Parsing: Context-Free Languages and Grammars

CS 4100
Gordon Stewart
Ohio University

Lexing and Parsing

Lexers like ocamllex convert concrete source programs (which are really just strings of characters) into sequences of ***tokens***

Enabling Formal Language Technology

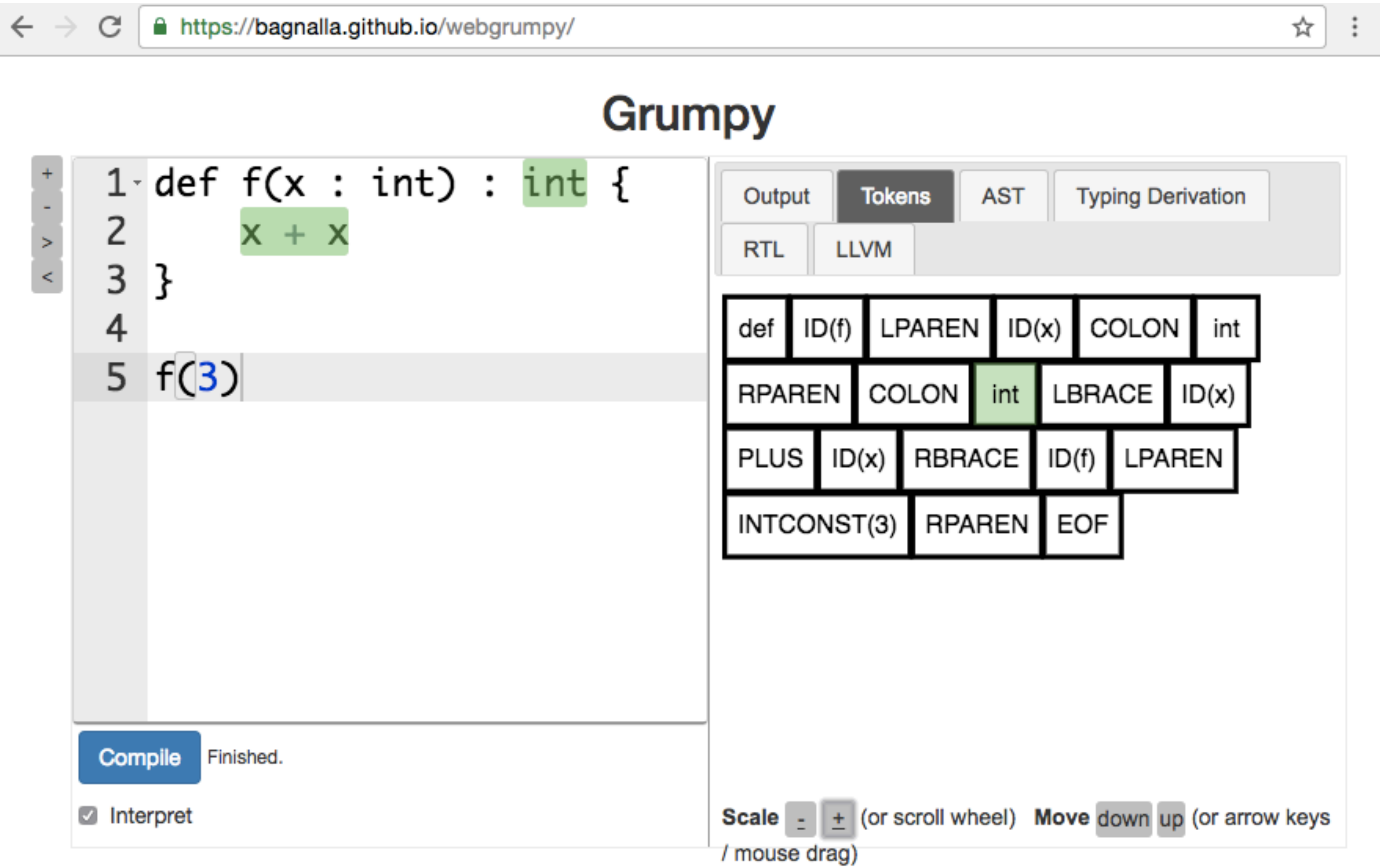
The reduction of **REs** to **NFAs** to **DFAs**

Parsers convert sequences of tokens into ***abstract syntax trees***

Enabling Formal Language Technology

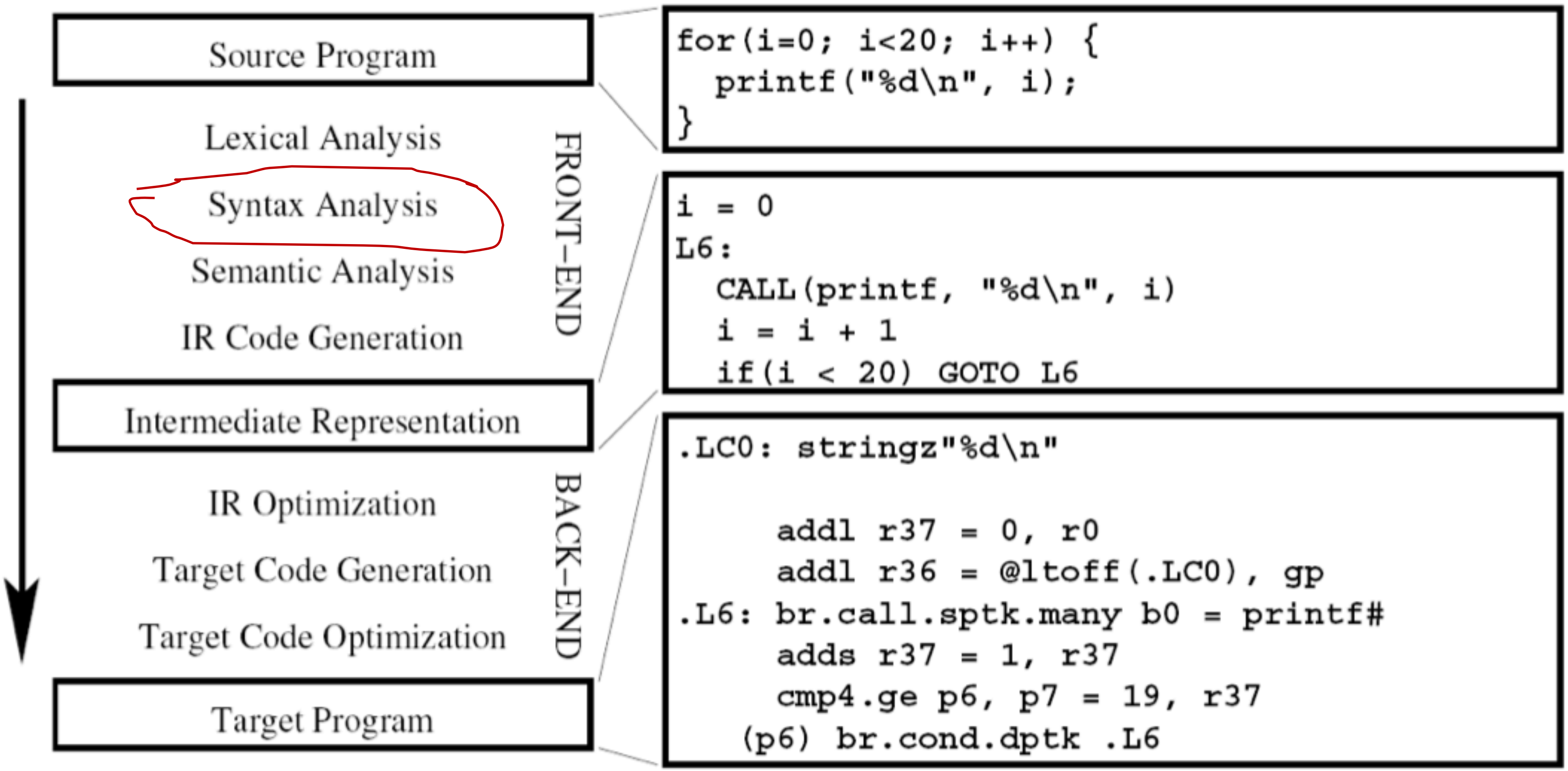
Context-Free Grammars (CFGs)

Lexing


https://bagnalla.github.io/webgrumpy/
Grumpy
def f(x : int) : int {
x + x
}
f(3)
Output
Tokens
AST
Typing Derivation
RTL
LLVM
def
ID(f)
LPAREN
ID(x)
COLON
int
RPAREN
COLON
int
LBRACE
ID(x)
PLUS
ID(x)
RBRACE
ID(f)
LPAREN
INTCONST(3)
RPAREN
EOF
Compile
Finished.
Interpret
Scale - + (or scroll wheel) Move down up (or arrow keys / mouse drag)

Parsing


https://bagnalla.github.io/webgrumpy/
Grumpy
1 def f(x : int) : int {
2 x + x
3 }
4
5 f(3)
Compile
Finished.
Interpret
Output
Tokens
AST
Typing Derivation
RTL
LLVM
fundefs: [●]m result: ●
fun f(x : TyInt) : TyInt {●}
ECall(f●)
EBinop(BPlus, ●, ●)
EInt(3)
EId(x)
EId(x)
Scale - + (or scroll wheel) Move left right down up (or arrow keys / mouse drag)


Source Program
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation
FRONT-END
Intermediate Representation
IR Optimization
Target Code Generation
Target Code Optimization
BACK-END
Target Program
for(i=0; i<20; i++) {
  printf("%d\n", i);
}
i = 0
L6:
  CALL(printf, "%d\n", i)
  i = i + 1
  if(i < 20) GOTO L6
.LC0: stringz"%d\n"
      addl r37 = 0, r0
      addl r36 = @ltoff(.LC0), gp
.L6: br.call.sptk.many b0 = printf#
      adds r37 = 1, r37
      cmp4.ge p6, p7 = 19, r37
   (p6) br.cond.dptk .L6

The Limits of Regular Languages

- A language is a set of strings over some alphabet
- The ***regular languages*** are those recognizable by regular expressions (or equivalently, NFAs or DFAs)

- Some languages are ***too complex*** to be recognized by regular expressions / DFAs / NFAs

The Limits of Regular Languages

- A language is a set of strings over some alphabet
- The ***regular languages*** are those recognizable by regular expressions (or equivalently, NFAs or DFAs)

- Some languages are ***too complex*** to be recognized by regular expressions / DFAs / NFAs

Example

Let L = the set of strings with a balanced number of left and right parentheses "(" and ")"

L = { "$(^n\ )^n$" | $n \in \{0, 1, 2, \ldots\}$ }

E.g., "()", "(( ))", "((( )))", …

The Limits of Regular Languages

- A language is a set of strings over some alphabet

Claim:
There is no DFA that matches L. Consequently, L is not regular.

Why:
A DFA with ***n*** states can remember parenthesis nesting depth no greater than ***n***.

Example

Let L = the set of strings with a balanced number of left and right parentheses "(" and ")"

L = { "$(^n\ )^n$" | $n \in \{0, 1, 2, \dots\}$ }

E.g., "()", "(( ))", "((( )))", …

Another Example

Let L = the set of strings with an equal number of **a**s and **b**s

E.g., “aabb”, “”, “abaabb”, …

The Limits of Regular Languages

Another Example

Let L = the set of strings with an equal number of **a**s and **b**s

E.g., “aabb”, “”, “abaabb”, …

Not regular: Intuitively, the corresponding DFA needs to “count” the number of **a**s, **b**s to ensure that they match, but only has a finite number of states…

The Limits of Regular Languages

Another Another Example

Let L = the set of strings with an equal number of occurrences of the substrings **ab** and **ba**

E.g., “abba”, “”, “bab”, “baab”, ...

The Limits of Regular Languages

Another Another Example

Let L = the set of strings with an equal number of occurrences of the substrings **ab** and **ba**

E.g., “abba”, “”, “bab”, “baab”, …

Regular!

Something to ponder…can you prove it?

The Pumping Lemma

- First proved by Michael Rabin and Dana Scott in 1959
  - http://www.cse.chalmers.se/~coquand/AUTOMATA/rs.pdf
- A general tool for proving that a language is ***nonregular***

Lemma: For any regular language ***L***, there is an integer p ≥ 1 (called the "pumping length") such that for any string ***s*** in ***L*** of length at least p, ***s*** can be broken up into three parts s = (x . y . z)

such that:

- length y >= 1
- length (x . y) <= p
- $\forall i \geq 0. \left(x \,.\, y^i \,.\, z\right) \in L$

Informally: within the first p characters in any string in L, there's a "loopable" substring y such that repeating y any number of times yields a string that's still in L.

Using Pumping Lemma to Prove a Language Nonregular

- **Claim:** The language that accepts n '('s followed by n ')' is nonregular.
  - $\boldsymbol{L} = \{\ (^n\ )^n \mid n \in [0..]\ \}$

- **Proof:**
  - Let p = pumping length of ***L***.
  - Let s = a string in ***L*** of length > 2p (thus s has p or more left parens, followed by p or more right parens)
  - By the pumping lemma, s is splittable into (x . y . z) such that length (x . y) <= p.
  - Thus y must contain only left parens.
  - But now the new string $x\ .\ y^i\ .\ z$ for any number i > 0 contains an unequal number of left and right parens.

CONTEXT-FREE LANGUAGES

Enter Context-Free

A context-free grammar matching the language

$$L = \{ “(^n\ )^n” \mid n \in \{0, 1, 2, \dots\} \}$$

S -> “(“ S “)”

S -> “”

Enter Context-Free

A context-free grammar matching the language

L = { $“(^{n}\ )^{n}”$ | $n \in \{0, 1, 2, \dots\}$ }

S -> “(“ S “)”

S -> “”

Every grammar is composed of a set of ***rules*** (also called ***productions)***

Enter Context-Free

A context-free grammar matching the language

$$L = \{ \text{“}(^n\ )^n\text{”} \mid n \in \{0, 1, 2, \dots\} \}$$

S -> “(“ S “)”

S -> “”

Every grammar is composed of a set of ***rules*** (also called ***productions)***

Every rule has a ***nonterminal*** symbol on the left…

And a sequence of ***terminal*** or ***nonterminal*** symbols on the right

Enter Context-Free

A context-free grammar matching the language

$$\text{L} = \{ \text{“}(^n\ )^n\text{”} \mid n \in \{0, 1, 2, \dots\} \}$$

Rules may be recursive

S -> “(“ S “)”

S -> “”

Every grammar is composed of a set of ***rules*** (also called ***productions)***

Every rule has a ***nonterminal*** symbol on the left…

And a sequence of ***terminal*** or ***nonterminal*** symbols on the right

Enter Context-Free

A context-free grammar matching the language

L = { "$(^n\ )^n$" | $n \in \{0, 1, 2, \dots\}$ }

Rules may be recursive

S -> "(" S ")"

S -> ""

Every grammar is composed of a set of ***rules*** (also called ***productions)***

One of the nonterminals is always designated the ***"start" symbol***

Every rule has a ***nonterminal*** symbol on the left…

And a sequence of ***terminal*** or ***nonterminal*** symbols on the right

When does a CFG match an input string?

The language of a CFG is defined as
the set of strings it derives.

A ***derivation*** begins from the "start" nonterminal, and at each step unfolds (replaces with RHS) one nonterminal according to the rules of the grammar.

(1) S -> "(" S ")"

(2) S -> ""

Derivation for "((( )))"

<u>S</u>	By rule 1

When does a CFG match an input string?

The language of a CFG is defined as
the set of strings it derives.

A ***derivation*** begins from the “start” nonterminal, and at each step unfolds (replaces with RHS) one nonterminal according to the rules of the grammar.

(1) S -> “(“ S “)”

(2) S -> “”

Derivation for “((( )))”

S	By rule 1
“(“ S “)”	By rule 1

When does a CFG match an input string?

The language of a CFG is defined as
the set of strings it derives.

A ***derivation*** begins from the "start" nonterminal, and at each step unfolds (replaces with RHS) one nonterminal according to the rules of the grammar.

(1) S -> "(" S ")"

(2) S -> ""

Derivation for "((( )))"

<u>S</u>	By rule 1
"(" <u>S</u> ")"	By rule 1
"((" <u>S</u> "))"	By rule 1

When does a CFG match an input string?

The language of a CFG is defined as
the set of strings it derives.

A ***derivation*** begins from the "start" nonterminal, and at each step unfolds (replaces with RHS) one nonterminal according to the rules of the grammar.

(1) S -> "(" S ")"

(2) S -> ""

Derivation for "((( )))"	
<u>S</u>	By rule 1
"(" <u>S</u> ")"	By rule 1
"((" <u>S</u> "))"	By rule 1
"(((" <u>S</u> ")))"	By rule 1

When does a CFG match an input string?

The language of a CFG is defined as
the set of strings it derives.

A ***derivation*** begins from the "start" nonterminal, and at each step unfolds (replaces with RHS) one nonterminal according to the rules of the grammar.

(1) S -> "(" S ")"

(2) S -> ""

Derivation for "((( )))"

S	By rule 1
"(" S ")"	By rule 1
"((" S "))"	By rule 1
"(((" S ")))"	By rule 1
"((( )))"	By rule 2

$CFL \supset RL$

Def: A language L is ***context-free*** iff there is a CFG that derives exactly the strings in L.

Thm: Every regular language is derivable by some CFG.

Examples

(a|b)*	**(a . b)***	**(a . b)+**
S ->	S ->	S -> abT
S -> aS	S -> abS	T ->
S -> bS		T -> abT

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

A terminal symbol matching integers ***n***

E -> E + E

E -> E * E

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

E -> E + E

E -> E * E

A terminal symbol matching integers ***n***

Derivation for "3 + 4 * 5$"

<u>S</u>

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

A terminal symbol matching integers ***n***

E -> E + E

E -> E * E

Derivation for "3 + 4 * 5$"

<u>S</u>

<u>E</u>$

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

A terminal symbol matching integers ***n***

E -> E + E

E -> E * E

Derivation for "3 + 4 * 5$"

<u>S</u>

<u>E</u>$

<u>E</u> * E$

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

A terminal symbol matching integers ***n***

E -> E + E

E -> E * E

Derivation for "3 + 4 * 5$"

<u>S</u>

<u>E</u>$

<u>E</u> * E$

<u>E</u> + E * E$

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

A terminal symbol matching integers ***n***

E -> E + E

E -> E * E

Derivation for "3 + 4 * 5$"

<u>S</u>

<u>E</u>$

<u>E</u> * E$

<u>E</u> + E * E$

3 + <u>E</u> * E$

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

A terminal symbol matching integers ***n***

E -> E + E

E -> E * E

Derivation for "3 + 4 * 5$"

S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$

When does a CFG match an input string?

S -> E $

"$": A terminal symbol matching end-of-file

E -> integer

A terminal symbol matching integers ***n***

E -> E + E

E -> E * E

Derivation for "3 + 4 * 5$"

<u>S</u>

<u>E</u>$

<u>E</u> * E$

<u>E</u> + E * E$

3 + <u>E</u> * E$

3 + 4 * <u>E</u>$

3 + 4 * 5$

PARSE TREES

Parse Trees

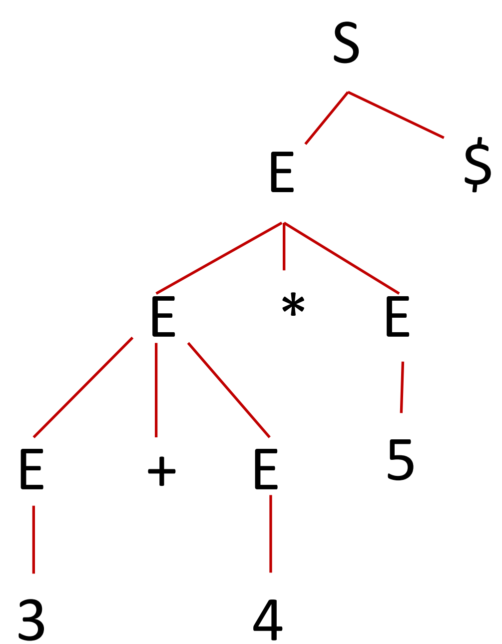
S -> E $

E -> integer

E -> E + E

E -> E * E

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.



Derivation for
"3 + 4 * 5$"

S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Parse Trees

S -> E $

E -> integer
E -> E + E
E -> E * E

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.

S

Derivation for "3 + 4 * 5$"

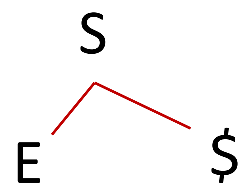
S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Parse Trees

S -> E $

E -> integer
E -> E + E
E -> E * E

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.



Derivation for "3 + 4 * 5$"

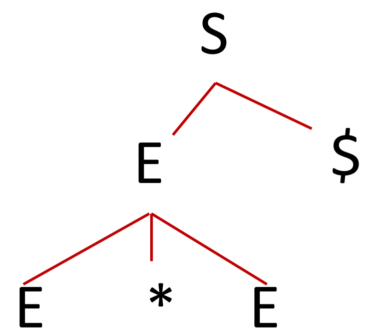
S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Parse Trees

S -> E $

E -> integer
E -> E + E
E -> E * E

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.



Derivation for "3 + 4 * 5$"

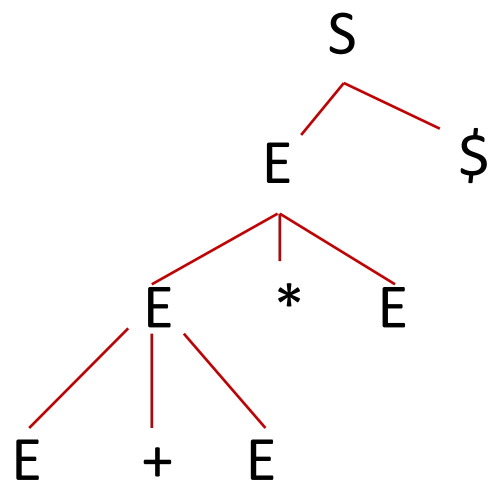
S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Parse Trees

S -> E $

E -> integer
E -> E + E
E -> E * E

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.



Derivation for "3 + 4 * 5$"

S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Parse Trees

S -> E $

E -> integer

E -> E + E

E -> E * E

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.

Derivation for "3 + 4 * 5$"

S

E$

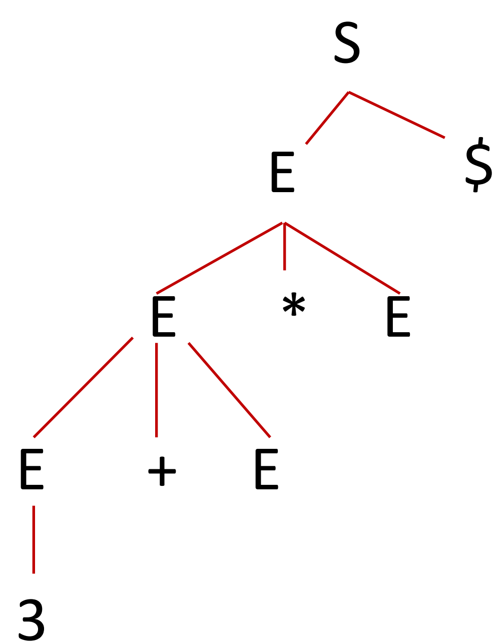
E * E$

E + E * E$

3 + E * E$

3 + 4 * E$

3 + 4 * 5$



Parse Trees

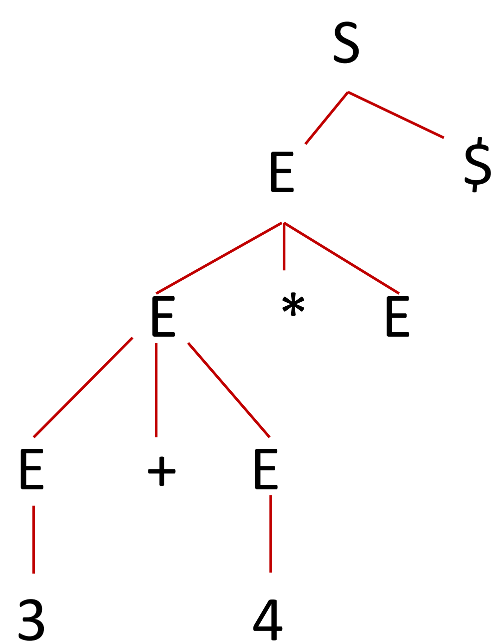
S -> E $

E -> integer

E -> E + E

E -> E * E

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.



Derivation for
"3 + 4 * 5$"

S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Parse Trees

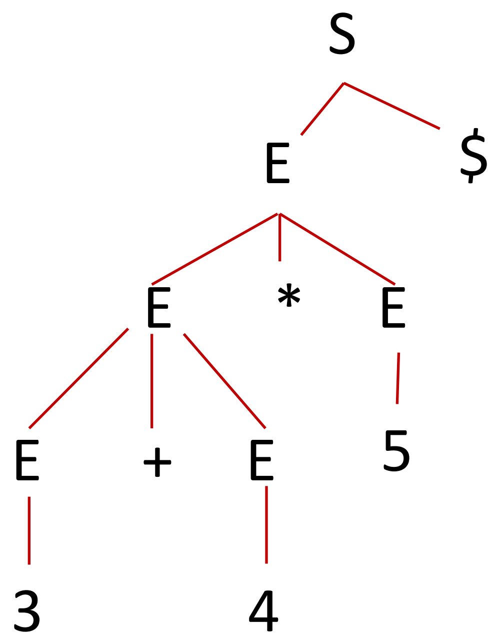
S -> E $

E -> integer
E -> E + E
E -> E * E

Derivation for "3 + 4 * 5$"

S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Every derivation corresponds to a parse tree, by connecting the terminal and nonterminal symbols to the nonterminals from which they were derived.



Leftmost Derivations

S -> E $

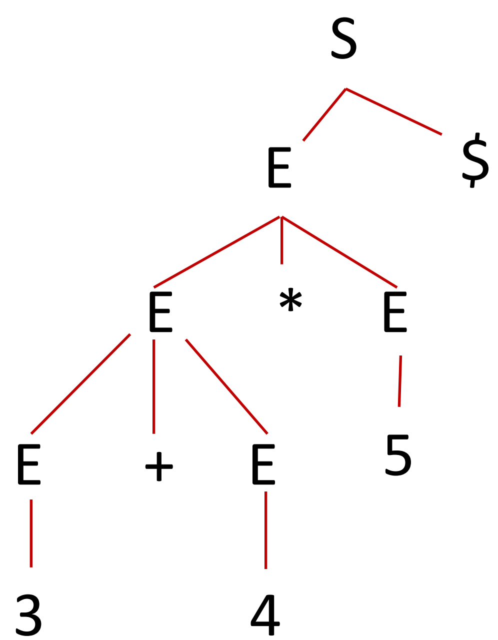
E -> integer
E -> E + E
E -> E * E

In the derivation at bottom-left, we always expanded the leftmost nonterminal symbol at each step.

This is a so-called ***leftmost derivation.***

Derivation for "3 + 4 * 5$"

S
E$
E * E$
E + E * E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

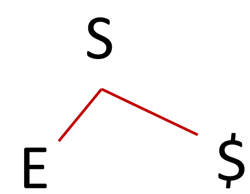


Rightmost Derivations

S -> E $

E -> integer
E -> E + E
E -> E * E

Here's an alternative "rightmost" derivation that generates the same parse tree…just in a different order!



Derivation for
"3 + 4 * 5$"

S
E$
E * E$
E * 5$
E + E * 5$
E + 4 * 5$
3 + 4 * 5$

Rightmost Derivations

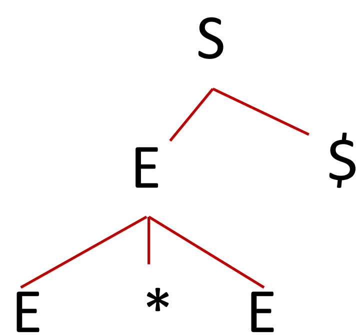
S -> E $

E -> integer

E -> E + E

E -> E * E

Here's an alternative "rightmost" derivation that generates the same parse tree…just in a different order!



Derivation for "3 + 4 * 5$"

S

E$

E * E$

E * 5$

E + E * 5$

E + 4 * 5$

3 + 4 * 5$

Rightmost Derivations

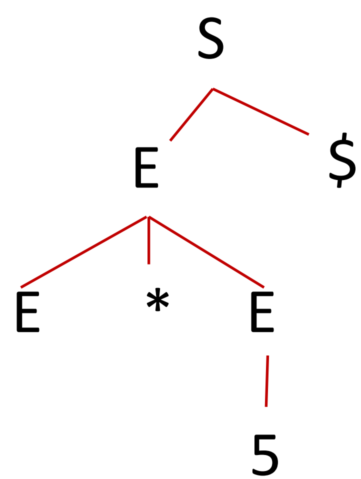
S -> E $

E -> integer

E -> E + E

E -> E * E

Here's an alternative "rightmost" derivation that generates the same parse tree…just in a different order!



Derivation for "3 + 4 * 5$"

S

E$

E * E$

E * 5$

E + E * 5$

E + 4 * 5$

3 + 4 * 5$

Rightmost Derivations

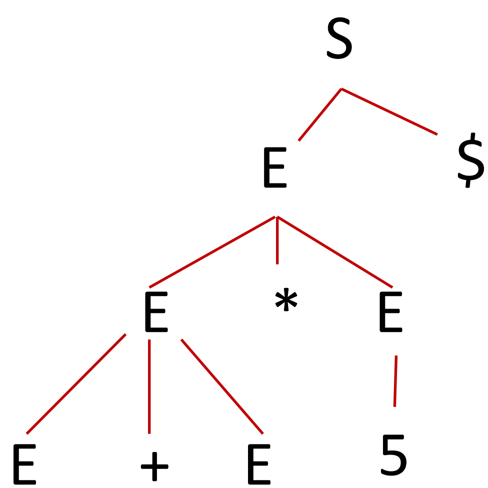
S -> E $

E -> integer

E -> E + E

E -> E * E

Here's an alternative "rightmost" derivation that generates the same parse tree...just in a different order!



Derivation for "3 + 4 * 5$"

S

E$

E * E$

E * 5$

E + E * 5$

E + 4 * 5$

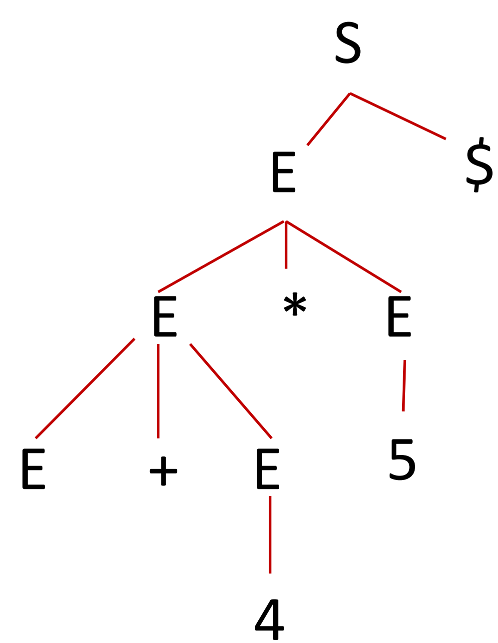
3 + 4 * 5$

Rightmost Derivations

S -> E $

E -> integer
E -> E + E
E -> E * E

Here's an alternative "rightmost" derivation that generates the same parse tree…just in a different order!



Derivation for "3 + 4 * 5$"

$\underline{\text{S}}$
$\underline{\text{E}}$\$
E * $\underline{\text{E}}$\$
$\underline{\text{E}}$ * 5\$
E + $\underline{\text{E}}$ * 5\$
$\underline{\text{E}}$ + 4 * 5\$
3 + 4 * 5\$

Rightmost Derivations

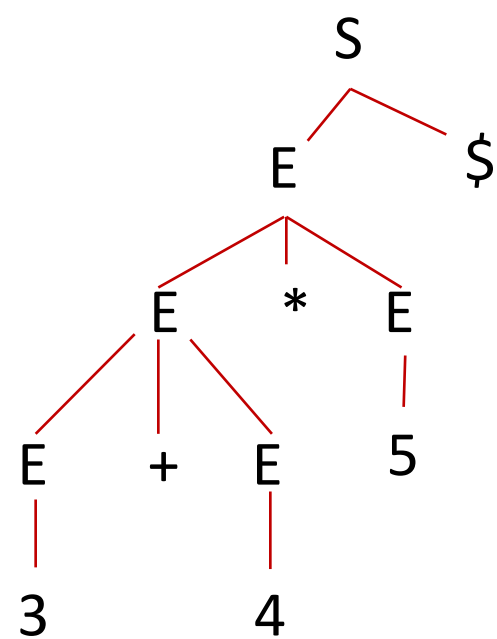
S -> E $

E -> integer

E -> E + E

E -> E * E

Here's an alternative "rightmost" derivation that generates the same parse tree…just in a different order!



Derivation for "3 + 4 * 5$"

S

E$

E * E$

E * 5$

E + E * 5$

E + 4 * 5$

3 + 4 * 5$


THURSDAY, MAY 31
SOUTHAMPTON
BRIDE TO BE
KILLED
BY BOY
RACERS
Daily Echo

AMBIGUITY


NO
PARKING
VIOLATORS
WILL
BE
TOWED

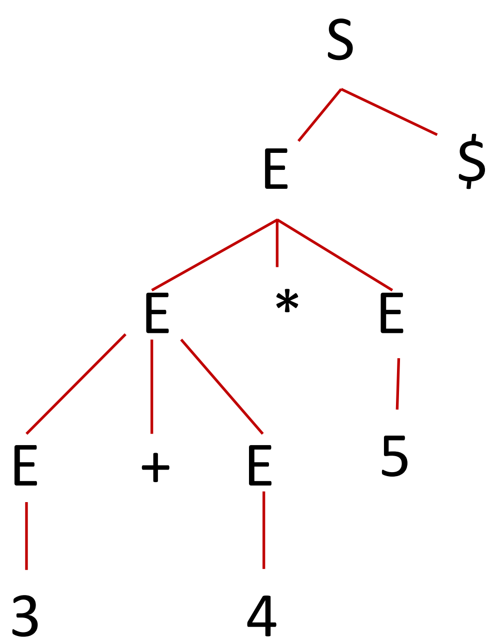
Ambiguous Grammars

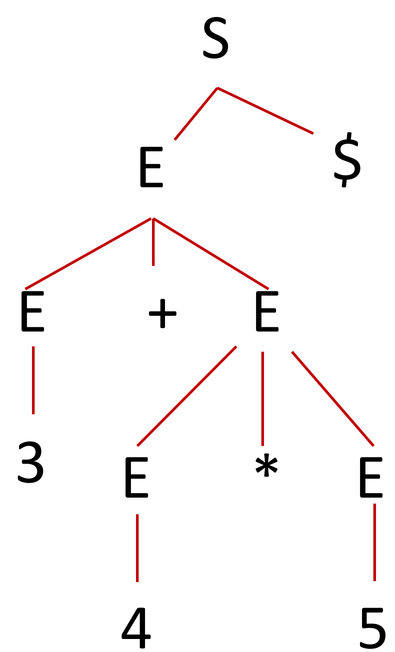
S -> E $
E -> integer
E -> E + E
E -> E * E

A grammar is ambiguous if it permits distinct parse trees for the same string.

"3 + 4 * 5$"

S
E$
E * E$
E * 5$
E + E * 5$
E + 4 * 5$
3 + 4 * 5$





"3 + 4 * 5$"

S
E$
E + E$
3 + E$
3 + E * E$
3 + 4 * E$
3 + 4 * 5$

Alternatively: The grammar admits two ***leftmost*** ***(or rightmost)*** derivations of the same string.

Disambiguating Grammars

An ambiguous grammar can often (but not always!) be reformulated as an equivalent unambiguous grammar.

Ambiguous

S -> E $

E -> integer
E -> E + E
E -> E * E
E -> (E)

Unambiguous

S -> E $

E -> E + T
E -> T

T stands for term

T -> T * F
T -> F

F stands for factor

F -> integer
F -> (E)

Precedence and Associativity

What's the result of the following arithmetic expression?

4 + 5 * 3 – 2

Precedence and Associativity

What's the result of the following arithmetic expression?

4 + 5 * 3 – 2

Ans: 4 + (5 * 3) – 2 = 4 + 1 5 – 2 = 17

Multiplication has higher precedence ("binds tighter") than addition and subtraction.

Precedence and Associativity

What’s the result of the following arithmetic expression?

4 + 5 * 3 – 2

Ans: 4 + (5 * 3) – 2 = 4 + 1 5 – 2 = 17

Multiplication has higher precedence (“binds tighter”) than addition and subtraction.

What about…

27 – 27 + 27

Precedence and Associativity

What's the result of the following arithmetic expression?

4 + 5 * 3 – 2

Ans: 4 + (5 * 3) – 2 = 4 + 1 5 – 2 = 17

Multiplication has higher precedence ("binds tighter") than addition and subtraction.

What about…

27 – 27 + 27

Ans: (27-27) + 27 = 0 + 27 = 27

Addition and subtraction have the same precedence (by convention) but are ***left associative.***

Disambiguating Grammars

Intuitively: Introduce a new **nonterminal** symbol for each precedence level (* has ***higher precedence/binds tighter*** than +)

S -> E $

E -> E + T

E -> T

A new nonterminal **T** for binary operators with higher precedence than (+)

T -> T * F

T -> F

A new nonterminal **F** for operators that have higher precedence than (*)

F -> integer

F -> (E)

Unambiguous

Disambiguating Grammars

S -> E $

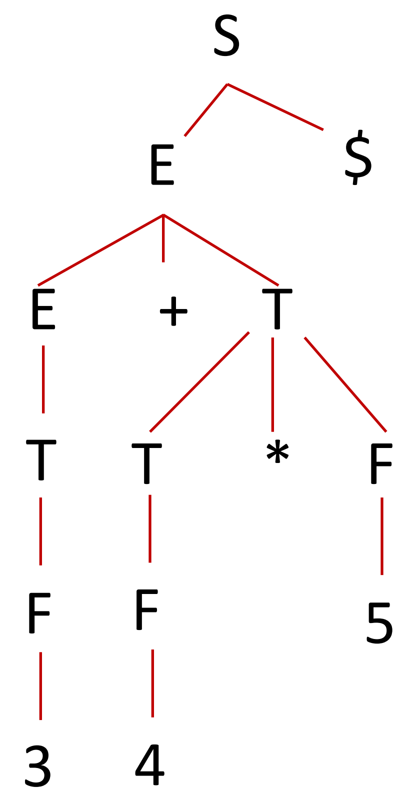
E -> E + T
E -> T

T -> T * F
T -> F

F -> integer
F -> (E)

Leftmost Derivation for "3 + 4 * 5$"

S
E$
E + T$
T + T$
F + T$
3 + T$
3 + T * F$
3 + F * F$
3 + 4 * F$
3 + 4 * 5$



Controlling Associativity

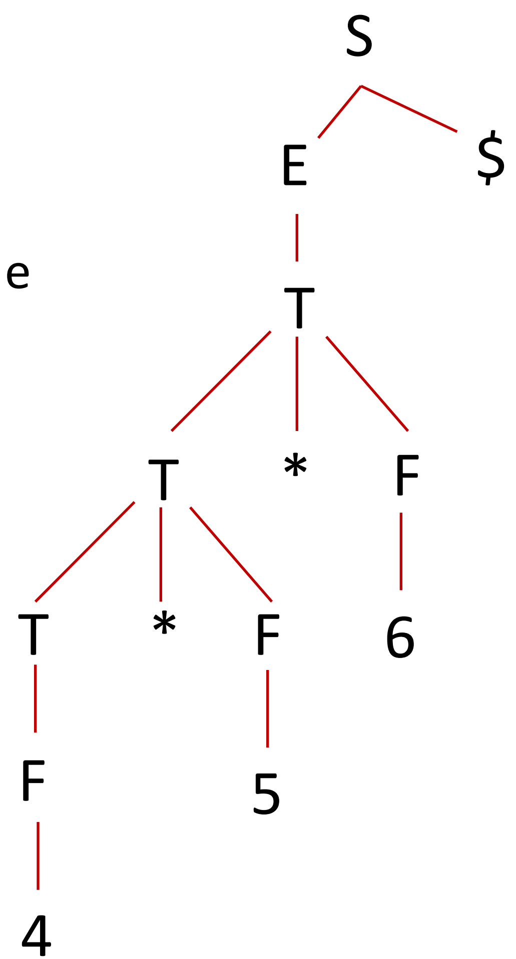
S -> E $

E -> E + T
E -> T

T -> T * F
T -> F

F -> integer
F -> (E)

Introducing **F** on the right enforces ***left associativity***



Controlling Associativity

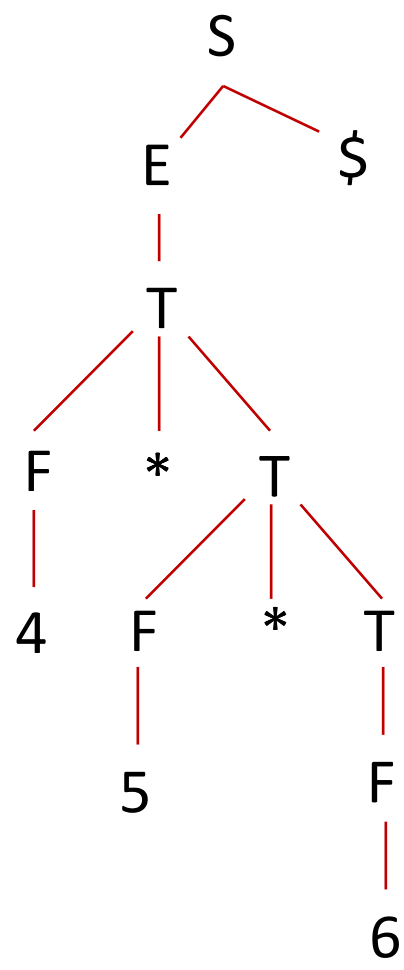
S -> E $

E -> E + T
E -> T

Now multiplication is right associative!

T -> **F * T**
T -> F

F -> integer
F -> (E)



Ambiguous Grammars

Another classic example:

S -> if E then S else S

S -> if E then S

S -> ...

E -> ...

if E then (if E then ...) else ...
if E then (if E then ... else ...)

This grammar is ambiguous for the string

if **E** then if **E** then **... else ...**

S
if E then S else ...
if E then ...

S
if E then S
if E then ... else ...

Disambiguating

S -> M (* for "matched" *)

S -> U (* for "unmatched" *)

M -> if E then M else M

U -> if E then S

U -> if E then M else U

if **E** then if **E** then **…** else **…**

S

U

if E then S

M

if E then … else …

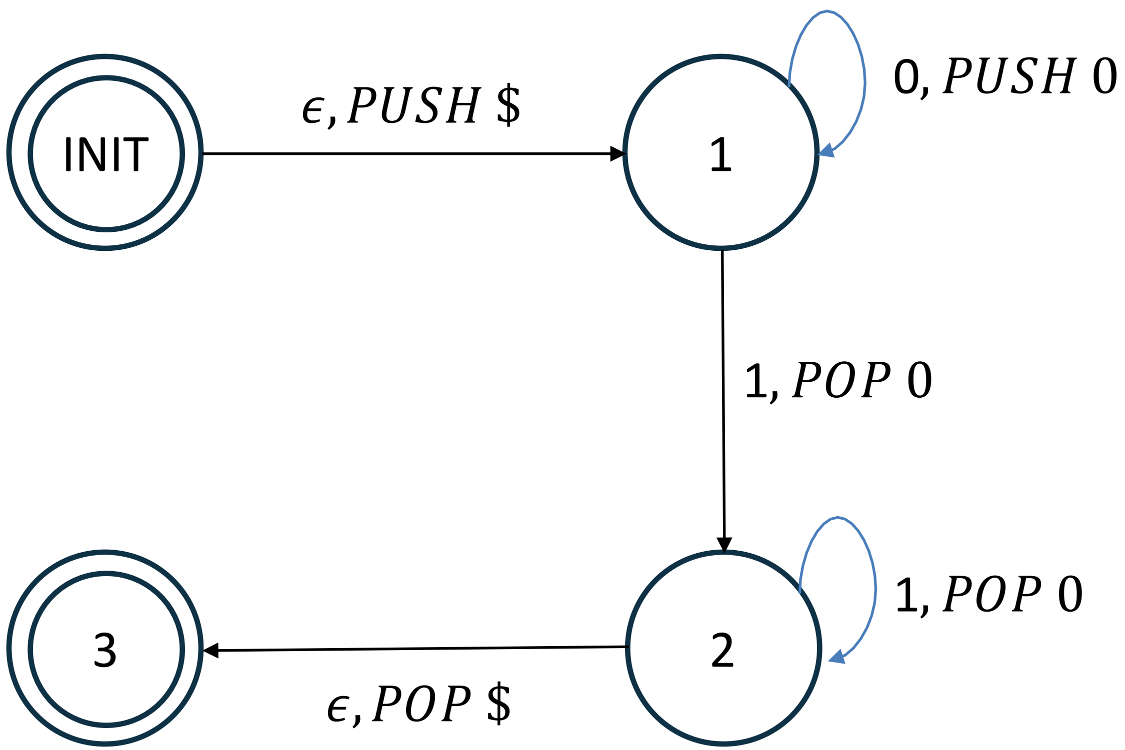
PUSHDOWN AUTOMATA

Pushdown Automata

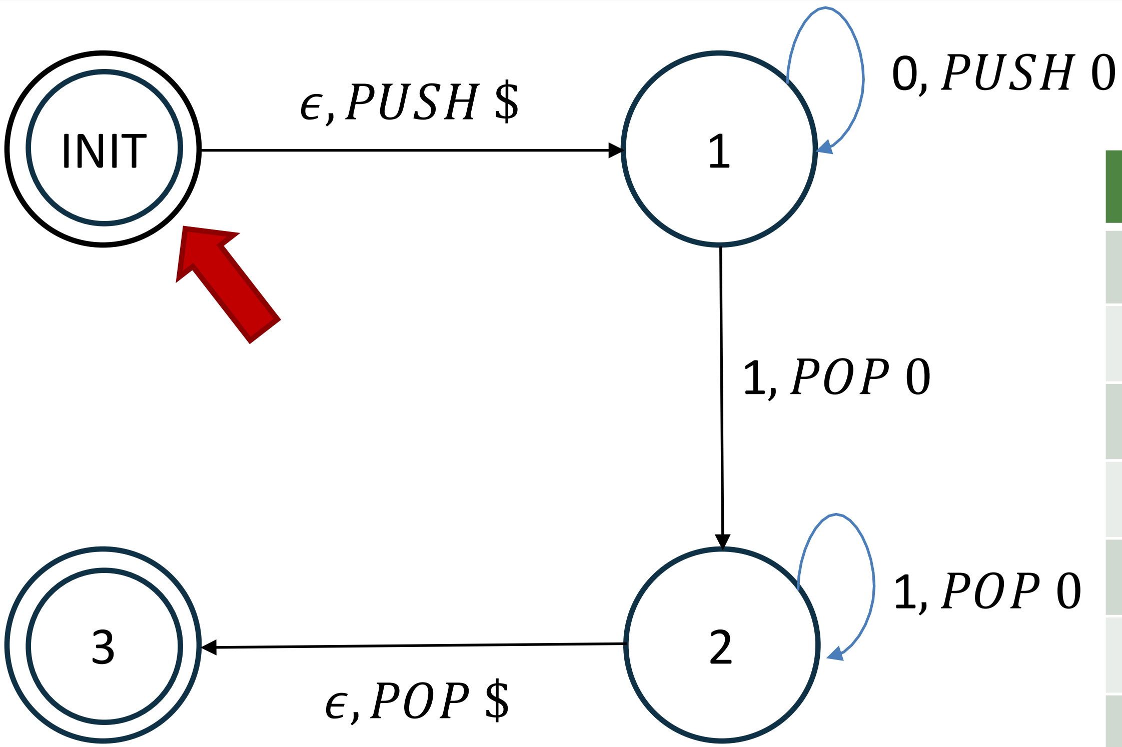
- ***Pushdown automaton (PDA):*** An NFA-like machine for recognizing whether strings are in a language which in addition has access to a ***stack***
  - In addition to reading symbols from the input and moving from state to state, PDA can also ***push*** and ***pop*** values onto an unbounded stack

- PDAs have equivalent power to CFGs:
  - The set of languages ***generatable*** by context-free grammars is exactly the set of languages ***recognizable*** by PDAs

PDA for L = $\{0^n 1^n \mid n \in [0 \dots]\}$

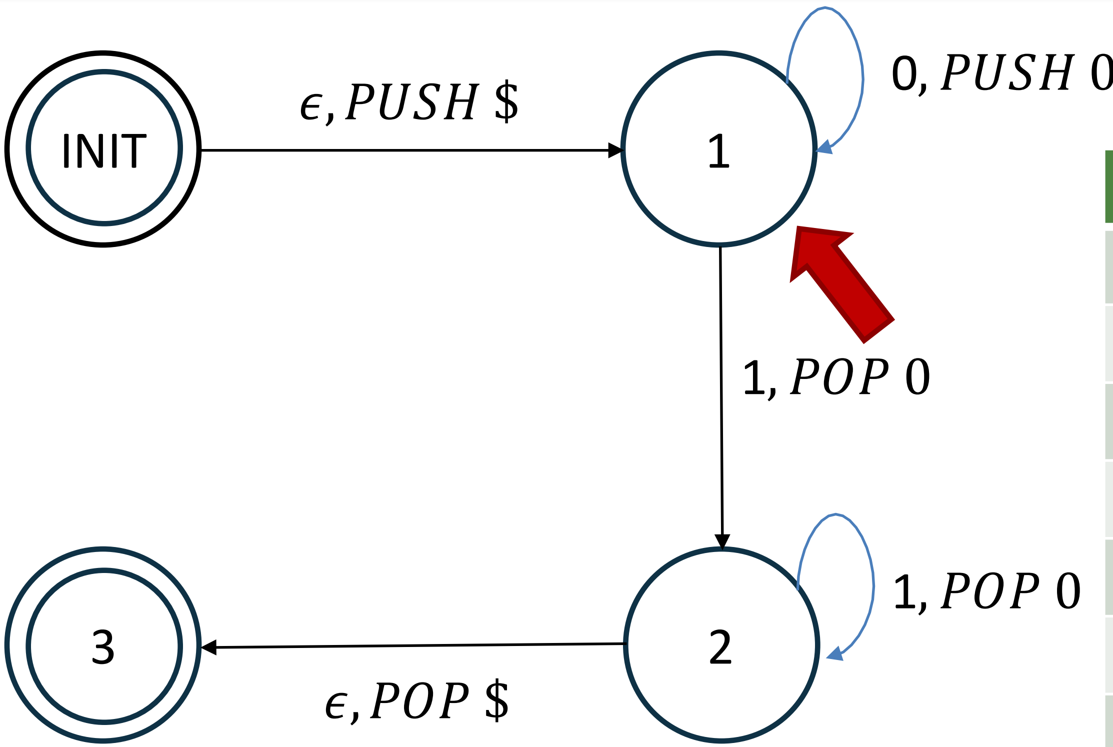


PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



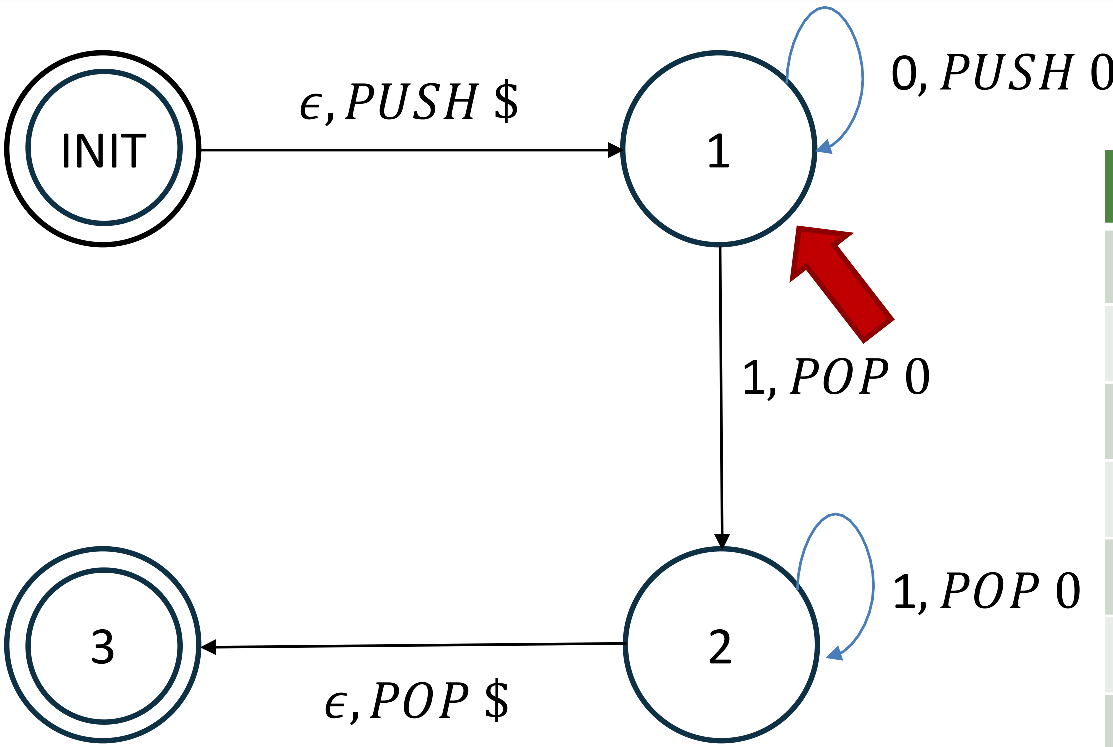
Stack	Input
	000111

PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



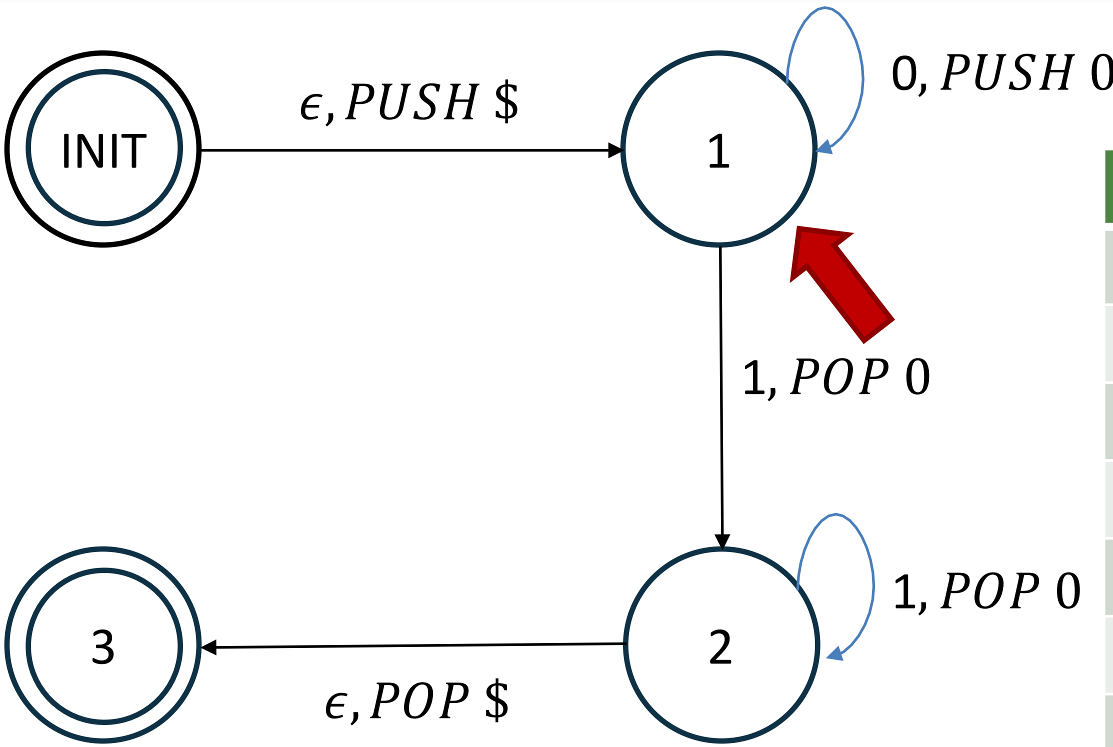
Stack	Input
	000111
$	000111

PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



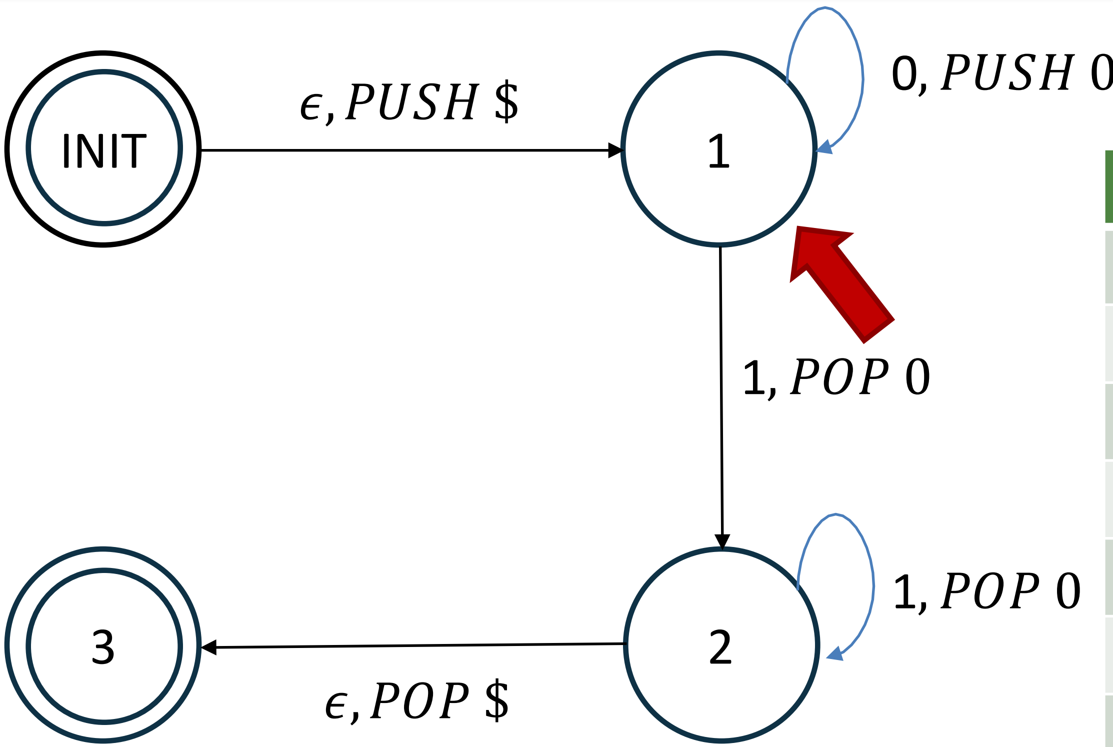
Stack	Input
	000111
$	000111
$0	00111

PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



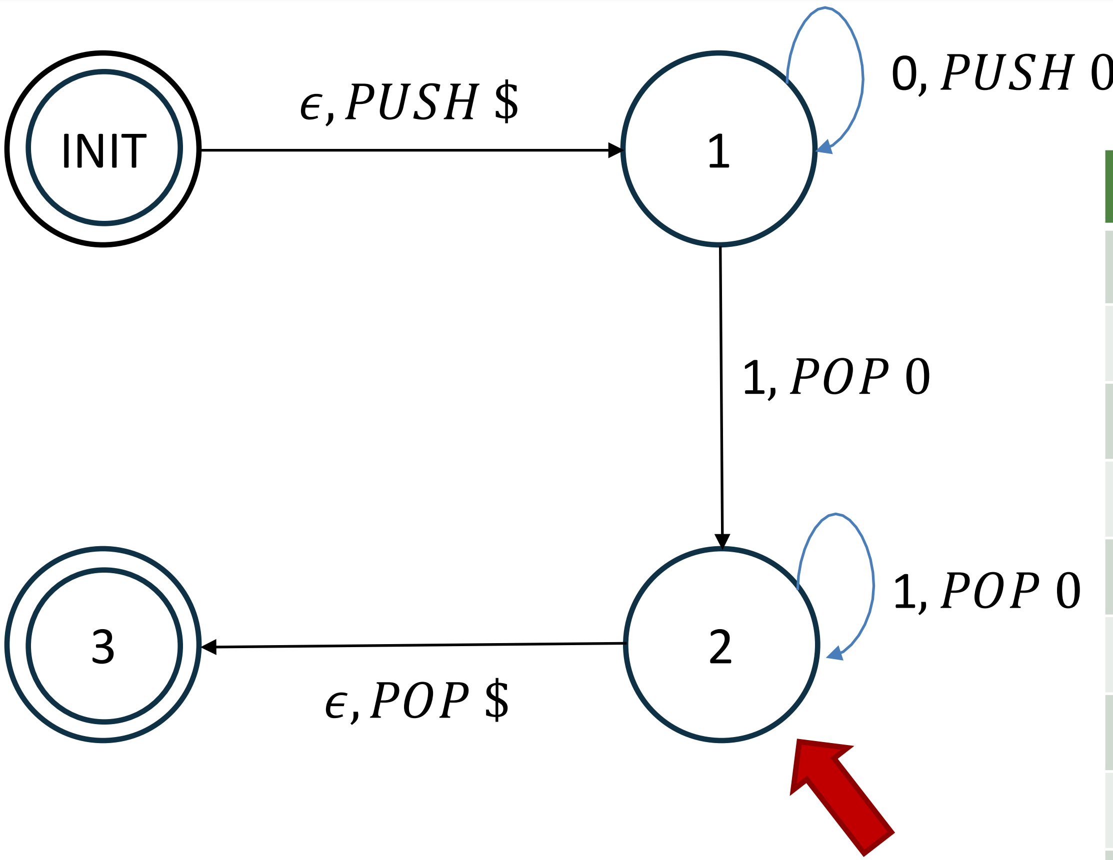
Stack	Input
	000111
$	000111
$0	00111
$00	0111

PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



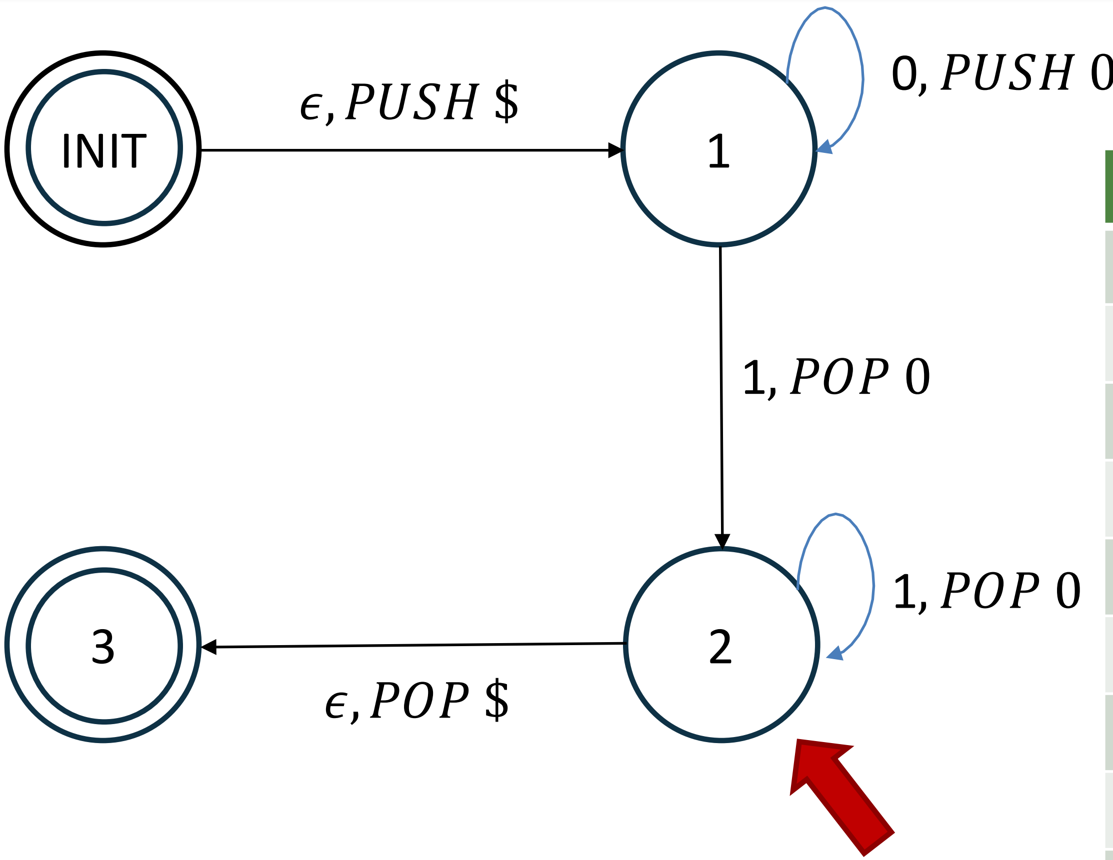
Stack	Input
	000111
$	000111
$0	00111
$00	0111
$000	111

PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



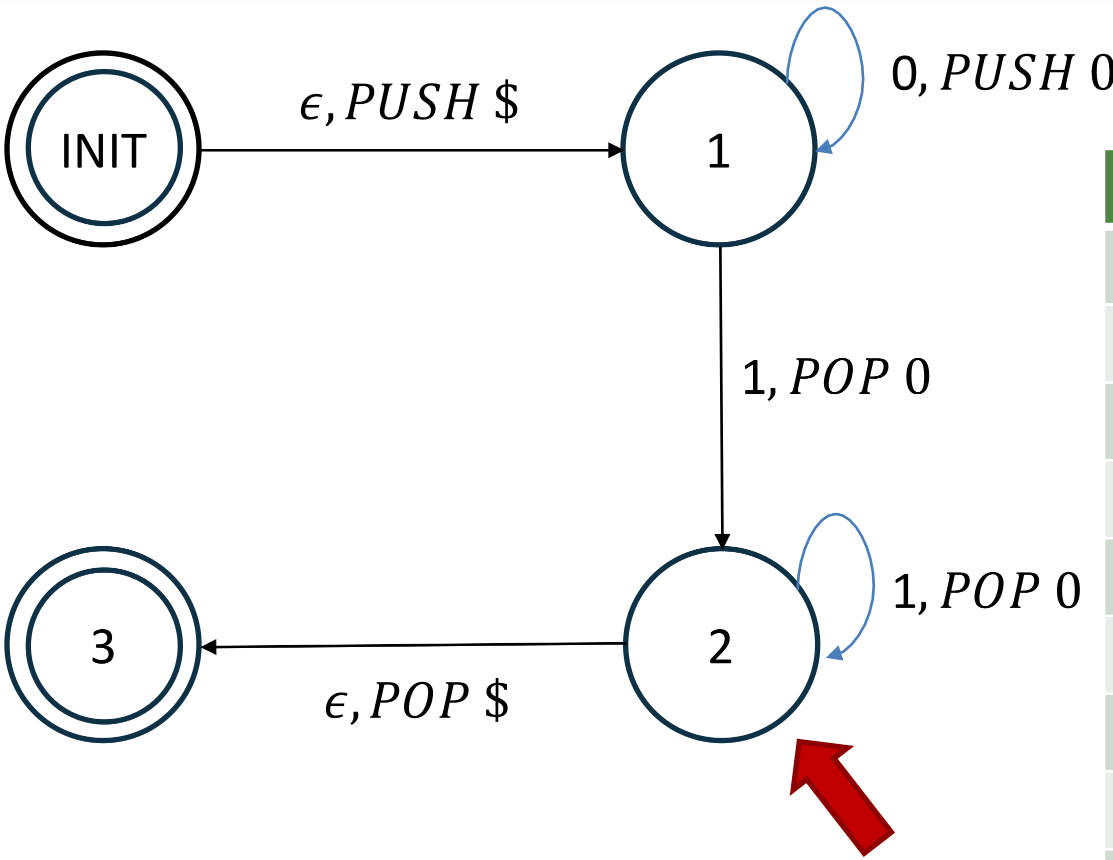
Stack	Input
	000111
$	000111
$0	00111
$00	0111
$000	111
$00	11

PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



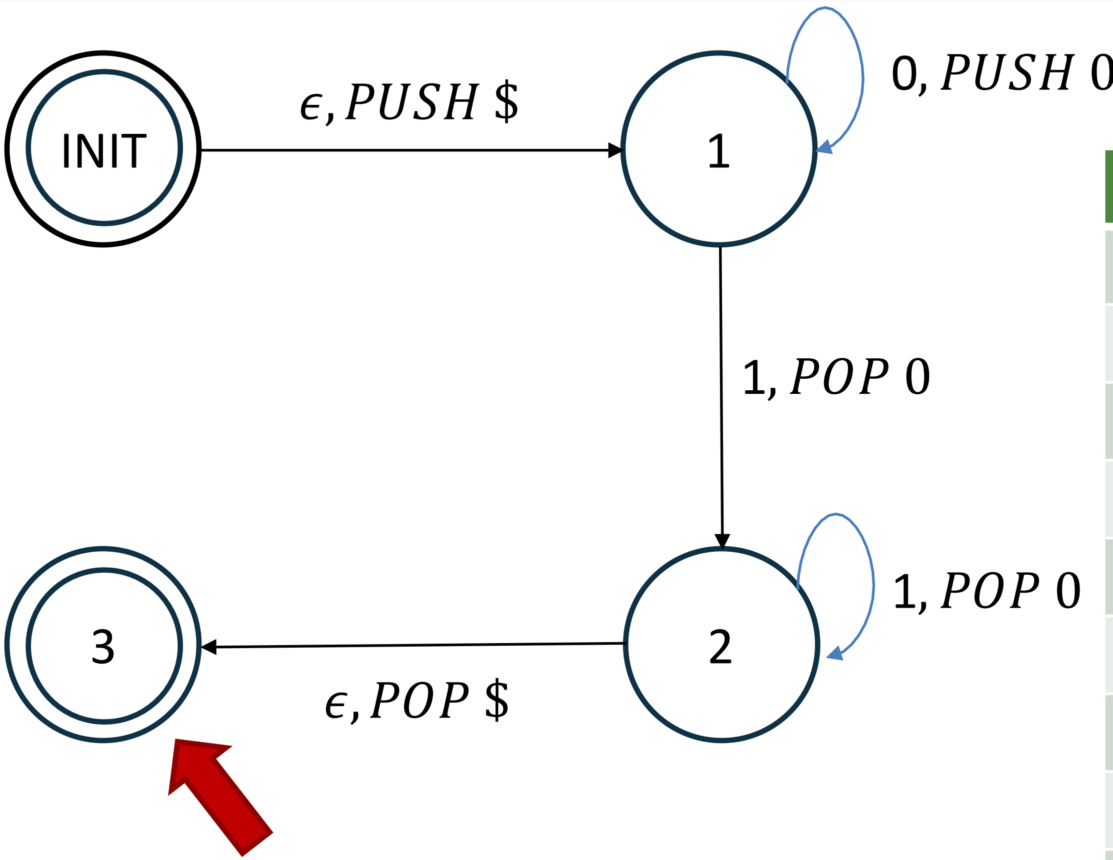
Stack	Input
	000111
$	000111
$0	00111
$00	0111
$000	111
$00	11
$0	1

PDA for L = $\{0^n 1^n \mid n \in [0 \ldots]\}$



Stack	Input
	000111
$	000111
$0	00111
$00	0111
$000	111
$00	11
$0	1
$	

PDA for L = $\{0^n 1^n \mid n \in [0 \dots]\}$



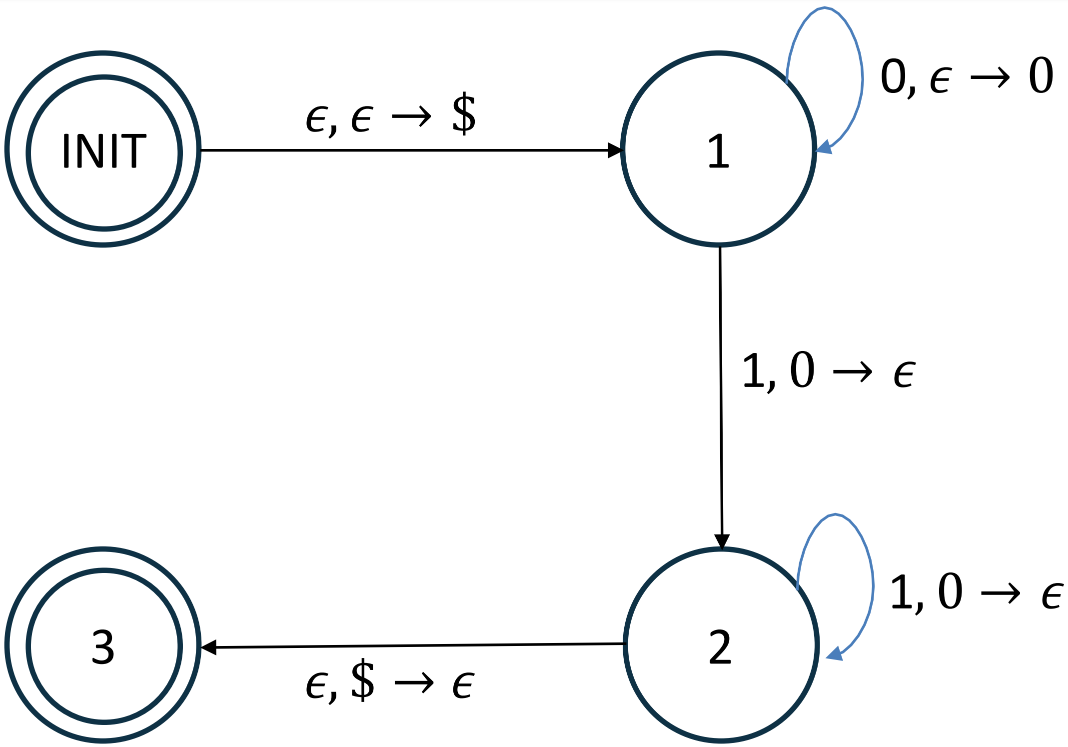
Stack	Input
	000111
$	000111
$0	00111
$00	0111
$000	111
$00	11
$0	1
$	

PDAs Formally

- A PDA consists of:
  - A finite set of states ***Q***
  - An alphabet $\Sigma \cup \{\epsilon\}$ of possible input characters
  - An alphabet $\Gamma \cup \{\epsilon\}$ of possible stack characters
  - An initial state INIT (drawn from ***Q***)
  - A set of FINAL states (a subset of ***Q***)

- together with a transition function δ
  - δ's inputs:
    - The current state $q \in Q$
    - The symbol to read, if any, drawn from $\Sigma \cup \{\epsilon\}$
    - The top symbol on the stack
  - δ's outputs:
    - $P(Q \times (\Gamma \cup \{\epsilon\}))$

PDAs Formally



- Q = {INIT,1,2,3}
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$
- INIT = INIT, FINAL = {INIT, 3}