=> 1 * 2 * 3

Parse tree



Transformed grammar: * right-associative

Ambiguous grammar

E → E + E
E → E * E
E → id
E → num
E → (E)

Let's derive 1 * 2 * 3

Unambiguous grammar

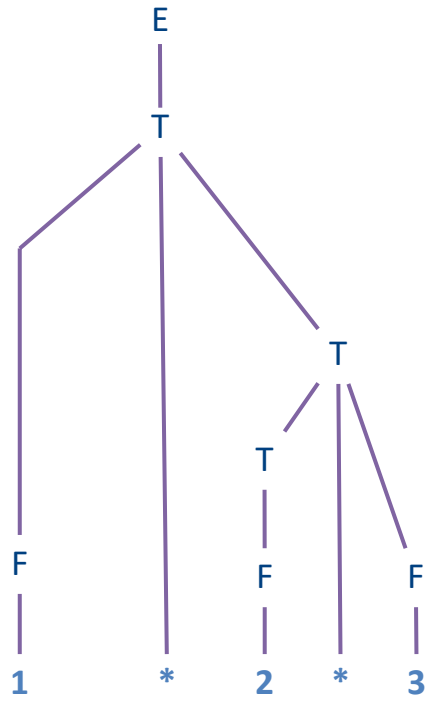
E → E + T
E → T
T → F * T
T → F
F → id
F → num
F → (E)

F * T defines right associativity

Derivation

E
=> T
=> F * T
=> 1 * T
=> 1 * F * T
=> 1 * F * F
=> 1 * 2 * F
=> 1 * 2 * 3

Parse tree

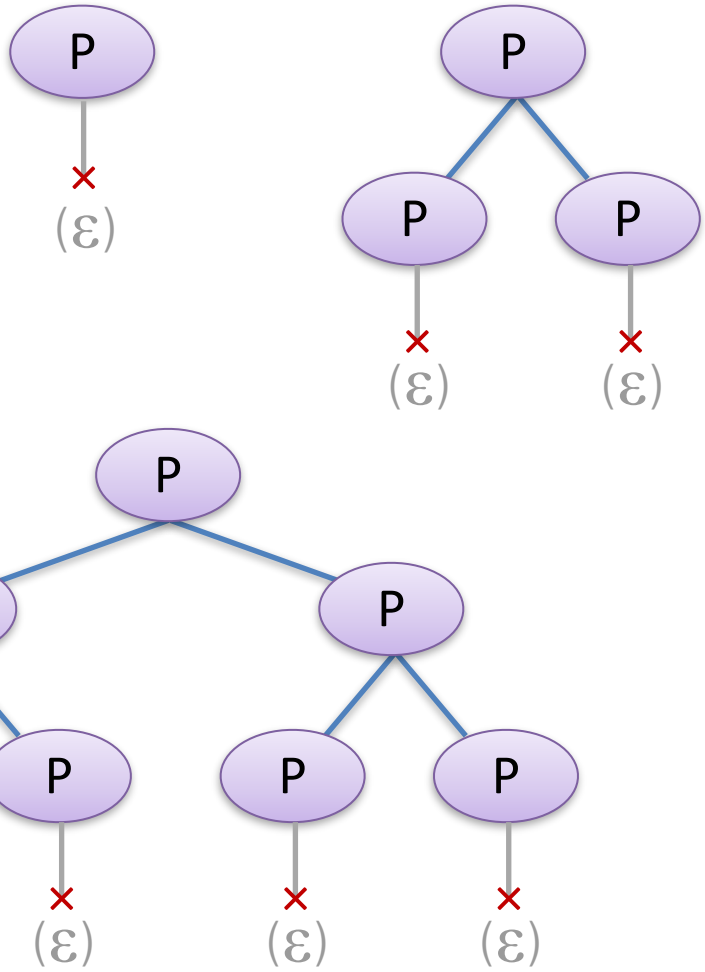


Another Example for Ambiguity

$P \longrightarrow \varepsilon$

$P \longrightarrow (P)$

$P \longrightarrow P\ P$

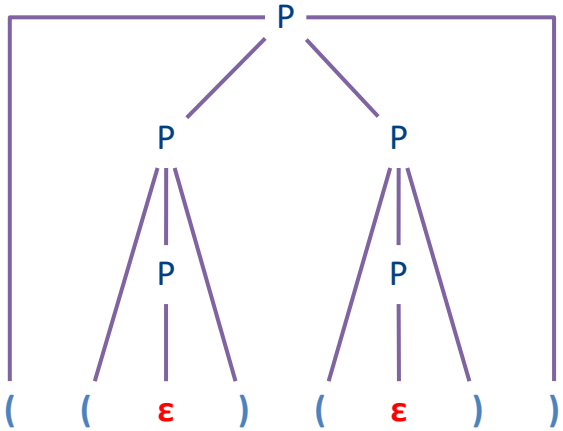


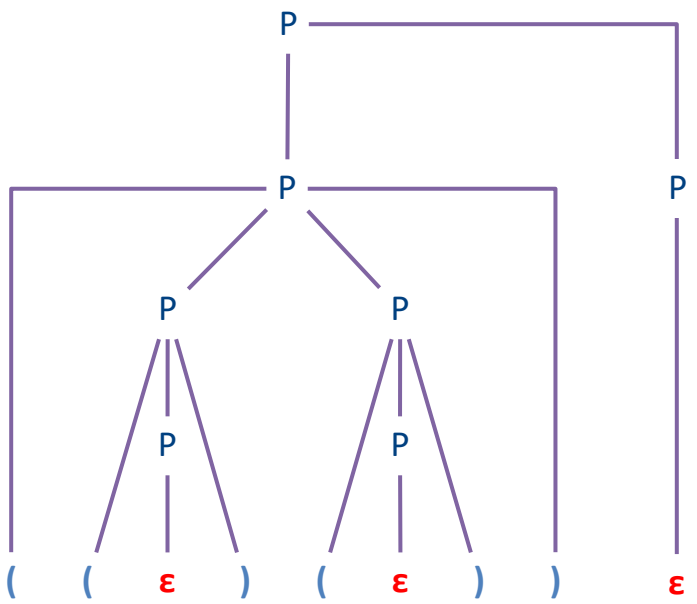
Yet Another Example for Ambiguity

$P \rightarrow \varepsilon$

$P \rightarrow (P)$

$P \rightarrow P P$





Another example for layering

Ambiguous grammar

P → ε
  | P P
  | (P)

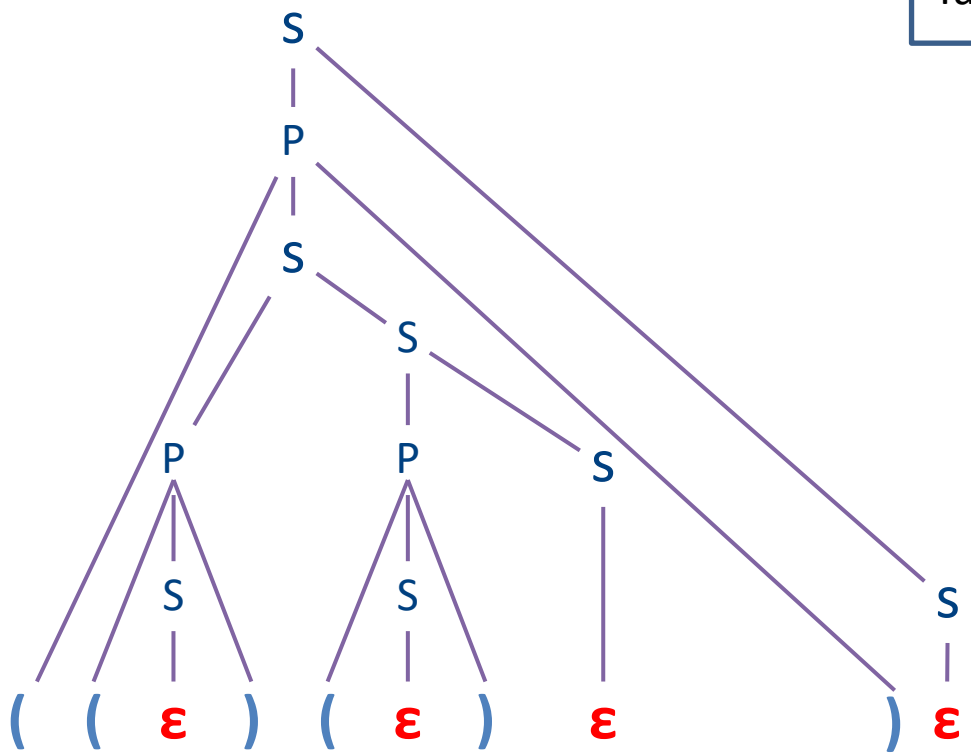
Unambiguous grammar

S → P S
  | ε
P → (S)

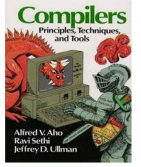
Takes care of "concatenation"

Takes care of nesting

(( ) ( ))



“dangling-else” example



p. 118

Ambiguous grammar

S → if E then S

S | if E then S else S

| other

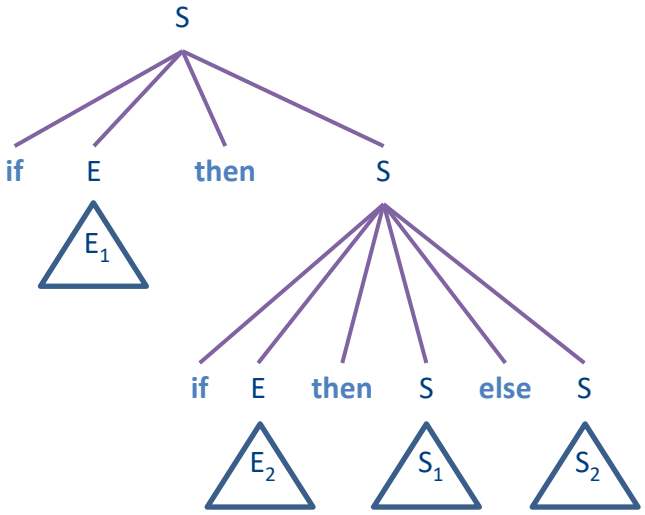
This is what we usually want: match **else** to closest unmatched **then**

Unambiguous grammar

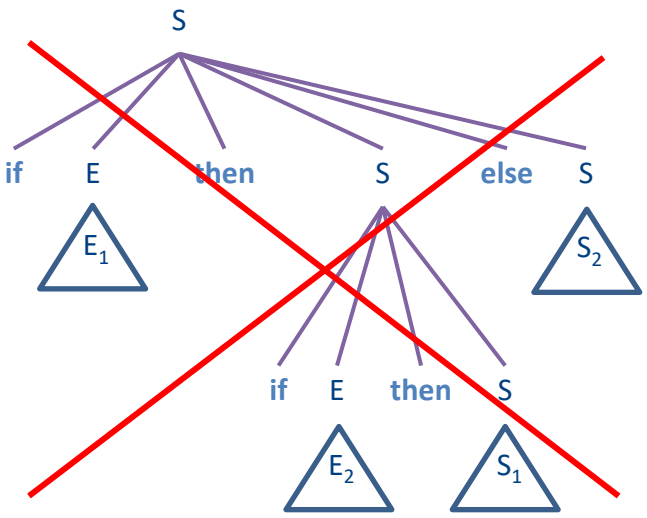
?

if E_1 then if E_2 then S_1 else S_2

if E_1 then (if E_2 then S_1 else S_2)

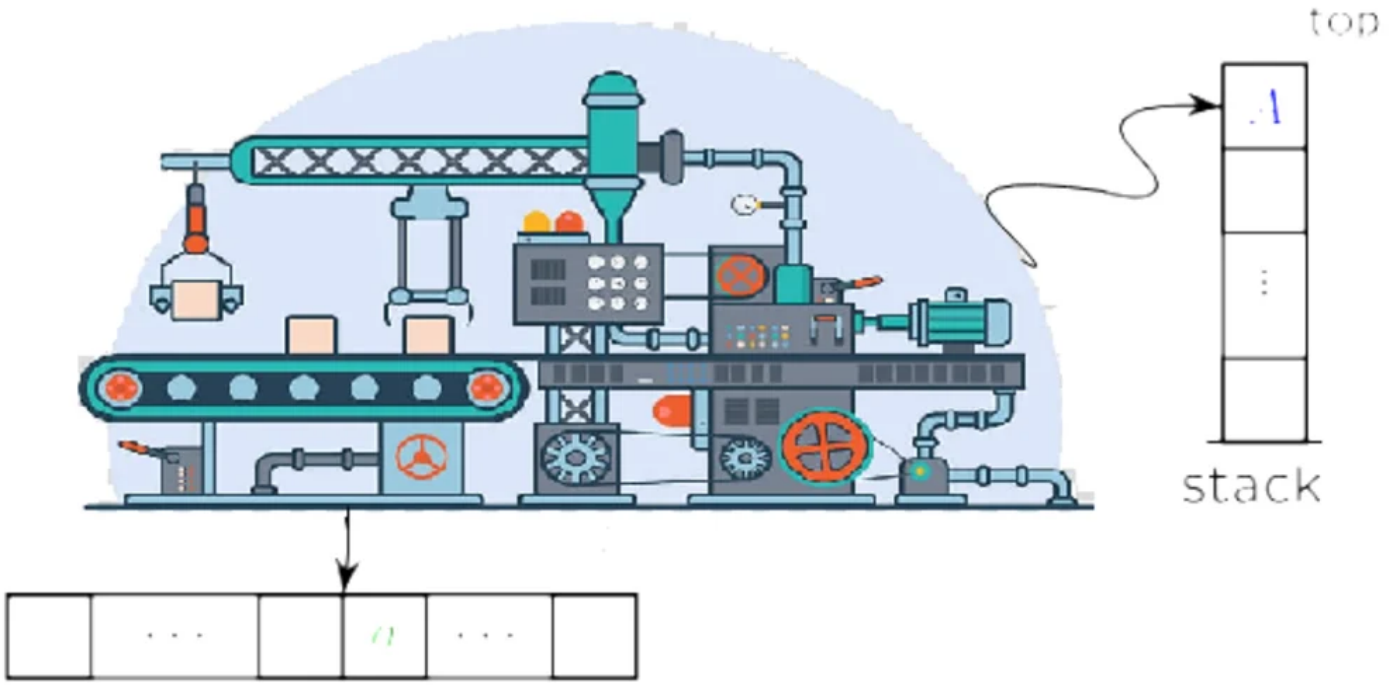


if E_1 then (if E_2 then S_1) else S_2



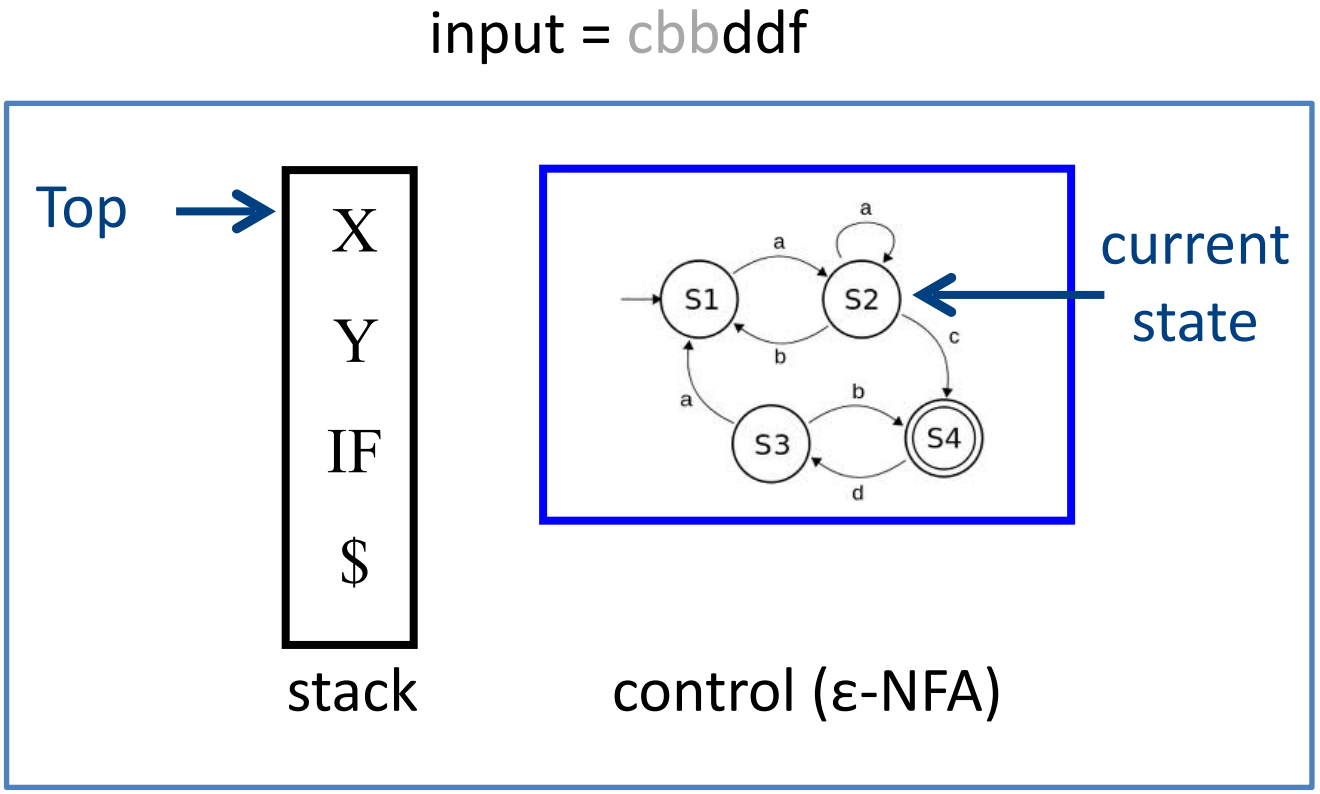
Pushdown Automata (PDA)

- CFGs generate CFLs
- PDAs recognize (accept) CFLs



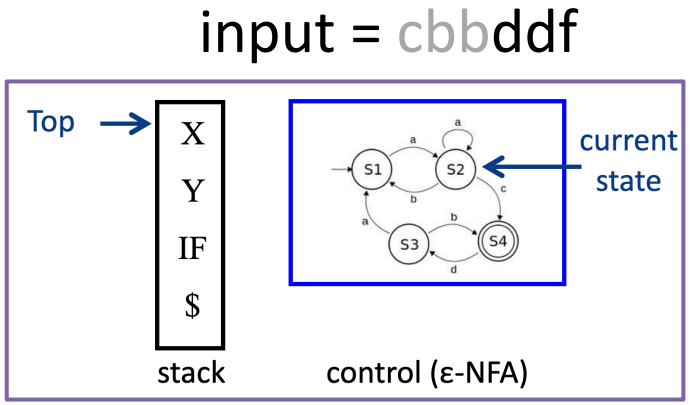
Intuition: PDA Components

- An ε-NFA with the additional power to manipulate **one unbounded** stack



Intuition: PDA Moves

- PDA **moves** are determined by:
  - The current state (of its "ε-NFA")
  - The current symbol on top of its stack
  - (Possibly) the current input symbol



- PDA moves by (non-deterministically):
  - Changing state of ε-NFA
  - Popping the symbol at the top of the stack and replacing it by 0…n symbols
  - (Possibly) removing current input symbol

Example #1 (Informal)

- PDA for $L = \{0^n 1^n \$ \mid n \geq 0\}$
  1. Start with a stack containing Z
  2. Read input symbols (from left to right)
     2.1. Push an *X* for each read *0*
     2.2. Pop an *X* for each read *1*
  3. Accept if when reading $ the top of the stack is Z

$ is often use in compilation theory as the end-of-input symbol

Example #2 (Informal)

- PDA for $L = \{a^i b^j c^k \$ \mid i = j \vee i = k\}$
  1. Start with a stack containing Z
  2. Read input symbols (from left to right)
     2.1. Read *a*'s and push *X* on the stack
     2.2. Either
        2.2.1. Pop and match *b*'s, and ignore *c*'s
        2.2.2. ignore *b*'s, and pop and match *c*'s
  3. Accept if when reading $ the top of the stack is Z

PDA Formalism

- PDA = (Q, Σ, Γ, δ, q_0, S, F):
  - Q: a finite set of states
  - Σ: Input symbols alphabet (a finite set)
  - Γ: stack symbols alphabet (a finite set)
  - δ: transition function
  - $q_0 \in Q$: start state
  - $S \in \Gamma$: start stack symbol*
  - $F \subseteq Q$: set of final (accepting) states**

* In some variants, it is assumed that the initial stack is empty

** In some variants, it is assumed that the PDA accepts when the run ends in an empty stack

The Transition Function

- $\delta: (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \rightarrow_{fin} \mathcal{P}(Q \times \Gamma^*)$
- $(p, \alpha) \in \delta(q, a, X)$ means
  - If the PDA is
    - In state q
    - X is the top stack symbol
    - a = the next input symbol (if $a = \varepsilon$, input is immaterial)
  - Then the PDA can
    - Change the state to p
    - Remove a from the front of the input
      - (if $a = \varepsilon$, the input is left as is)
    - Replace X on the top of the stack by α
      - if $\alpha = X_1 \cdots X_k$, then X_1 becomes the top of the stack
      - α can be ε

Example (Formal)

- Design a PDA to accept $L = \{0^n 1^n \$ \mid n > 0\}$
- PDA = (Q, Σ, Γ, δ, q_0, S, F):
  - Q = { p, q, f }
  - Σ = { 0, 1, $ }
  - Γ = { X, Z }
  - δ: …
  - q_0 = q
  - S = Z
  - F = { f }

$ is often use in compilation theory as the end-of-input symbol

Example: States

- q = We have not seen 1 so far
  - start state
- p = we have seen at least one 1 and no 0s since
- f = final state; accept

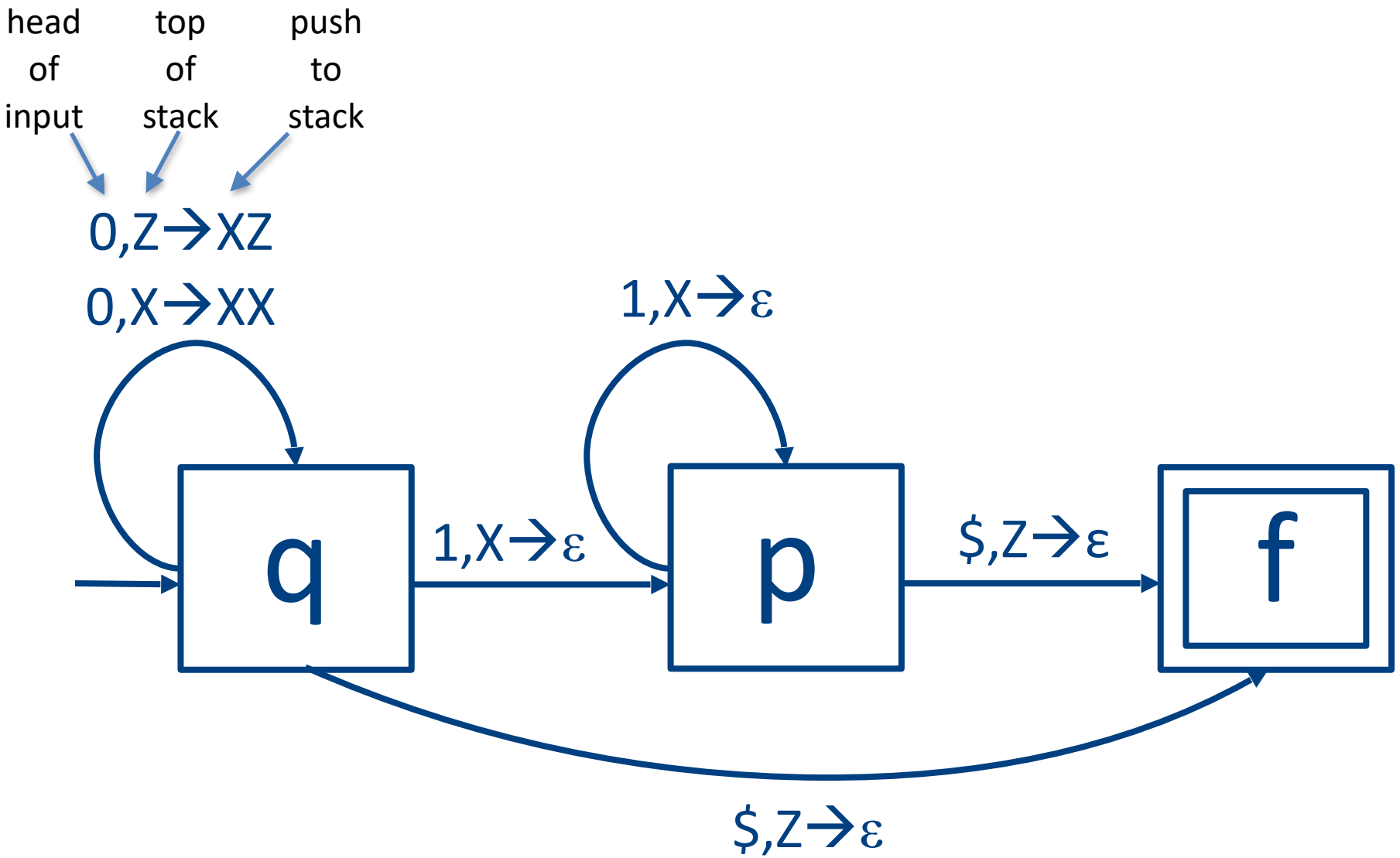
Example: Stack Symbols

- X = “counter”
  - Counts the number of *0*’s we saw so far
- Z = bottom of the stack
  - A common trick: Allows to detect an empty stack
  - Indicates when we have counted the same number of 1’s as 0’s

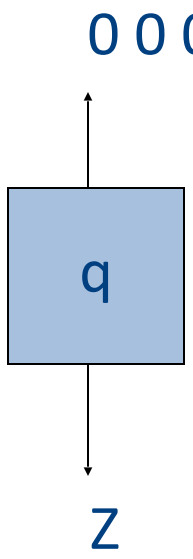
Example: Transition Function

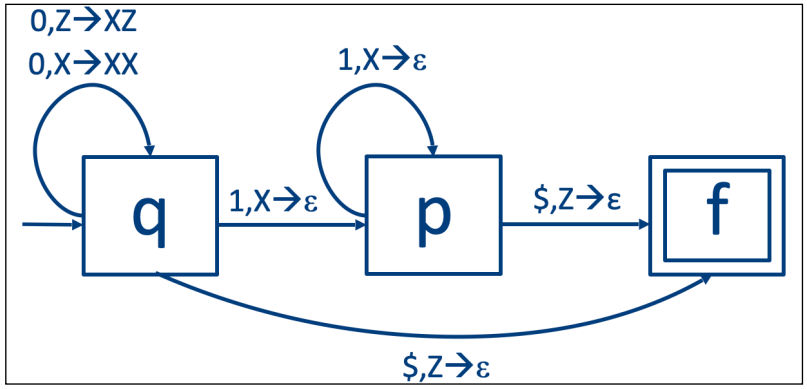
- $\delta(q, 0, Z) = \{(q, XZ)\}$
- $\delta(q, 0, X) = \{(q, XX)\}$
  - These two rules cause one X to be pushed onto the stack for each 0 read from the input
- $\delta(q, 1, X) = \{(p, \varepsilon)\}$
  - When we see a 1, go to state p and pop one X
- $\delta(p, 1, X) = \{(p, \varepsilon)\}$
  - Pop one X per 1
- $\delta(p, \$, Z) = \{(f, \varepsilon)\}$
  - Accept at bottom
- $\delta(p, 0, X) = \delta(p, 0, Z) = \{\}$
  - If a 0 appears after a 1, get stuck

Example: Graphical Notation



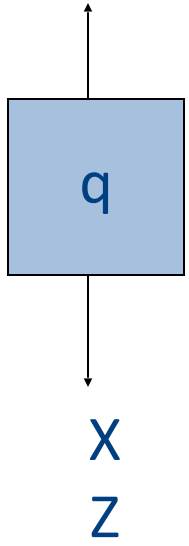
Run of the Example PDA

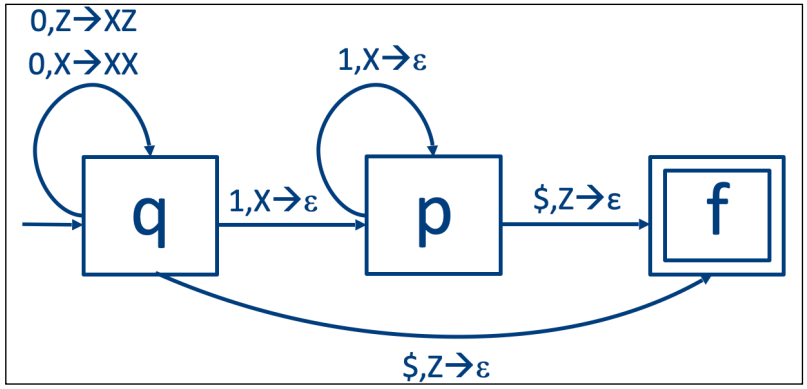




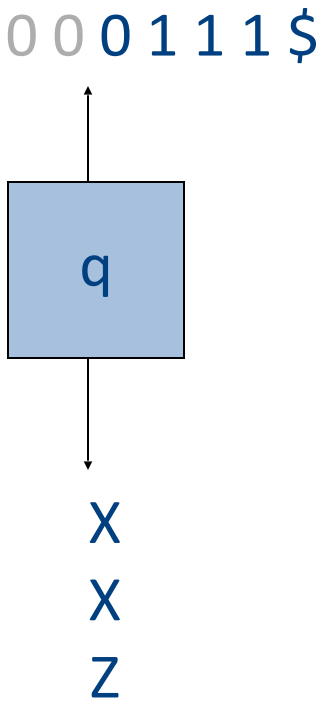
Run of the Example PDA

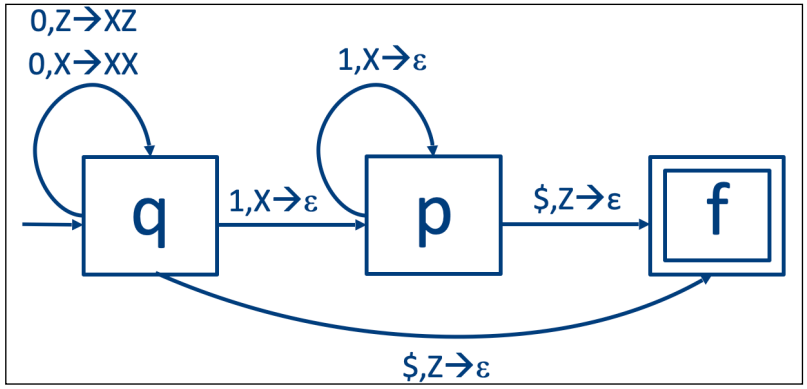
0 0 0 1 1 1 $



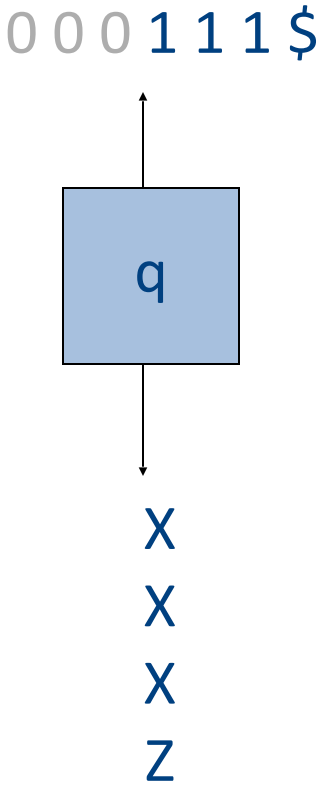


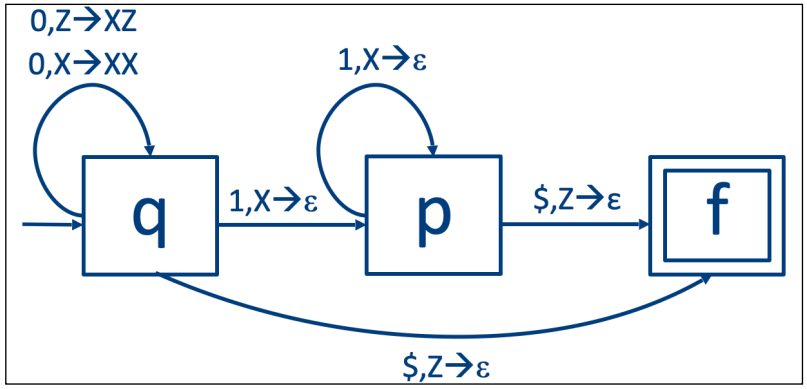
Run of the Example PDA





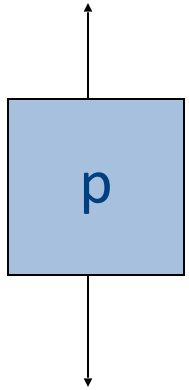
Run of the Example PDA



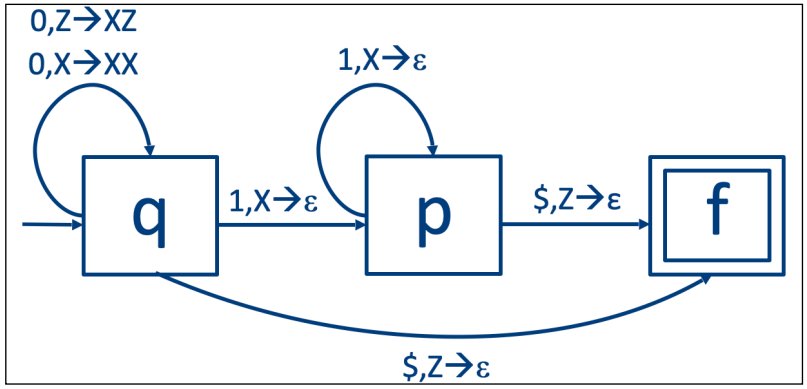


Run of the Example PDA

0 0 0 1 1 1 $

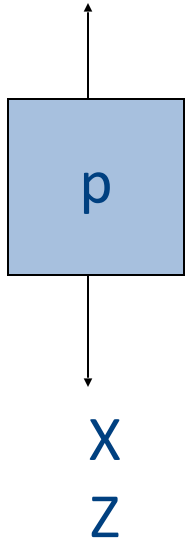


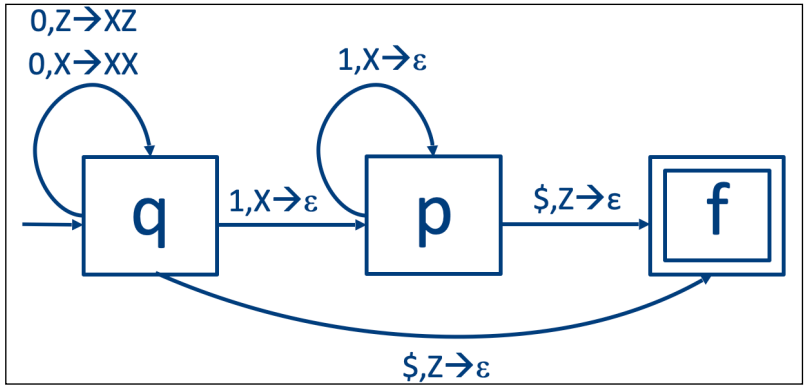
X
X
Z



Run of the Example PDA

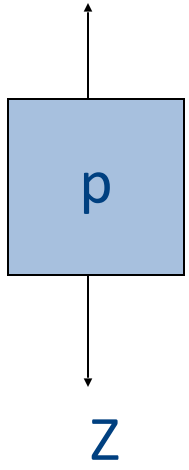
0 0 0 1 1 1 $

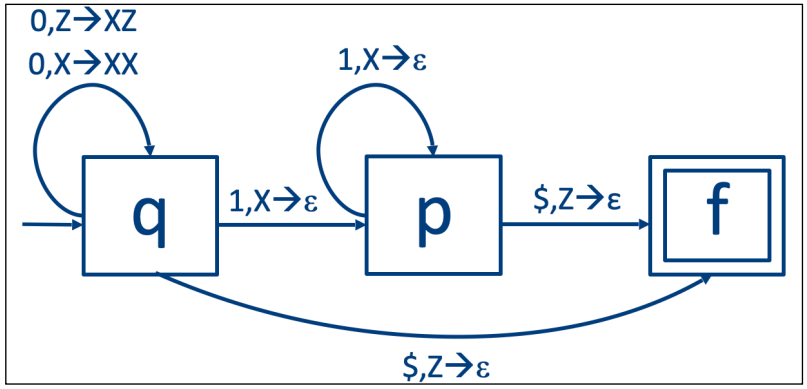




Run of the Example PDA

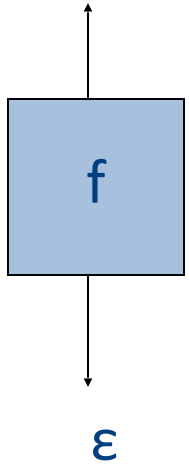
0 0 0 1 1 1 $

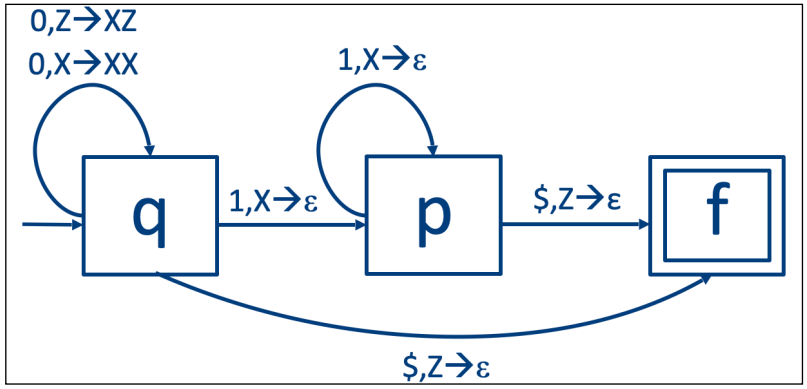




Run of the Example PDA

0 0 0 1 1 1 $





Model of Computation

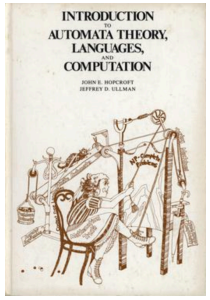
- Let P = (Q, Σ, Γ, δ, q_0, S, F) be a PDA
- P **accepts** $\omega \in \Sigma^*$ if there is a "run" of P that reads all the input and ends in an accepting state
  - Alt. P **accepts** (by an empty stack) there is a "run" of P that reads all the input and ends with an empty stack

- The **language** of P is $L(P) = \{\omega \in \Sigma^* \mid P \text{ accepts } \omega\}$

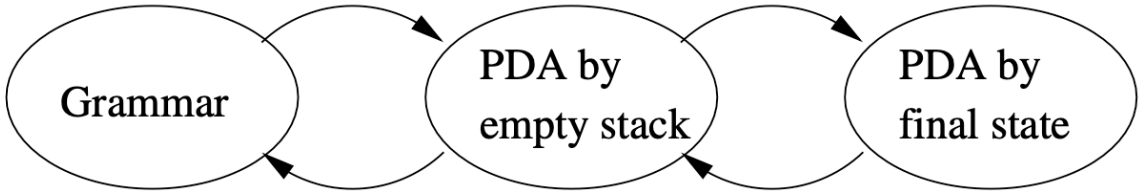
Equivalence of PDA's and CFG's

The following three classes of languages are the same

- Context-free languages (i.e., languages defined by a CFG)
- The languages that are accepted by final state for some PDA
- The languages that are accepted by empty stack for some PDA



p. 243



PDA vs DPDA

Question: What's the (Qualitative) Difference from a PDA's Point-of-View?

- $L_{palindrome\#} = \{p\#p' \mid p \in \{a,b\}^* , p'=reverse(p)\}$

- $L_{even\text{-}palindrome} = \{pp' \in \{a,b\}^* \mid p'=reverse(p)\}$

e.g., aa#aa vs aaaa

Determinism

- A PDA is **deterministic** if it never has multiple choices for which move to take

- A PDA P = (Q, Σ, Γ, δ, q_0, S, F) is **deterministic** if
  - $\forall q \in Q, a \in \Sigma \cup \{\epsilon\}, X \in \Gamma$:
    - $|\delta(q, a, X)| \leq 1$
    - If $\delta(q, a, X) \neq \emptyset$ then $\delta(q, \epsilon, X) = \emptyset$

Question: What's the (Qualitative) Difference from a PDA's Point-of-View?

- $L_{palindrome\#} = \{p\#p' \mid p \in \{a,b\}^* , p'=reverse(p)\}$

  Can be accepted by a DPDA

- $L_{even\text{-}palindrome} = \{pp' \in \{a,b\}^* \mid p'=reverse(p)\}$

  Cannot be accepted by a DPDA

DPDA and Regular Languages

- **Theorem**: If L is regular then $L=L(\mathrm{P})$ for some DPDA P accepting with final states

- **Proof**: Build a $\mathrm{P} = (Q, \Sigma, \Gamma, \delta, q_0, S, F)$ such that (Q, Σ, q_0, F) accepts L and $\Gamma = \{ Z \}$

DPDA: Final State vs. Empty Stack

- **Claim**: If L=L(P) for a DPDA P accepting with empty stack,
  then L = L(P') for some DPDA P' accepting with final state
  - $\leftarrow$ direction requires L to have the prefix property:
    No $\omega \in L$ is a prefix of another $\omega' \in L$
    - 0^* cannot be accepted with empty stack

DPDA and Context-Free Languages

- The languages accepted by PDPA by final state
  - Include all regular languages
  - A proper subset of context-free languages

DPDA and Ambiguous Grammars

- **Theorem**: If L is accepted by some DPDA then L has an unambiguous grammar

- There are languages with unambiguous grammar that no DPDA can accept
  - e.g., $S \rightarrow aSa \mid bSb \mid \epsilon$ is an unambiguous CFG

Summary

- CFGs
  - Languages
  - Derivations
    - Leftmost
    - rightmost
  - Parse trees
    - Yield
  - Ambiguity
    - Layering
- PDAs
  - Configurations & moves
  - Acceptance criteria
  - L(PDA)=L(CFG)
- DPDAs
  - Acceptance Criteria
  - Accepted Languages

Missing

- Pumping lemma for CFLs
- Converting CFL ⇔ PDA
- Algorithmic questions regarding CFLs
  - ☑ L(G) = ∅ ?
  - ☒ L(G) = Σ^* ?
  - ☒ $L(G_1) = L(G_2)$?
  - ☒ Is G ambiguous?
  - ☒ Is L inherently ambiguous?
- Closure properties of CFLs
  - ☑ Union
  - ☑ Concatenation
  - ☑ Intersection with regular languages
  - ☑ Homomorphism + inverse homomorphism
  - ☒ Intersection
  - ☒ Completion

Back to Compilation

Parsing

- Goals
  - Decide whether the sequence of tokens a valid program in the source language
  - Construct a structured representation of the input
    - Parse tree ≈ “Diagramming” the input
  - Error detection and reporting
- Challenges
  - How do you describe the programming language?
  - How do you check validity of an input?
  - Where do you report an error?

Checking Membership in CFLs

Checking Membership in CFLs

- Challenge: Given a CFG *G* and a string ω, decide if $\omega \in L(G)$

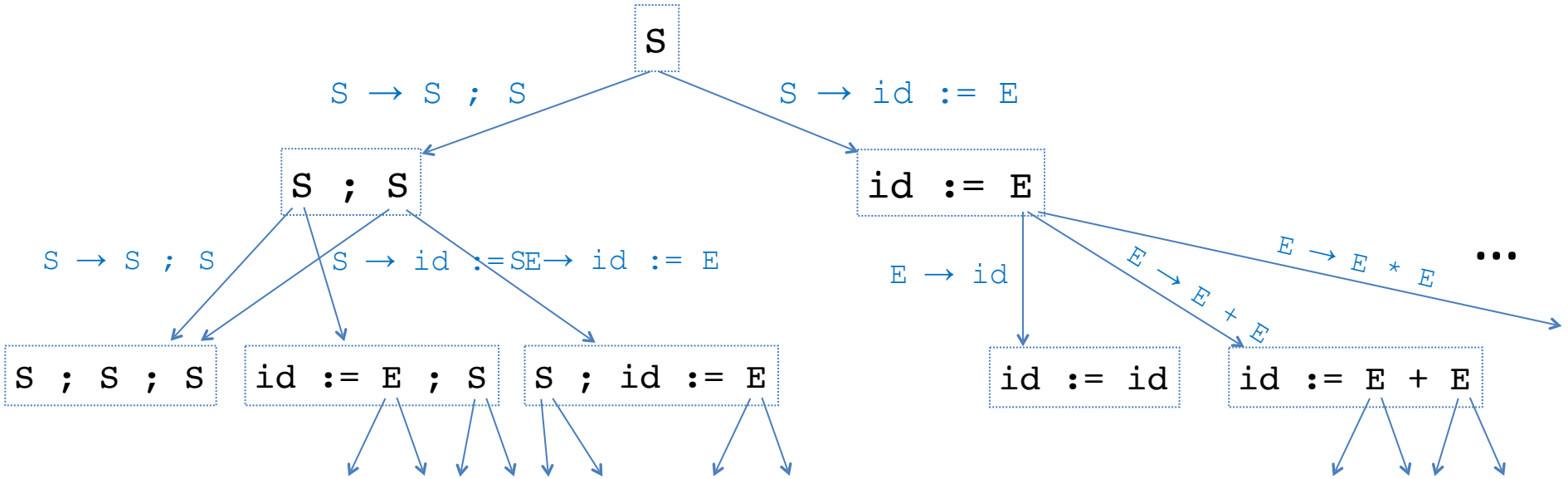
Solution I: “Brute-force” Parsing

- Parsing can be seen as a **search problem**
  - Can you find a derivation from the start symbol to the input word?
  - Possible (but very expensive) to solve with backtracking

“Brute-force” Parsing

```
x := z;
y := x + z
```

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```



(not a parse tree… a **search** for a parse by exhaustively applying all rules)

Problem: When to Stop?

- What if $\omega \notin L(G)$?

- Solution: Convert to **Chomsky Normal Form** (CNF)
  - A CFG G = ⟨V, T, P, S⟩ is in CNF if
    - The only ε-rule, if there is one, is $S \rightarrow \epsilon$
    - All derivation rules are:
      - $A \rightarrow BC$, where $B, C \in V$
      - $A \rightarrow t$, where $t \in T$

- Problem: Exponential search

Solution III: “DPDA”s

- We already know: Any context-free language can be recognized by a non-deterministic pushdown automaton

- We want ***efficient*** parsers that we can ***execute***
  - Deterministic computation model preferred
  - Linear in input size

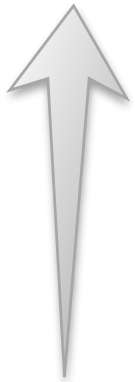
⇒ **We will sacrifice generality for efficiency**

⇒ **Use grammars for which we can build “DPDA”**

Efficient Parsers

- Top-down (predictive)
  - Construct the leftmost derivation
  - Apply rules “from left to right”
  - Predict what rule to apply based on nonterminal and token

- Bottom up (shift-reduce)
  - Construct the rightmost derivation (in reverse)
  - Apply rules “from right to left”
  - Reduce a right-hand side of a production to its non-terminal



One More Thing

Lexical/Syntactic Analysis?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

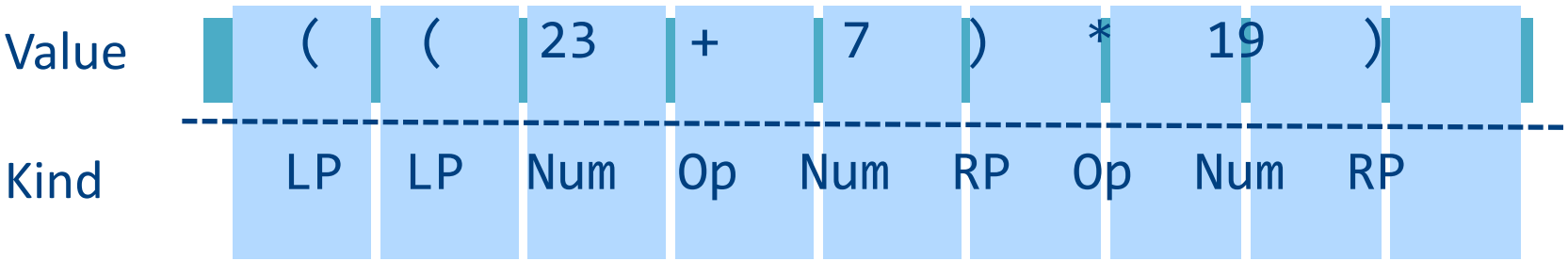
Dig → ‘0’|‘1’|‘2’|‘3’|‘4’|‘5’|‘6’|‘7’|‘8’|‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

Token Token …



Lexical/Syntactic Analysis?

Is Lexical analysis required?

- Well, not strictly necessary
  - Regular languages $\subseteq$ Context-Free languages

but

- Simplifies the syntax analysis (parsing)
  - And language definition
- Modularity
- Reusability
- Efficiency

Next …

- Top-Down Parsing
- Bottom-Up Parsing


EXTRA
EXTRA
EXTRA

Example

- $L_{\text{even-palindrome}} = \{pp' \mid p \in \{a,b\}^*, p'=\text{reverse}(p)\}$
  - E.g., ε, abbbba

- $G = \langle \{S\}, \{a,b\}, P, S \rangle$, where $P = S \rightarrow aSa \mid bSb \mid \varepsilon$

- **Claim** $L_{\text{even-palindrome}} = L(G)$
- Proof
  - $\subseteq$ by induction on the length of the word
  - $\supseteq$ by induction on the number of derivation steps

Proof

$L_{even\text{-}palindrome} \subseteq L(G)$

We prove by induction on the length of z, that $z \in L_{even\text{-}palindrome} \Rightarrow z \in L(G)$

Base: $|z|=0$
Since $(S \to \varepsilon) \in P$, it holds that $z=\varepsilon \in L(G)$

Inductive step:
Let $z = ww^R \in L_{even\text{-}pali\ldots}$ be a word of size 2k
Assume $z = \sigma w'$ $(\sigma \in \{a, b\})$
Hence, $z = \sigma w''(w'')^R\sigma$ (since $z = ww^R$)
By i.h. $S \Rightarrow^* w'(w')^R$
Hence, $S \Rightarrow \sigma S\sigma \Rightarrow^* \sigma w'(w')^R\sigma = z$

$L_{even\text{-}palindrome} \supseteq L(G)$

We prove by induction on the number of derivation steps used to derive z,
that $z \in L(G) \Rightarrow z \in L_{polyndrom}$

Base (Single derivation): The only possible derivation is $S \to \varepsilon$, and indeed $\varepsilon \in L_{even\text{-}pali\ldots}$

Inductive step:
Assume $S \Rightarrow^* z$ in $k > 1$ derivations steps.
First derivation is $S \Rightarrow \sigma S\sigma$ for $\sigma \in \{a, b\}$
Hence $z = \sigma z'\sigma$, and z' is derived from S using $k-1$ steps
By i.h., $z' = w'(w')^R$
Hence, $z = \sigma w'(w')^R\sigma = ww^R$ for $w = \sigma w'$ is an even-palindrome

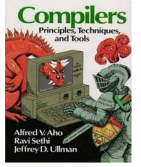
w^R=reverse(w)

$S \longrightarrow aSa \mid bSb \mid \varepsilon$

Context Insensetivity

- Let $G = \langle V, T, P, S \rangle$ be a CFG
- Theorem: If $A \Longrightarrow^* \beta$ then $\gamma A \delta \Longrightarrow^* \gamma\beta\delta$
- Lemma: If $\alpha \Longrightarrow^* \beta$ then $\mu\alpha\delta \Longrightarrow^* \mu\beta\delta$
  - Proof by induction on length of derivation
    - Basis: Trivial
    - Induction step: if $\alpha \Longrightarrow \alpha_1 \Longrightarrow^n \beta$ then
      $\alpha = \alpha' A \alpha'' \Longrightarrow \alpha_1 = \alpha'\gamma\alpha''$ for some $A \longrightarrow \gamma \in P$.
      By definition $\mu\alpha\delta = \mu\alpha' A \alpha'' \delta \Longrightarrow \mu\alpha'\gamma\alpha''\delta = \mu\alpha_1\delta$.
      As $\alpha_1 \Longrightarrow^n \beta$, by IH, $\mu\alpha_1\delta \Longrightarrow^* \mu\beta\delta$.
      Thus, $\mu\alpha\delta \Longrightarrow^* \mu\beta\delta$

"dangling-else" example



p. 212

Ambiguous grammar

S → if E then S

S | if E then S else S

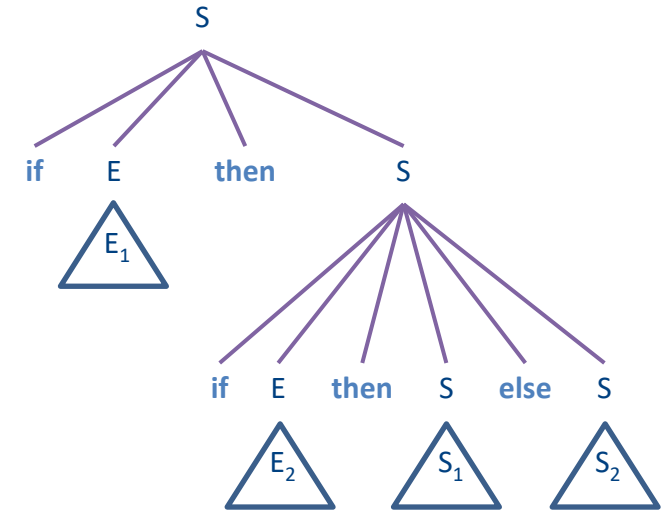
| other

Unambiguous grammar

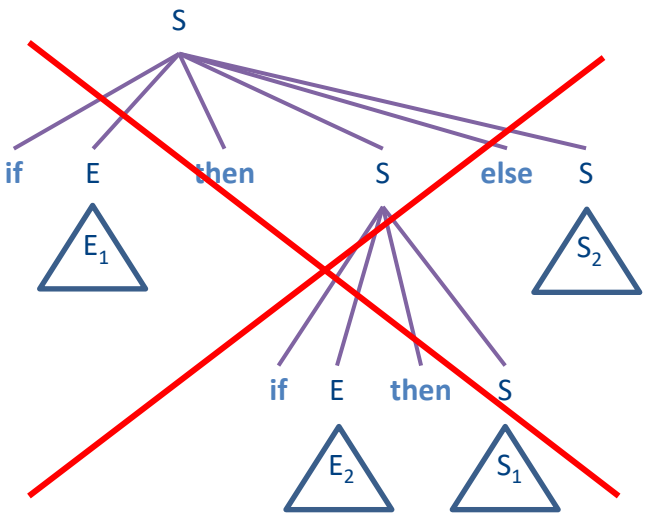
$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

if E_1 then if E_2 then S_1 else S_2

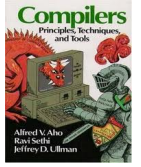
if E_1 then (if E_2 then S_1 else S_2)



if E_1 then (if E_2 then S_1) else S_2



“dangling-else” example



p. 212

Ambiguous grammar

S → if E then S

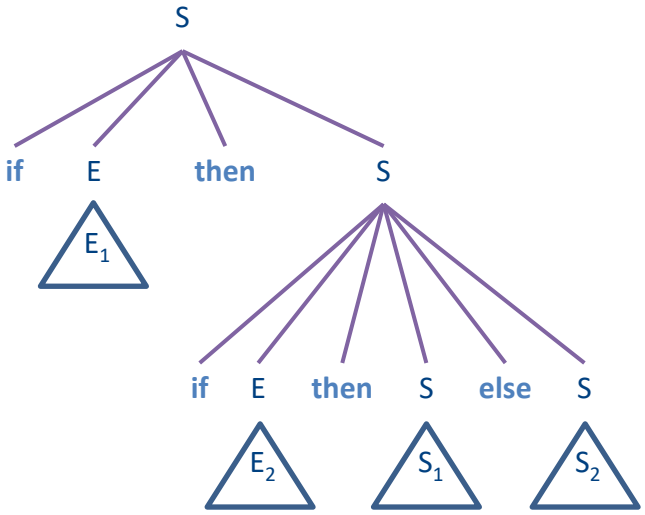
S | if E then S else S

| other

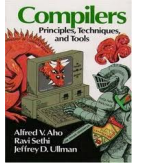
Unambiguous grammar

stmt	→	*matched_stmt*
	|	*open_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	**other**
open_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

if E_1 then (if E_2 then S_1 else S_2)



“dangling-else” example



p. 212

Ambiguous grammar

S → if E then S

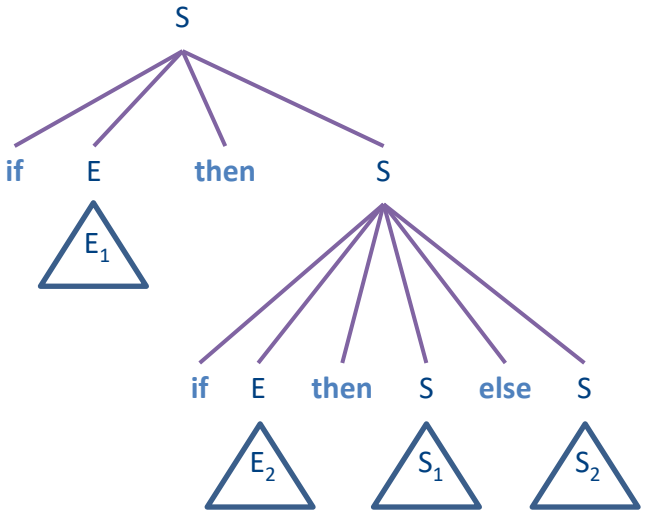
S | if E then S else S

| other

Unambiguous grammar

stmt	→	*matched_stmt*
	|	*open_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	**other**
open_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

if E_1 then (if E_2 then S_1 else S_2)



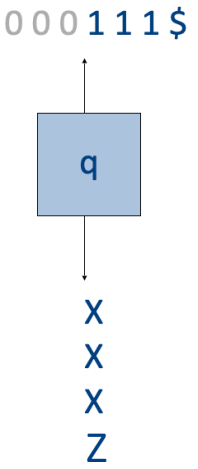
Example 4.16: We can rewrite the dangling-else grammar (4.14) as the following unambiguous grammar. The idea is that a statement appearing between a **then** and an **else** must be "matched"; that is, the interior statement must not end with an unmatched or open **then**. A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement. Thus, we may use the grammar in Fig. 4.10. This grammar generates the same strings as the dangling-else grammar (4.14), but it allows only one parsing for string (4.15); namely, the one that associates each **else** with the closest previous unmatched **then**. □

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \\
 & | & \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{matched\_stmt} \\
 & | & \textbf{other} \\
\textit{open\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{open\_stmt}
\end{array}
$$

Formal Model of Computation

- Let P = (Q, Σ, Γ, δ, q_0, S, F) be a PDA
- A **configuration** of P is a triple $(q, \omega, \sigma) \in Q \times \Gamma^* \times \Sigma^*$
  - q : the current state
  - ω : the remaining input
  - σ : the contents of the stack
- A configuration is an **instantaneous description** of the computation

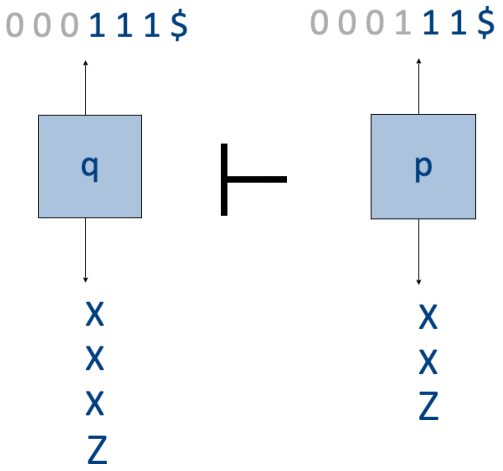
$(q, 111\$, XXXZ)$



Formal Model of Computation

- Let P = (Q, Σ, Γ, δ, q_0, S, F) be a PDA
- A **configuration** of P is a triple $(q, \omega, \sigma) \in Q \times \Gamma^* \times \Sigma^*$
- A configuration $(q, a\omega, X\sigma)$ can **move** to $(p, \omega, \alpha\sigma)$, denoted by $(q, a\omega, X\sigma) \vdash_P (p, \omega, \alpha\sigma)$, if $(p, \alpha) \in \delta(q, a, X)$
  - possibly, a=ε

$$(q,111\$, XXXZ) \vdash (p,11\$, XXZ)$$

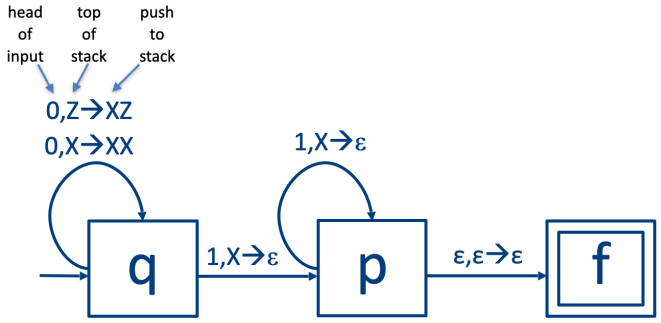


Formal Model of Computation

- Let P = (Q, Σ, Γ, δ, q_0, S, F) be a PDA

by final state

- P **accepts** $\omega \in \Sigma^*$ if $(q_0, \omega, S) \vdash^* (q, \varepsilon, \sigma)$ for some $\sigma \in \Gamma^*$ and $q \in F$
  - $\vdash^*$ is the reflexive transitive closure of $\vdash$
- The **language** of P is $L(P) = \{\omega \in \Sigma^* \mid P \text{ accepts } \omega\}$



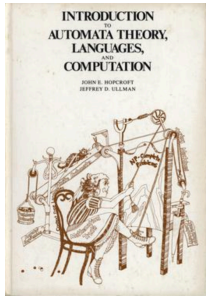
- Alt. P **accepts** $\omega \in \Sigma^*$ if $(q_0, \omega, S) \vdash^* (q, \varepsilon, \varepsilon)$

by empty stack

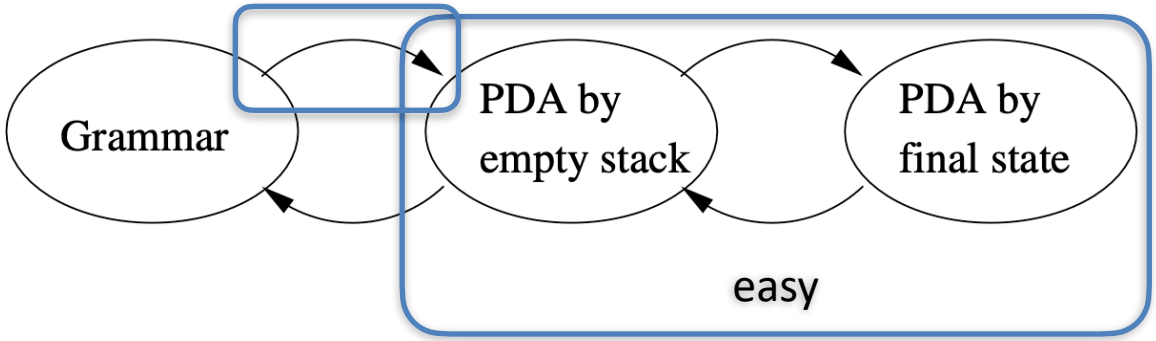
Equivalence of PDA's and CFG's

The following three classes of languages are the same

- Context-free languages (i.e., languages defined by a CFG)
- The languages that are accepted by final state for some PDA
- The languages that are accepted by empty stack for some PDA



p. 243



From CFL to PDA

- Theorem: if $S \Rightarrow_{lm}^{*} \alpha \Rightarrow_{lm}^{+} \omega$

  then $\alpha = \omega' \underbrace{A\beta}_{\text{tail}}$ and $\omega = \omega'\omega''$

- Example:
  - G = E → E + E | **num**
  - ω = **num + num + num**

Leftmost derivation

E $\Rightarrow_{lm}$

E + E $\Rightarrow_{lm}$

num + E $\Rightarrow_{lm}$

num + E + E $\Rightarrow_{lm}$

num + num + E $\Rightarrow_{lm}$

num + num + num

From CFL to PDA

- Idea: Simulate leftmost derivation
  - $S \Rightarrow_{lm}^{*} \alpha = \omega' A \beta \Rightarrow_{lm}^{+} \omega = \omega' \omega''$
  - $\omega' A \beta \rightsquigarrow (q, \omega'', A\beta)$

- Example:
  - G = E → E + E | **num**
  - ω = **num + num + num**

$(q, num + num, E + E)$

$(q, num + num, num + E)$

(q, num, E)

Leftmost derivation

|E ⇒lm

|E + E ⇒lm

num +|E ⇒lm

num +|E + E ⇒lm

num + num +|E ⇒lm

num + num + num|

From CFL to PDA

- For $G = \langle V, T, Q, S \rangle$, build a PDA
  $P = (\{q\}, T, V \cup T, \delta, q, S)$ accepting by empty stack
  - For each $A \in V$,
    $$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \in Q\}$$
  - For each $a \in T$,
    $$\delta(q, a, a) = \{(q, \epsilon)\}$$

- Example: G = E ⟶ E **+** E | **num**
  $$P = (\{q\}, \{+, \mathbf{num}\}, \{+, \mathbf{num}, E\}, \delta, q, E)$$
  - $\delta(q, \epsilon, E) = \{(q, E + E)\}, \delta(q, \epsilon, E) = \{(q, \mathbf{num})\}$
  - $\delta(q, +, +) = \{(q, \epsilon)\}, \delta(q, \mathbf{num}, \mathbf{num}) = \{(q, \epsilon)\}$

From CFL to PDA

- For $G = \langle V, T, Q, S \rangle$, build a PDA
  $P = (\{q\}, T, V \cup T, \delta, q, S)$ accepting by empty stack
  - For each $A \in V$,
    $$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \in Q\}$$
  - For each $a \in T$,
    $$\delta(q, a, a) = \{(q, \epsilon)\}$$

- Theorem: *L*(P)= *L*(G)

  → if $S \Rightarrow_{lm}^{n} \omega' A\beta$ then $(q, \omega, S) \vdash^{*} (q, \omega'', A\beta)$

  ← if $(q, \omega, A) \vdash^{*} (q, \epsilon, \epsilon)$ then $A \Rightarrow_{lm}^{*} \omega$ then

PROOF: We shall prove that w is in $N(P)$ if and only if w is in $L(G)$.

(If) Suppose w is in $L(G)$. Then w has a leftmost derivation

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$$

We show by induction on i that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, where y_i and α_i are a representation of the left-sentential form γ_i. That is, let α_i be the tail of γ_i, and let $\gamma_i = x_i\alpha_i$. Then y_i is that string such that $x_iy_i = w$; i.e., it is what remains when x_i is removed from the input.

BASIS: For $i = 1$, $\gamma_1 = S$. Thus, $x_1 = \epsilon$, and $y_1 = w$. Since $(q, w, S) \overset{*}{\vdash} (q, w, S)$ by 0 moves, the basis is proved.

INDUCTION: Now we consider the case of the second and subsequent left-sentential forms. We assume

$$(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$$

and prove $(q, w, S) \overset{*}{\vdash} (q, y_{i+1}, \alpha_{i+1})$. Since α_i is a tail, it begins with a variable A. Moreover, the step of the derivation $\gamma_i \Rightarrow \gamma_{i+1}$ involves replacing A by one of its production bodies, say β. Rule (1) of the construction of P lets us replace A at the top of the stack by β, and rule (2) then allows us to match any terminals on top of the stack with the next input symbols. As a result, we reach the ID $(q, y_{i+1}, \alpha_{i+1})$, which represents the next left-sentential form γ_{i+1}.

To complete the proof, we note that $\alpha_n = \epsilon$, since the tail of γ_n (which is w) is empty. Thus, $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$, which proves that P accepts w by empty stack.

(Only-if) We need to prove something more general: that if P executes a sequence of moves that has the net effect of popping a variable A from the top of its stack, without ever going below A on the stack, then A derives, in G, whatever input string was consumed from the input during this process. Precisely:

- If $(q, x, A) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon)$, then $A \overset{*}{\underset{G}{\Rightarrow}} x$.

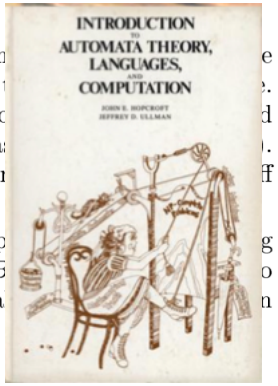
The proof is an induction on the number of moves taken by P.

BASIS: One move. The only possibility is that $A \to \epsilon$ is a production of G, and this production is used in a rule of type (1) by the PDA P. In this case, $x = \epsilon$, and we know that $A \Rightarrow \epsilon$.

INDUCTION: Suppose P takes n moves, where $n > 1$. The first move must be of type (1), where A is replaced by one of its production bodies on the top of the stack. The reason is that a rule of type (2) can only be used when there is a terminal on top of the stack. Suppose the production used is $A \to Y_1Y_2 \cdots Y_k$, where each Y_i is either a terminal or variable.

The next $n - 1$ moves of P must consume x from
net effect of popping each of Y_1, Y_2, and so on from
We can break x into $x_1x_2 \cdots x_k$, where x_1 is the portic
until Y_1 is popped off the stack (i.e., the stack first is a
Then x_2 is the next portion of the input that is consu
the stack, and so on.

Figure 6.9 suggests how the input x is broken up
effects on the stack. There, we suggest that β was B
three parts $x_1x_2x_3$, where $x_2 = a$. Note that in genera
x_i must be that terminal.



p. 245

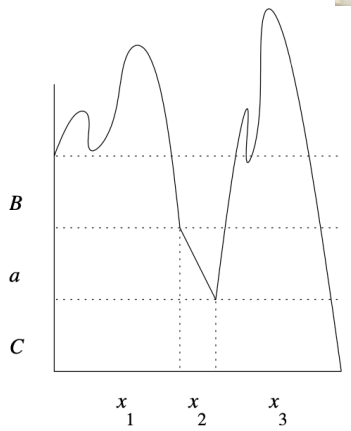


Figure 6.9: The PDA P consumes x and pops BaC from its stack

Formally, we can conclude that $(q, x_ix_{i+1} \cdots x_k, Y_i) \overset{*}{\vdash} (q, x_{i+1} \cdots x_k, \epsilon)$ for all $i = 1, 2, \ldots, k$. Moreover, none of these sequences can be more than $n - 1$ moves, so the inductive hypothesis applies if Y_i is a variable. That is, we may conclude $Y_i \overset{*}{\Rightarrow} x_i$.

If Y_i is a terminal, then there must be only one move involved, and it matches the one symbol of x_i against Y_i, which are the same. Again, we can conclude $Y_i \overset{*}{\Rightarrow} x_i$; this time, zero steps are used. Now we have the derivation

$$A \Rightarrow Y_1Y_2 \cdots Y_k \overset{*}{\Rightarrow} x_1Y_2 \cdots Y_k \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} x_1x_2 \cdots x_k$$

That is, $A \overset{*}{\Rightarrow} x$.

To complete the proof, we let $A = S$ and $x = w$. Since we are given that w is in $N(P)$, we know that $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$. By what we have just proved inductively, we have $S \overset{*}{\Rightarrow} w$; i.e., w is in $L(G)$. $\square$