Context-Free Grammars

Friday Four Square!

Today at 4:15PM, Outside Gates

Announcements

- Problem Set 5 due right now.
  - Due on **Tuesday** at 12:50PM if you're using a late day.
- Problem Set 6 out, due **Monday**, February 25 at 12:50PM.
  - Play around with PDAs, CFGs, and their limits!
- Problem Set 4 graded, will be returned at the end of lecture.
- Midterms should be graded by Wednesday.

Generation vs. Recognition

- We saw two approaches to describe regular languages:
  - Build **automata** that accept precisely the strings in the language.
  - Design **regular expressions** that describe precisely the strings in the language.
- Regular expressions **generate** all of the strings in the language.
  - Useful for listing off all strings in the language.
- Finite automata **recognize** all of the strings in the language.
  - Useful for detecting whether a specific string is in the language.

Context-Free Languages

- Last time, we saw the **context-free languages**, which are those that can be recognized by **pushdown automata**.
- Is there some way to build a system that can **generate** the context-free languages?

Context-Free Grammars

- A **context-free grammar** (or **CFG**) is an entirely different formalism for defining the context-free languages.
- CFGs are best explained by example...

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E** `*` `(`**E Op E**`)`

⇒ `int * (`**E Op E**`)`

⇒ `int * (int` **Op E**`)`

⇒ `int * (int` **Op** `int)`

⇒ `int * (int + int)`

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \mathbf{E\ Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int}\ \mathbf{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int / int}
\end{aligned}
$$

Context-Free Grammars

- Formally, a context-free grammar is a collection of four objects:
  - A set of **nonterminal symbols** (also called **variables**),
  - A set of **terminal symbols** (the **alphabet** of the CFG)
  - A set of **production rules** saying how each nonterminal can be converted by a string of terminals and nonterminals, and
  - A **start symbol** (which must be a nonterminal) that begins the derivation.

E → `int`

E → **E Op E**

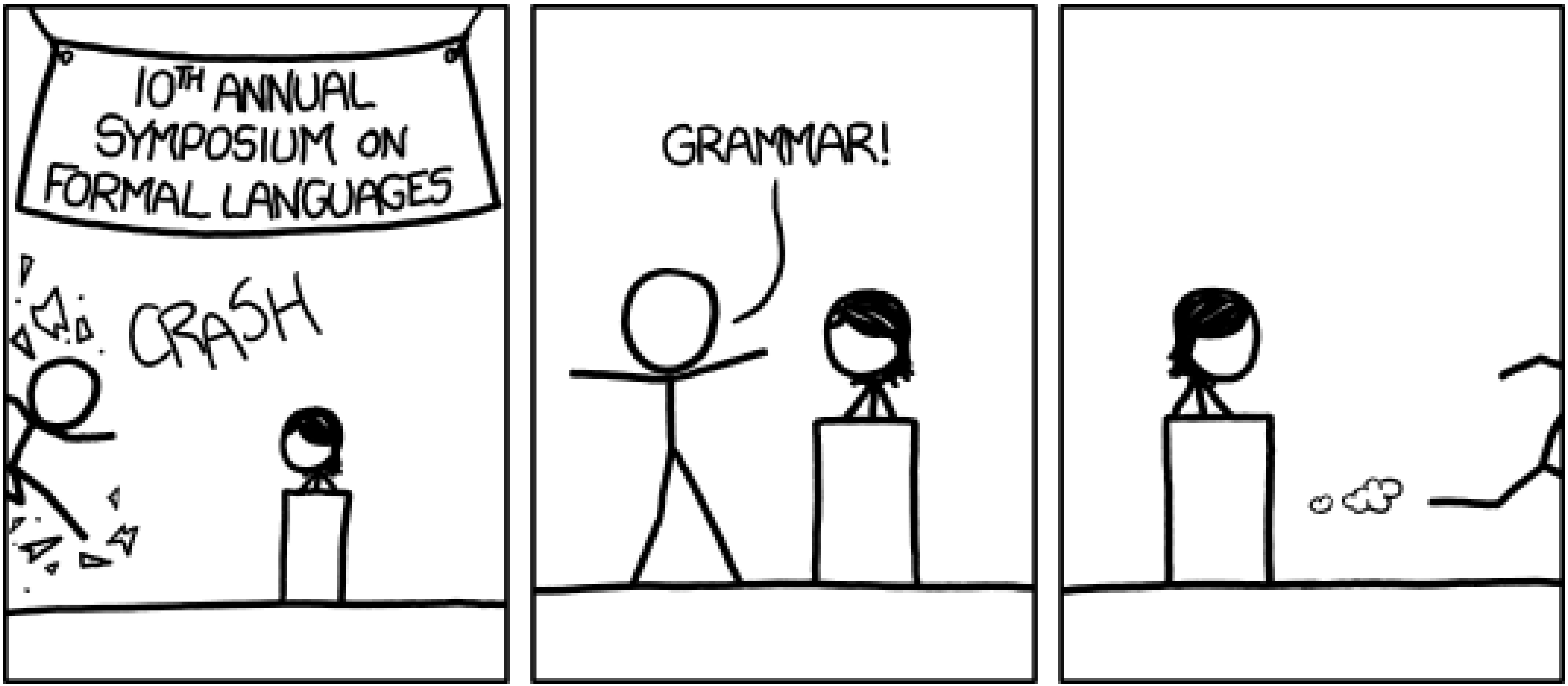
E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`



http://xkcd.com/1090/

A Notational Shorthand

$\mathbf{E} \rightarrow$ `int`

$\mathbf{E} \rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$

$\mathbf{E} \rightarrow$ `(`$\mathbf{E}$`)`

$\mathbf{Op} \rightarrow$ `+`

$\mathbf{Op} \rightarrow$ `-`

$\mathbf{Op} \rightarrow$ `*`

$\mathbf{Op} \rightarrow$ `/`

A Notational Shorthand

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \texttt{a*b}$$

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \mathbf{A}\mathtt{b}$$

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \mathbf{A}\mathtt{b}$$
$$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \varepsilon$$

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

S → `a(b|c*)`

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \texttt{a}\mathbf{X}$$

$$\mathbf{X} \rightarrow \texttt{(b|c*)}$$

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$

$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathtt{c}^*$$

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathbf{C}$$

From Regexes to CFGs

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathbf{C}$$
$$\mathbf{C} \rightarrow \mathbf{C}\mathtt{c} \mid \varepsilon$$

More Context-Free Grammars

- Chemicals!
  - $C_{19}H_{14}O_5S$
  - $Cu_3(CO_3)_2(OH)_2$
  - MnO_4^-
  - S^{2-}

Form → **Cmp** | **Cmp Ion**

Cmp → **Term** | **Term Num** | **Cmp Cmp**

Term → **Elem** | `(`**Cmp**`)`

Elem → `H` | `He` | `Li` | `Be` | `B` | `C` | …

Ion → `+` | `-` | **IonNum** `+` | **IonNum** `-`

IonNum → `2` | `3` | `4` | ...

Num → `1` | **IonNum**

CFGs for Chemistry

Form → **Cmp** | **Cmp Ion**

Cmp → **Term** | **Term Num** | **Cmp Cmp**

Term → **Elem** | `(`**Cmp**`)`

Elem → `H` | `He` | `Li` | `Be` | `B` | `C` | …

Ion → `+` | `-` | **IonNum** `+` | **IonNum** `-`

IonNum → `2` | `3` | `4` | ...

Num → `1` | **IonNum**

Form

⇒ **Cmp Ion**

⇒ **Cmp Cmp Ion**

⇒ **Cmp Term Num Ion**

⇒ **Term Term Num Ion**

⇒ **Elem Term Num Ion**

⇒ `Mn` **Term Num Ion**

⇒ `Mn` **Elem Num Ion**

⇒ `MnO` **Num Ion**

⇒ `MnO` **IonNum Ion**

⇒ MnO_4 **Ion**

⇒ MnO_4^-

CFGs for Programming Languages

BLOCK	→	**STMT**
	|	`{` **STMTS** `}`
STMTS	→	ε
	|	**STMT STMTS**
STMT	→	**EXPR** `;`
	|	`if (`**EXPR**`)` **BLOCK**
	|	`while (`**EXPR**`)` **BLOCK**
	|	`do` **BLOCK** `while (`**EXPR**`);`
	|	**BLOCK**
	|	…
EXPR	→	`identifier`
	|	`constant`
	|	**EXPR** `+` **EXPR**
	|	**EXPR** `-` **EXPR**
	|	**EXPR** `*` **EXPR**
	|	…

Some CFG Notation

- Capital letters in **Bold Red Uppercase** will represent nonterminals.
  - i.e. **A**, **B**, **C**, **D**
- Lowercase letters in `blue monospace` will represent terminals.
  - i.e. `t`, `u`, `v`, `w`
- Lowercase Greek letters in ***gray italics*** will represent arbitrary strings of terminals and nonterminals.
  - i.e. ***α***, ***γ***, ***ω***

Examples

- We might write an arbitrary production as

$$\mathbf{A} \to \boldsymbol{\omega}$$

- We might write a string of a nonterminal followed by a terminal as

$$\mathbf{A}\mathtt{t}$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$\mathbf{B} \to \boldsymbol{\alpha}\mathbf{A}\mathtt{t}\boldsymbol{\omega}$$

Derivations

$$\mathbf{E} \to \mathbf{E}\ \mathbf{Op}\ \mathbf{E} \mid \texttt{int} \mid (\mathbf{E})$$
$$\mathbf{Op} \to \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/}$$

$\mathbf{E}$

$\Rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$

$\Rightarrow \mathbf{E}\ \mathbf{Op}\ \texttt{(}\mathbf{E}\texttt{)}$

$\Rightarrow \mathbf{E}\ \mathbf{Op}\ \texttt{(}\mathbf{E}\ \mathbf{Op}\ \mathbf{E}\texttt{)}$

$\Rightarrow \mathbf{E}\ \texttt{*}\ \texttt{(}\mathbf{E}\ \mathbf{Op}\ \mathbf{E}\texttt{)}$

$\Rightarrow \texttt{int *}\ \texttt{(}\mathbf{E}\ \mathbf{Op}\ \mathbf{E}\texttt{)}$

$\Rightarrow \texttt{int * (int}\ \mathbf{Op}\ \mathbf{E}\texttt{)}$

$\Rightarrow \texttt{int * (int}\ \mathbf{Op}\ \texttt{int)}$

$\Rightarrow \texttt{int * (int + int)}$

- This sequence of steps is called a **derivation**.
- A string $\alpha A \omega$ **yields** string $\alpha \gamma \omega$ iff $A \to \gamma$ is a production.
- If α yields β, we write $\alpha \Rightarrow \beta$.
- We say that α **derives** β iff there is a sequence of strings where

$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \beta$$

- If α derives β, we write $\alpha \Rightarrow^* \beta$.

The Language of a Grammar

- If G is a CFG with alphabet Σ and start symbol **S**, then the **language of *G*** is the set

$$\mathscr{L}(G) = \{ \boldsymbol{\omega} \in \Sigma^* \mid \mathbf{S} \Rightarrow^* \boldsymbol{\omega} \}$$

- That is, the set of strings derivable from the start symbol.

The Language of a Grammar

- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{0}\mathbf{S}\mathtt{1} \mid \varepsilon$$

- What strings can this generate?

0	0	0	0	0	0	S	1	1	1	1	1	1

The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{0}\mathbf{S}\mathtt{1} \mid \varepsilon$$

- What strings can this generate?

0	0	0	0	0	0	1	1	1	1	1	1

The Language of a Grammar

- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{0}\mathbf{S}\mathtt{1} \mid \varepsilon$$

- What strings can this generate?

0	0	0	0	0	0	1	1	1	1	1	1

$$\mathscr{L}(G) = \{ \mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N} \}$$

Designing CFGs

- Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft.
- When thinking about CFGs:
  - Think recursively: Build up bigger structures from smaller ones.
  - Have a construction goal: Know in what order you will build up the string.

Designing CFGs

- Let $\Sigma = \{\texttt{0}, \texttt{1}\}$ and let $L = \{w \in \Sigma^* \mid w$ is a palindrome $\}$
- We can design a CFG for L by thinking inductively:
  - Base case: ε, 0, and 1 are palindromes.
  - If ω is a palindrome, then $\texttt{0}\omega\texttt{0}$ and $\texttt{1}\omega\texttt{1}$ are palindromes.

$$\mathbf{S} \to \varepsilon \mid \texttt{0} \mid \texttt{1} \mid \texttt{0}\mathbf{S}\texttt{0} \mid \texttt{1}\mathbf{S}\texttt{1}$$

Designing CFGs

- Let $\Sigma = \{$ **(**, **)** $\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced parentheses $\}$
- We can think about how we will build strings in this language as follows:
  - The empty string is balanced.
  - Any two strings of balanced parentheses can be concatenated.
  - Any string of balanced parentheses can be parenthesized.

$$\mathbf{S} \rightarrow \mathbf{SS} \mid \mathbf{(S)} \mid \varepsilon$$

CFGs and PDAs

Context-Free Languages

- Recall from last time that we defined the context-free languages as follows:

A language L is context-free iff there is a PDA P such that $\mathscr{L}(P) = L$.

- The term "context-free grammar" has "context-free" in it.
- Can we show that a language L is context-free iff there is a CFG for it?

From CFGs to PDAs

- ***Theorem:*** If G is a CFG for a language L, then there exists a PDA for L as well.
- **Idea:** Build a PDA that simulates expanding out the CFG from the start symbol to some particular string.
- Stack holds the part of the string we haven't matched yet.

From CFGs to PDAs

- Example: Let $\Sigma = \{ \mathtt{1}, \geq \}$ and let $GE = \{ \mathtt{1}^m \geq \mathtt{1}^n \mid m, n \in \mathbb{N} \land m \geq n \}$
  - $\mathtt{111{\geq}11} \in GE$
  - $\mathtt{11{\geq}11} \in GE$
  - $\mathtt{1111{\geq}11} \in GE$
  - $\geq \in GE$
- One CFG for *GE* is the following:

$$\mathbf{S} \rightarrow \mathtt{1}\mathbf{S}\mathtt{1} \mid \mathtt{1}\mathbf{S} \mid \geq$$

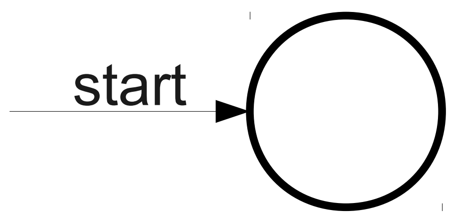
- How would we build a PDA for *GE*?

From CFGs to PDAs

$$
\begin{aligned}
\mathbf{S} &\rightarrow \mathtt{1}\mathbf{S}\mathtt{1} \\
\mathbf{S} &\rightarrow \mathtt{1}\mathbf{S} \\
\mathbf{S} &\rightarrow \geq
\end{aligned}
$$

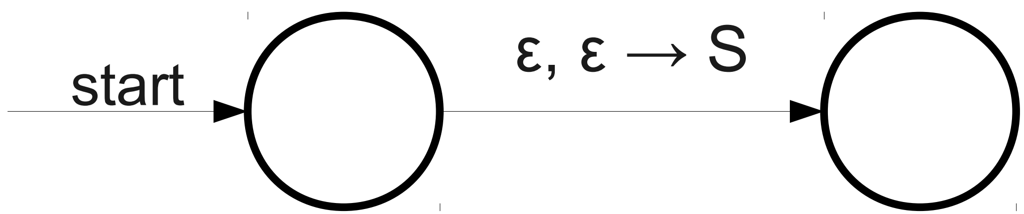
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to \geq$



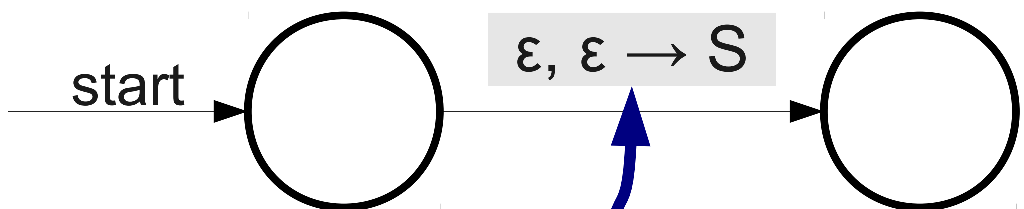
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



From CFGs to PDAs

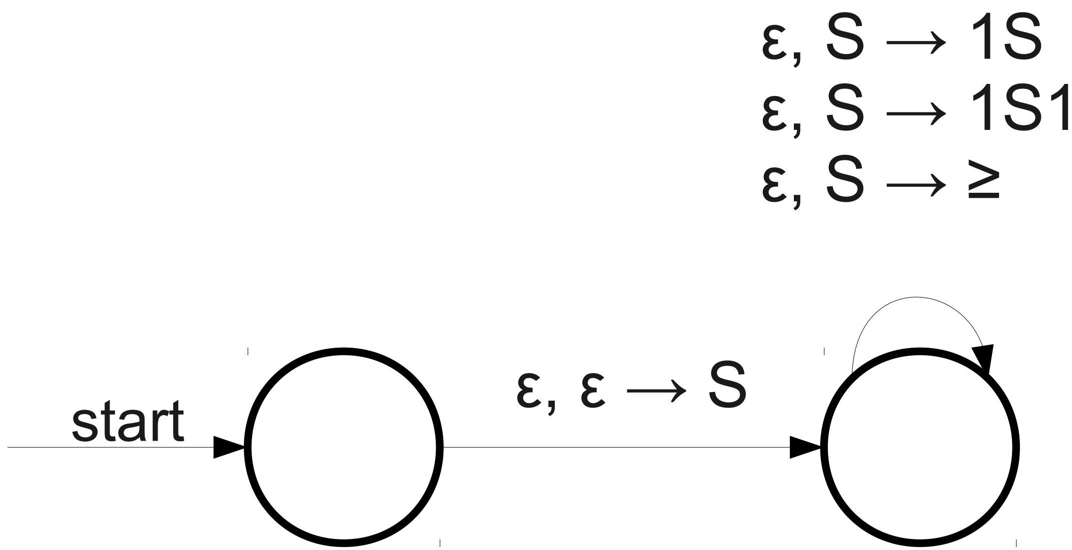
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



We begin by putting the start symbol of the grammar onto the stack so that we can begin applying productions.

From CFGs to PDAs

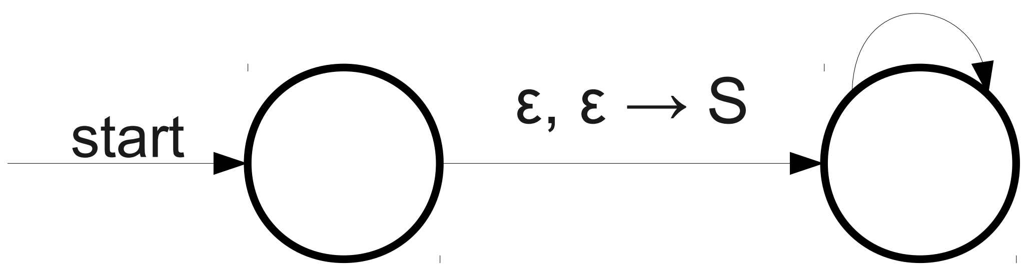
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥



From CFGs to PDAs

S → 1S1
S → 1S
S → ≥

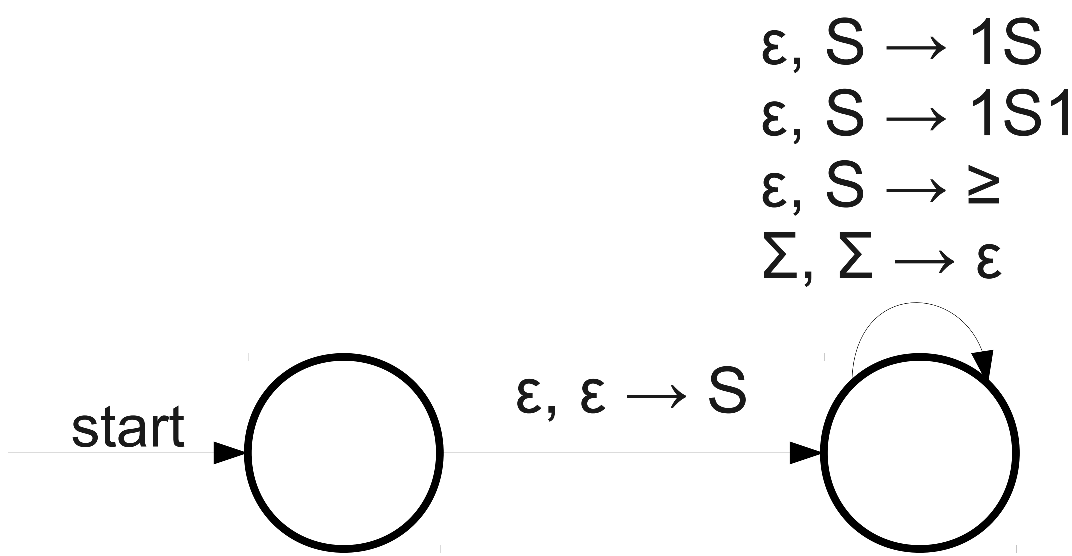
ε, S → 1S
ε, S → 1S1
ε, S → ≥



These transitions allow us to nondeterministically guess which production to use when the top of the stack is a nonterminal.

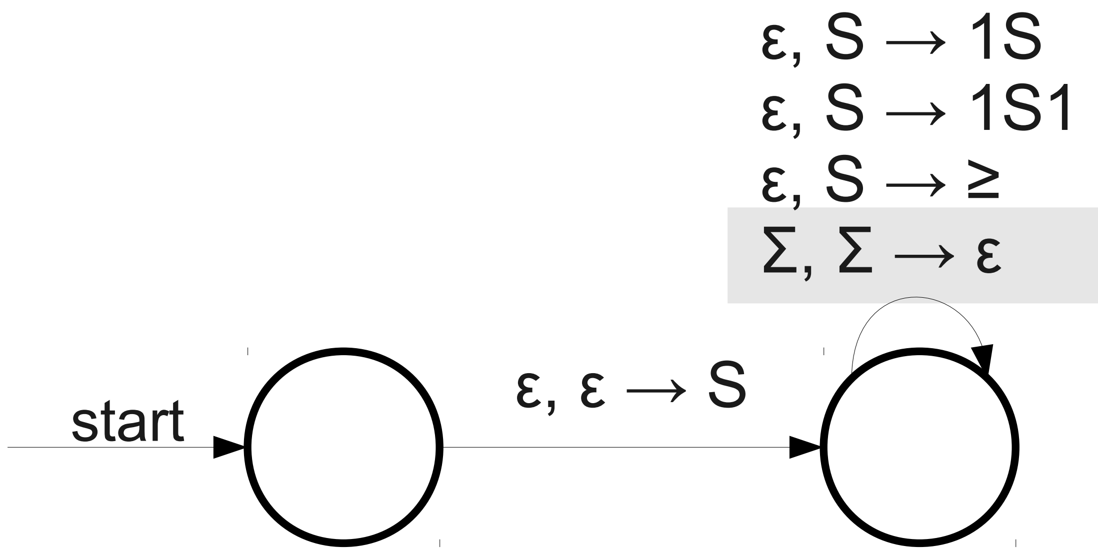
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to$ ≥



From CFGs to PDAs

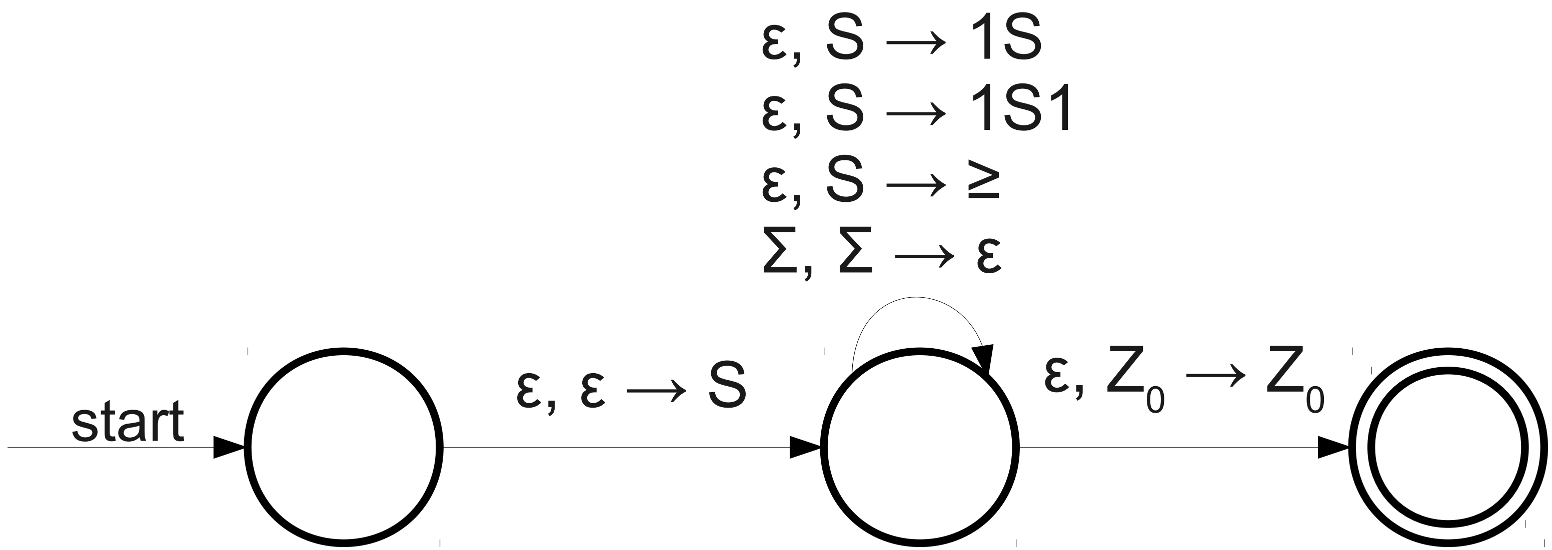
$\mathbf{S} \to \mathbf{1S1}$
$\mathbf{S} \to \mathbf{1S}$
$\mathbf{S} \to \geq$



Once we have guessed the right production, this rule lets us match the next character from the input with the next terminal we produced.

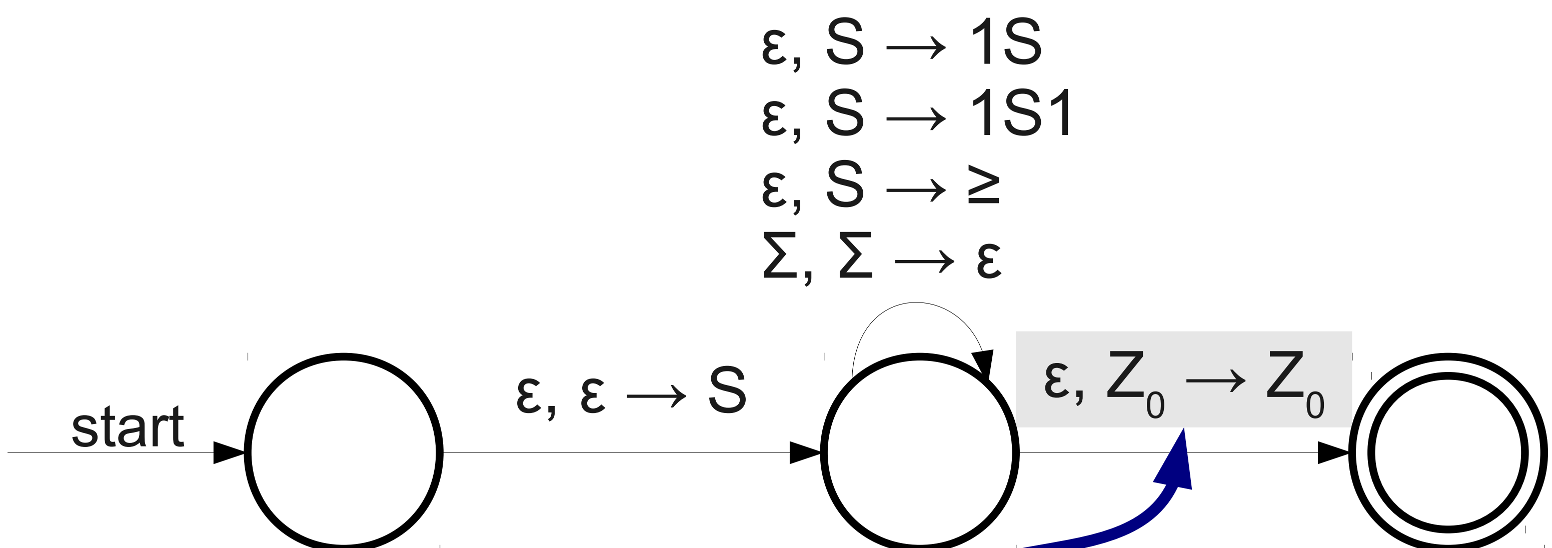
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



From CFGs to PDAs

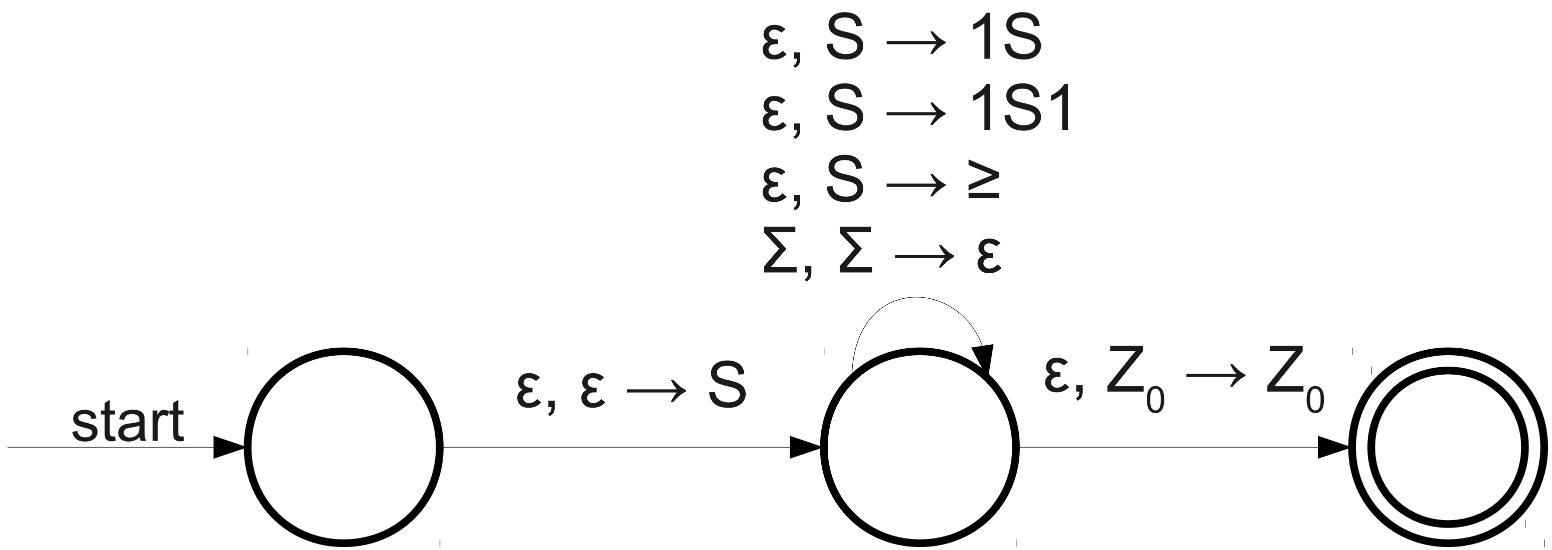
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



Once we have fully expanded out all nonterminals and matched all the terminals on the stack, we can transition into the accepting state.

From CFGs to PDAs

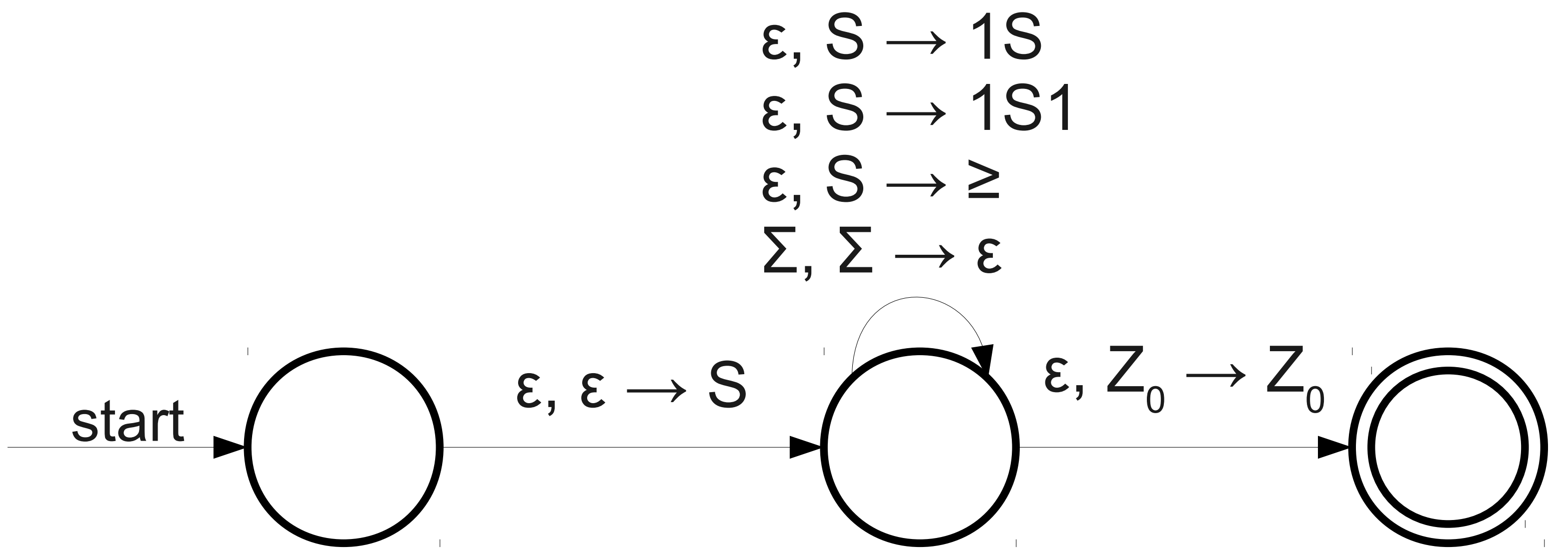
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



1 1 1 ≥ 1 1

From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

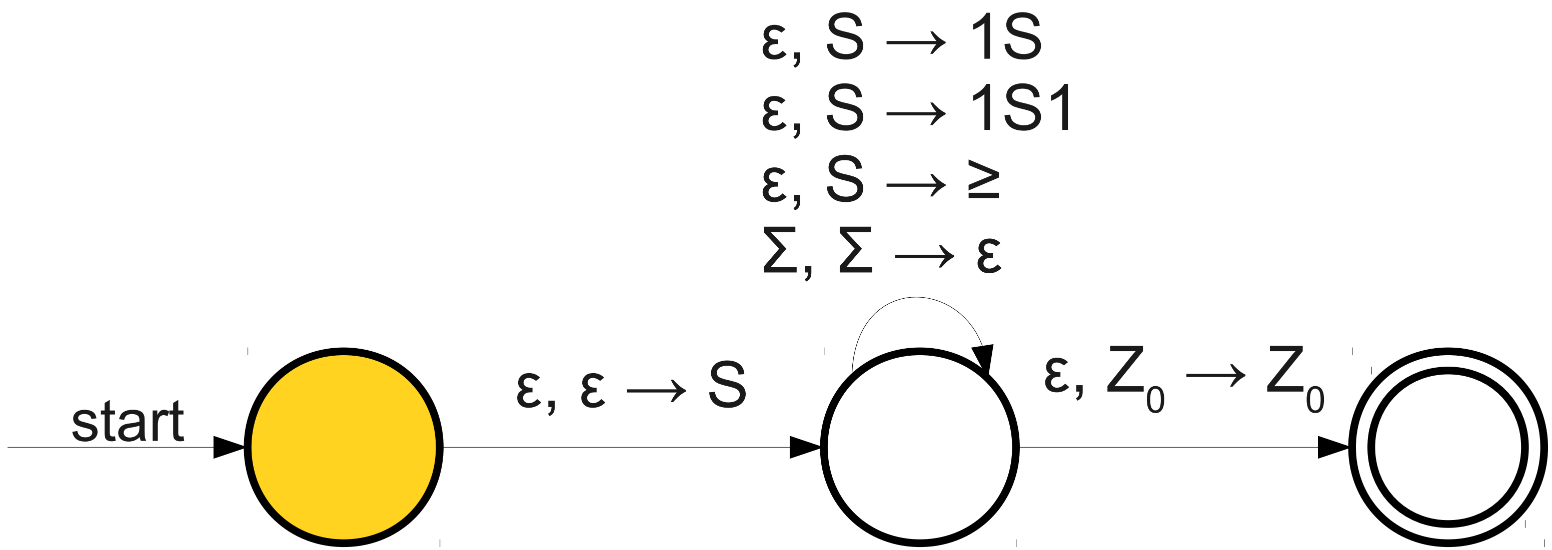


1 1 1 ≥ 1 1

Z_0

From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

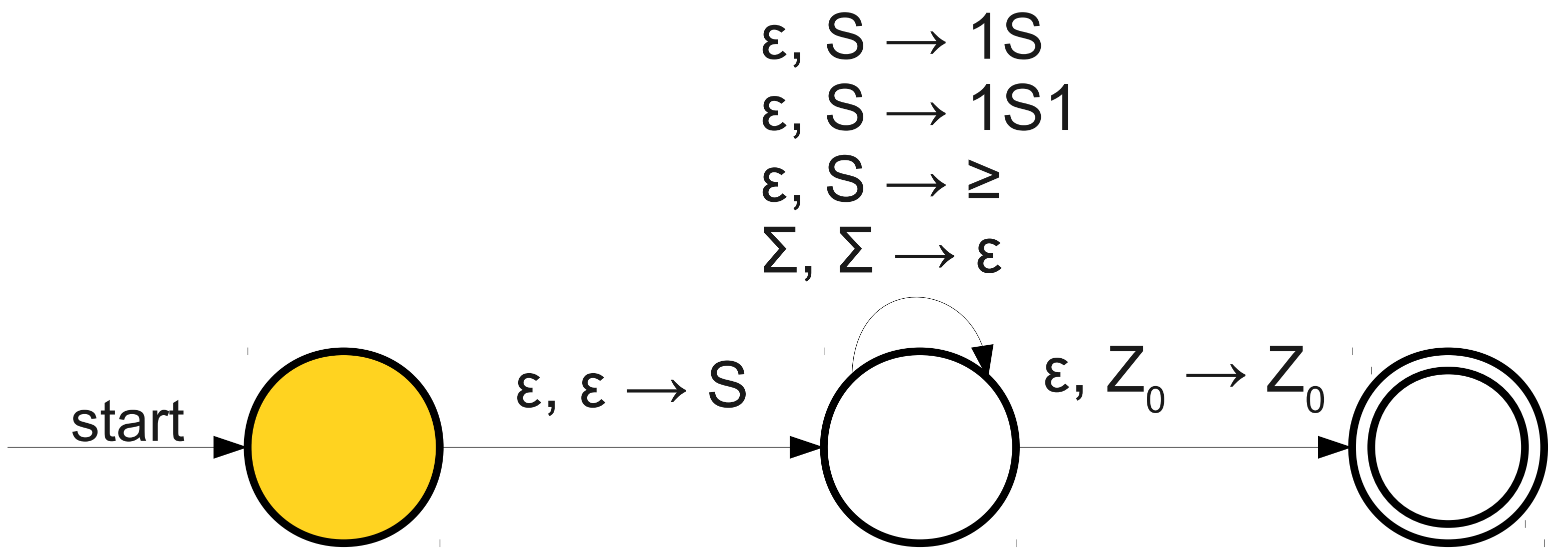


1 1 1 ≥ 1 1

Z_0

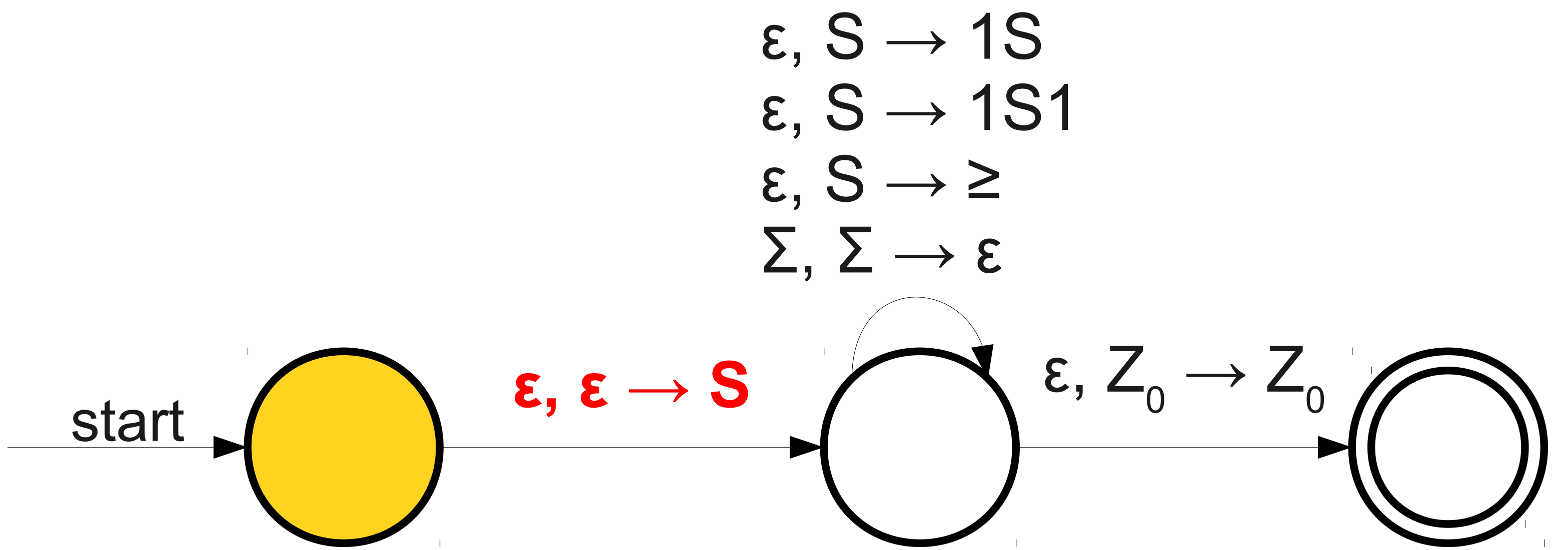
From CFGs to PDAs

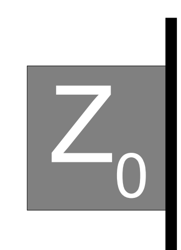
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥



From CFGs to PDAs

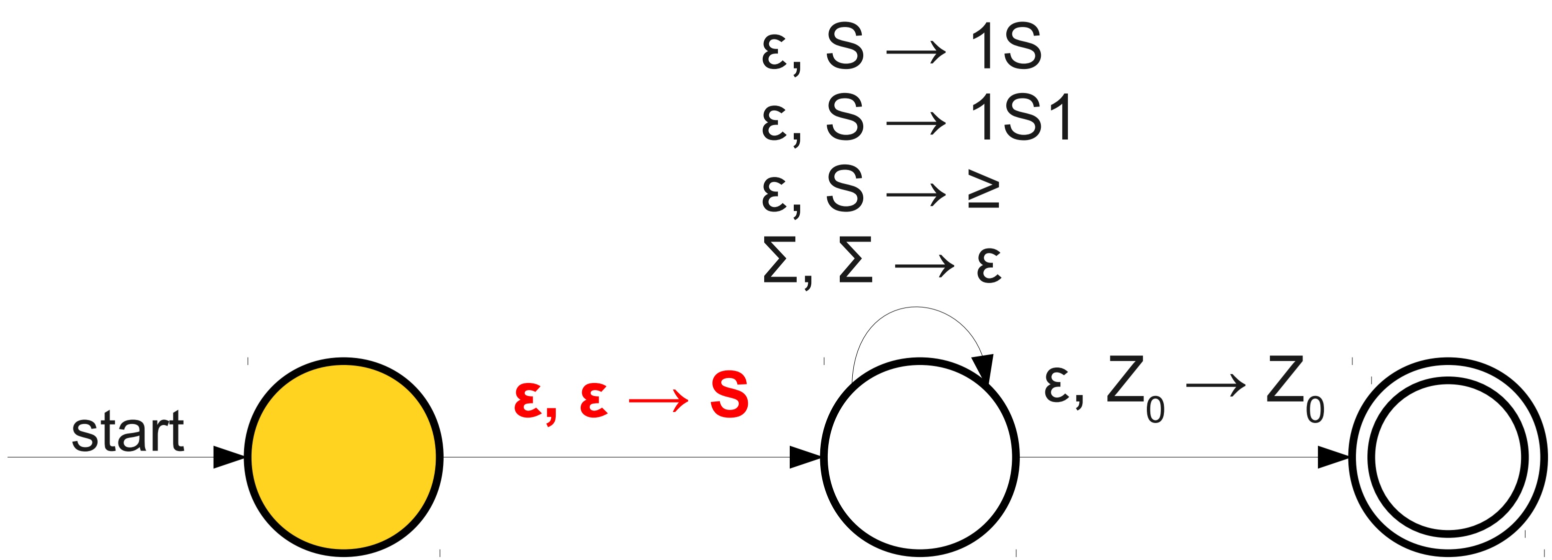
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

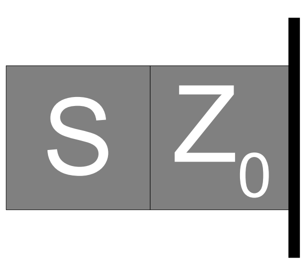




From CFGs to PDAs

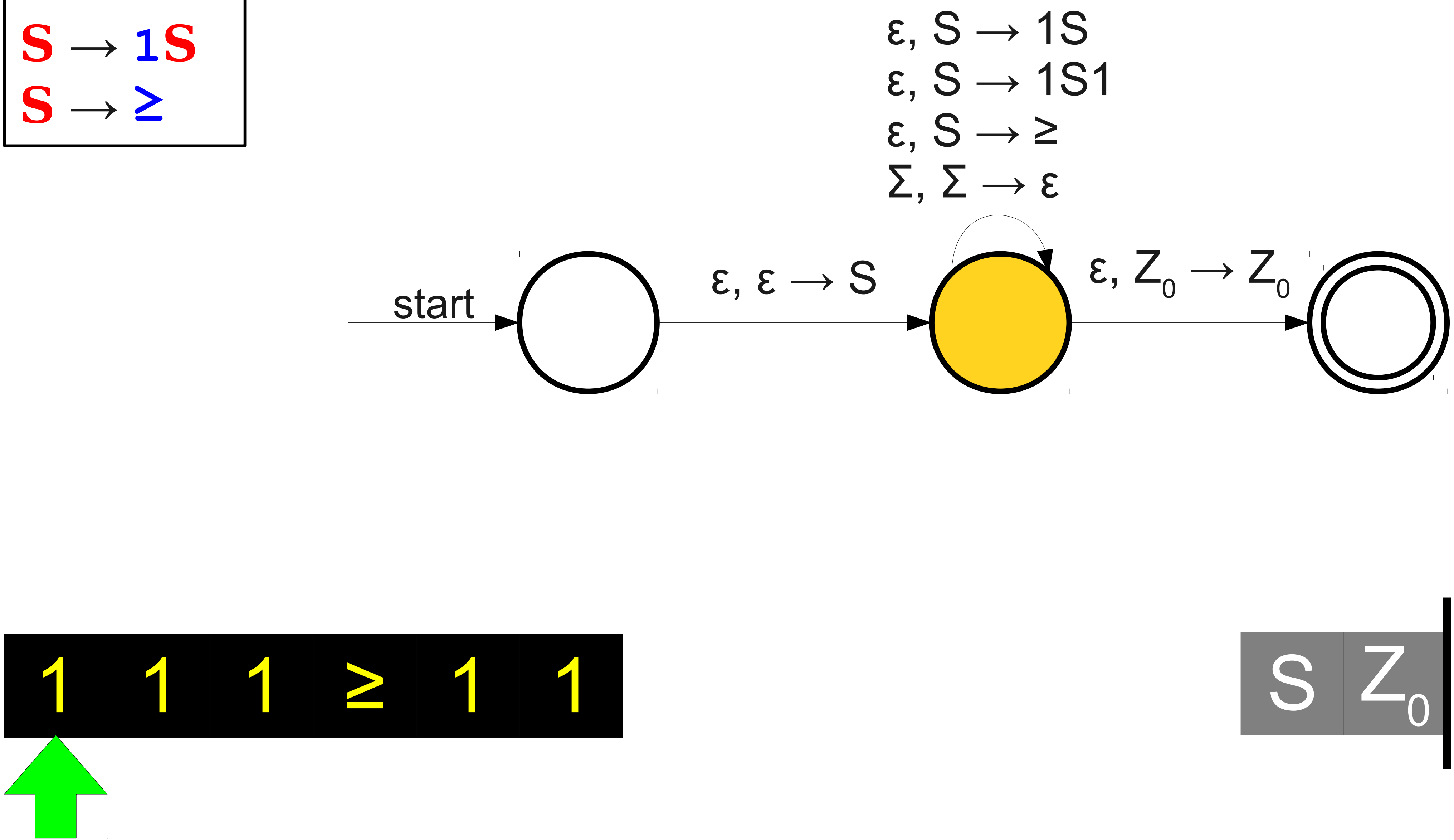
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$





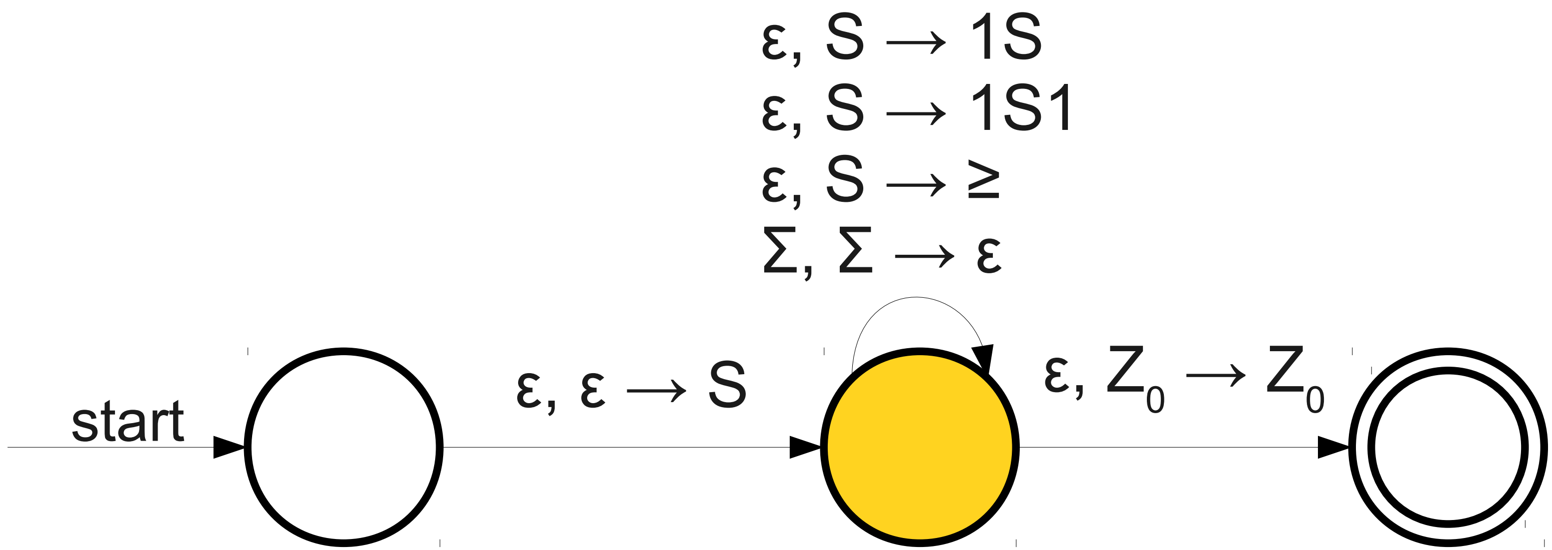
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to \geq$



From CFGs to PDAs

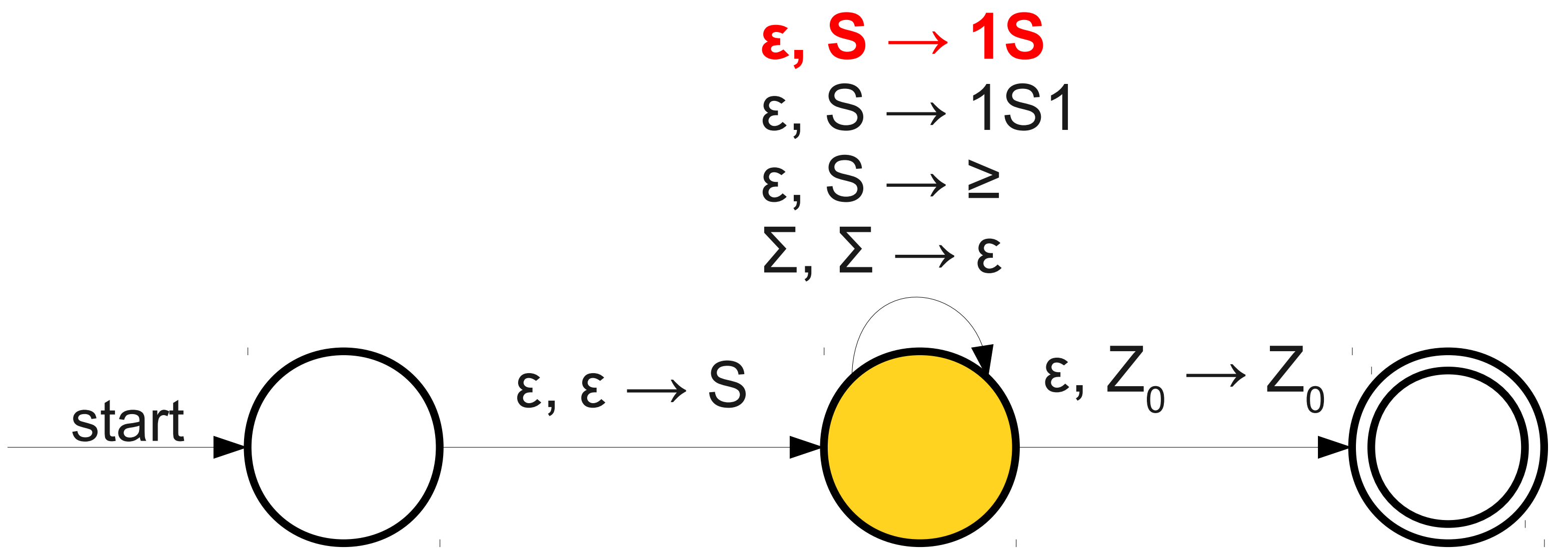
$$S \rightarrow 1S1$$
$$S \rightarrow 1S$$
$$S \rightarrow \geq$$



Now that the stack top is a nonterminal, we guess which production to use.

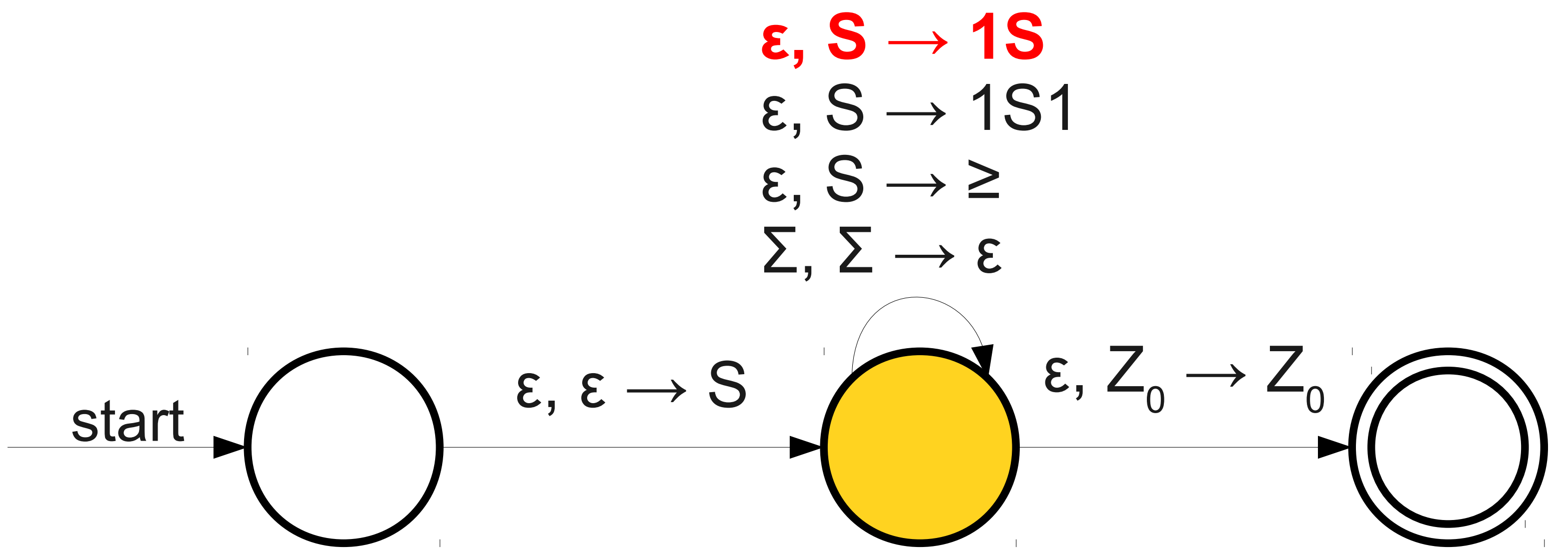
From CFGs to PDAs

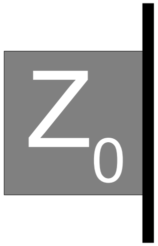
$S \to 1S1$
$S \to 1S$
$S \to \geq$



From CFGs to PDAs

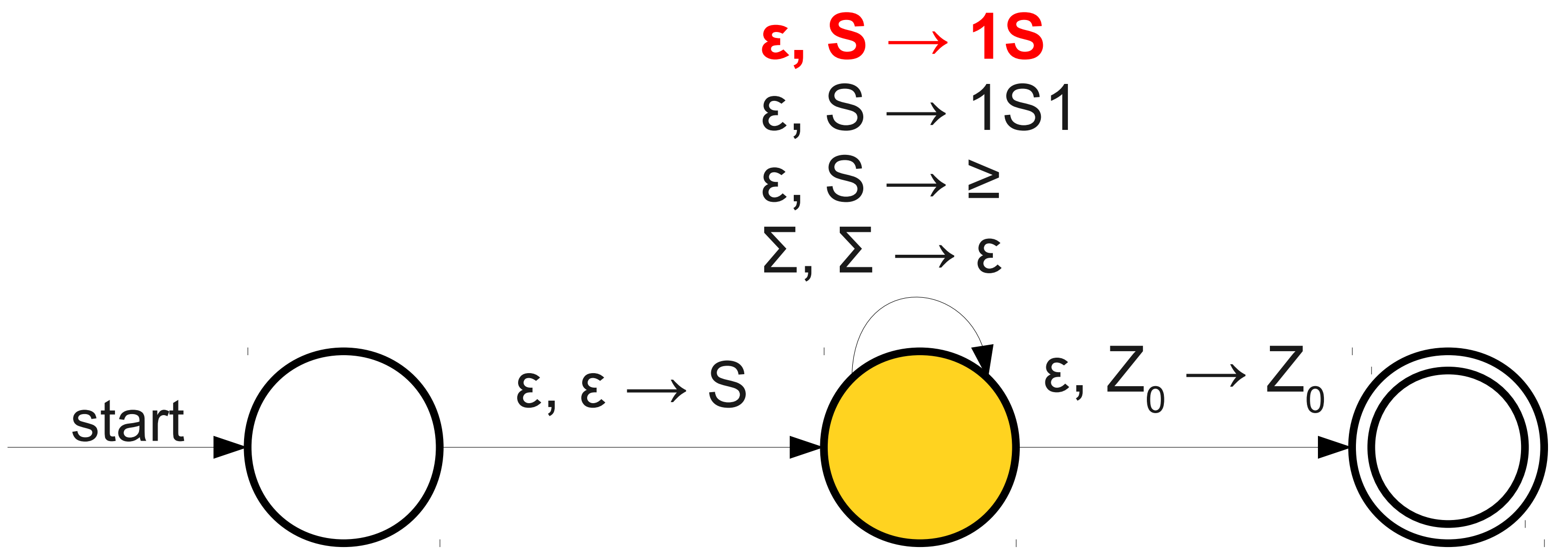
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$





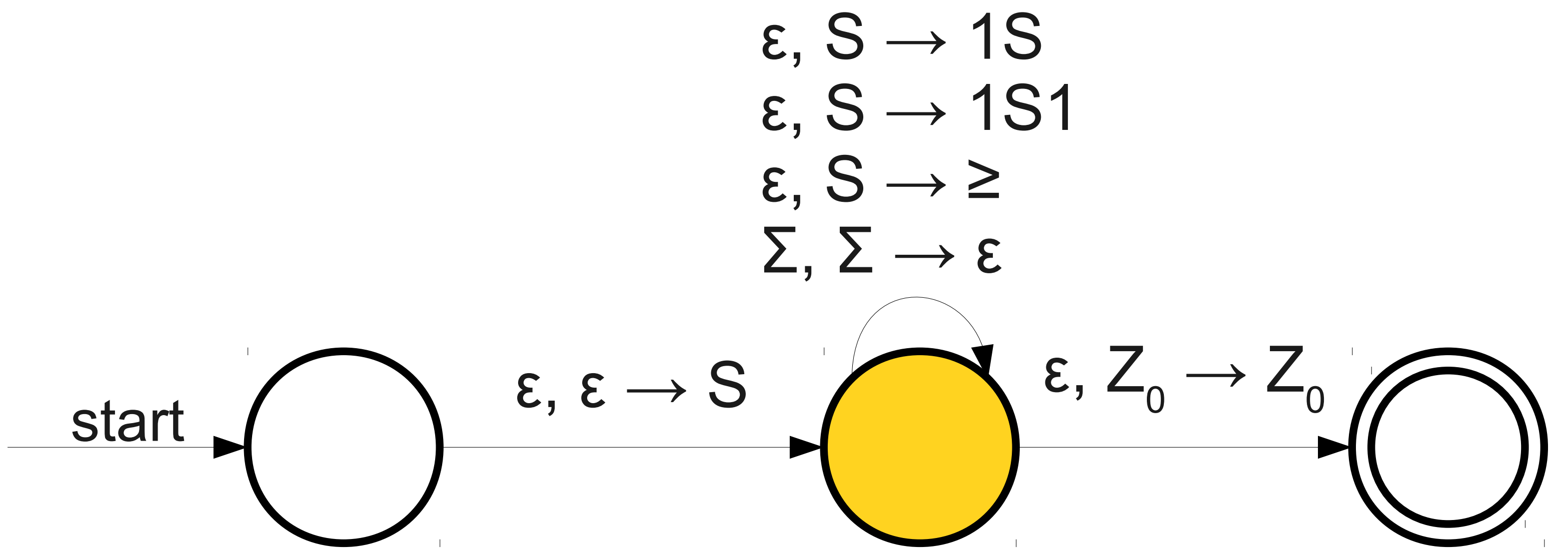
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$



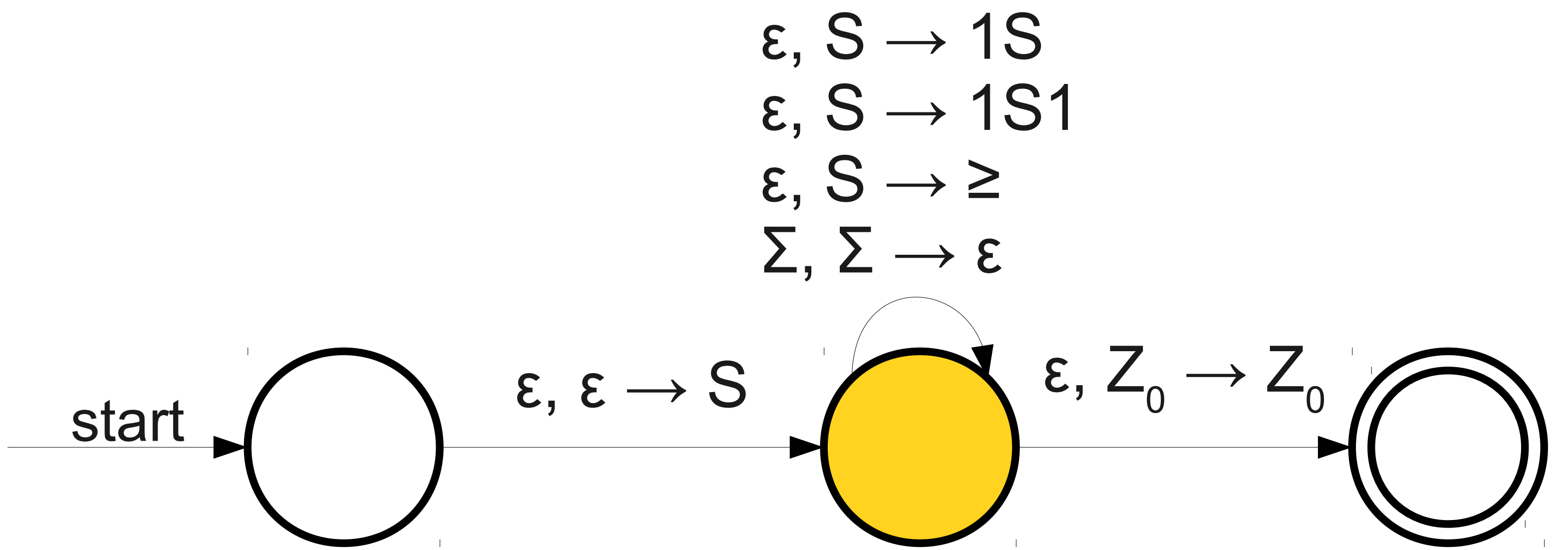
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

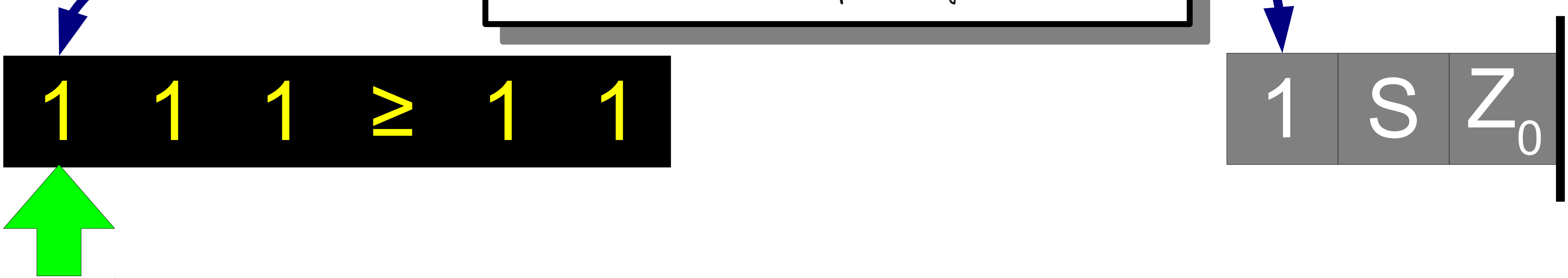


From CFGs to PDAs

S → **1S1**
S → **1S**
S → **≥**

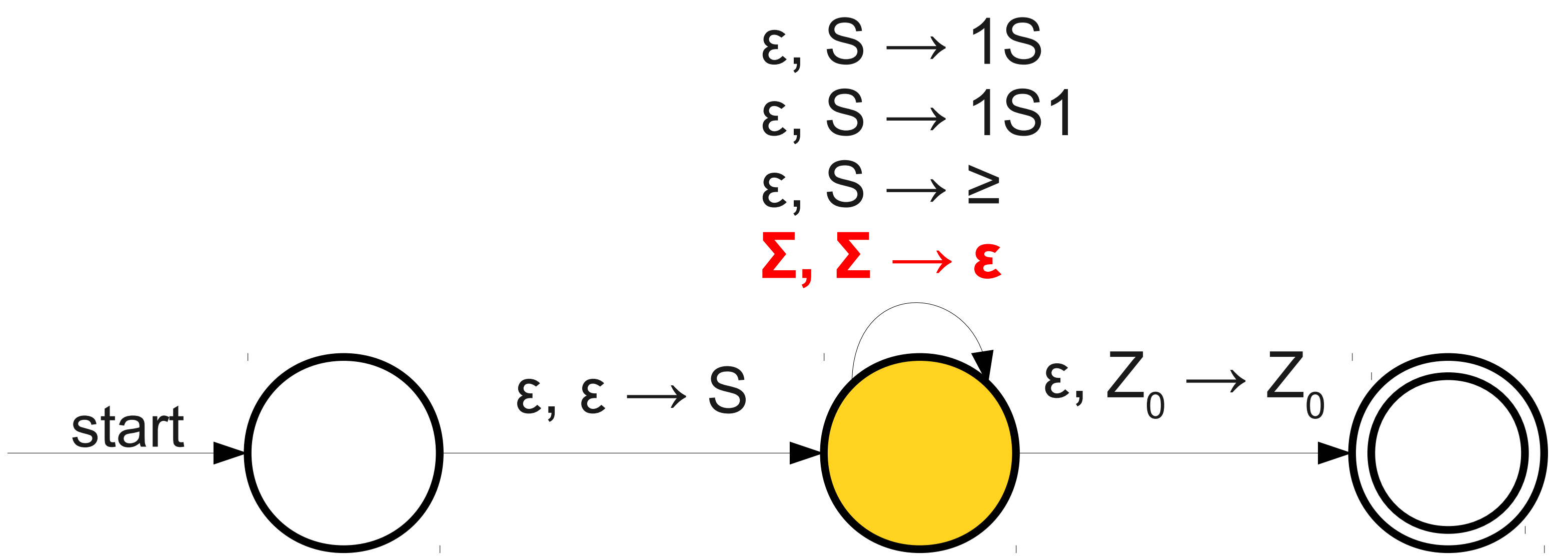


Since the top of the stack is a terminal, we can match it with the next input symbol.



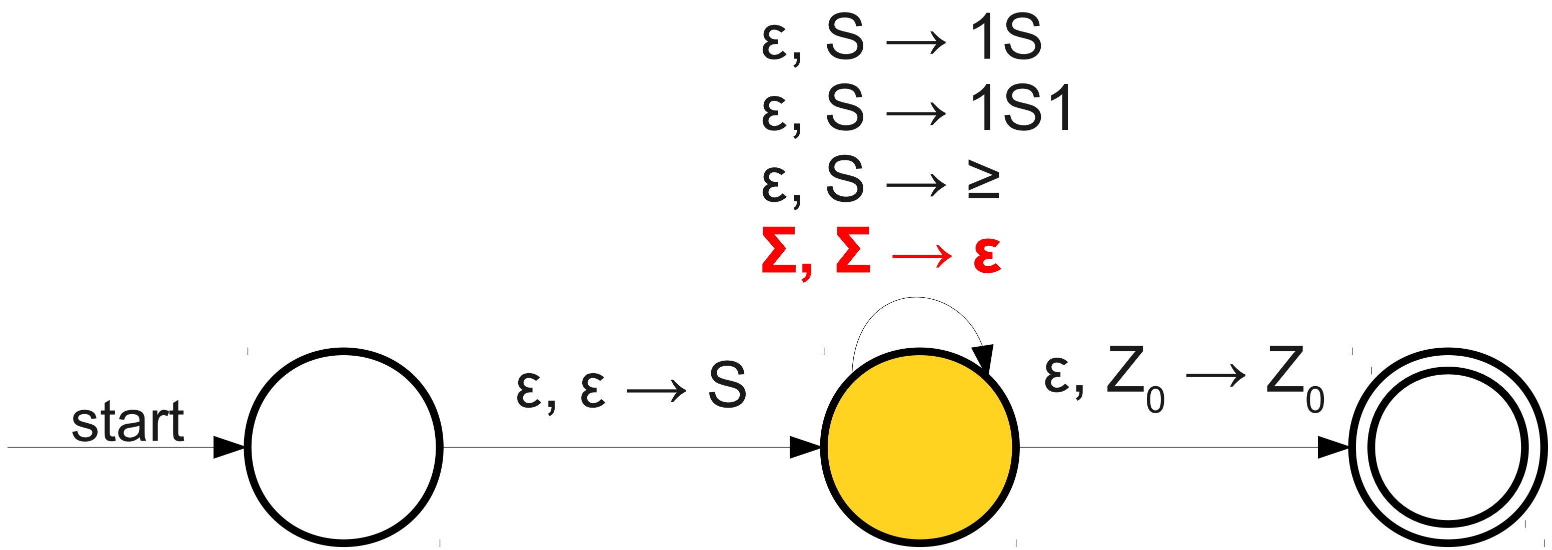
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to ≥$



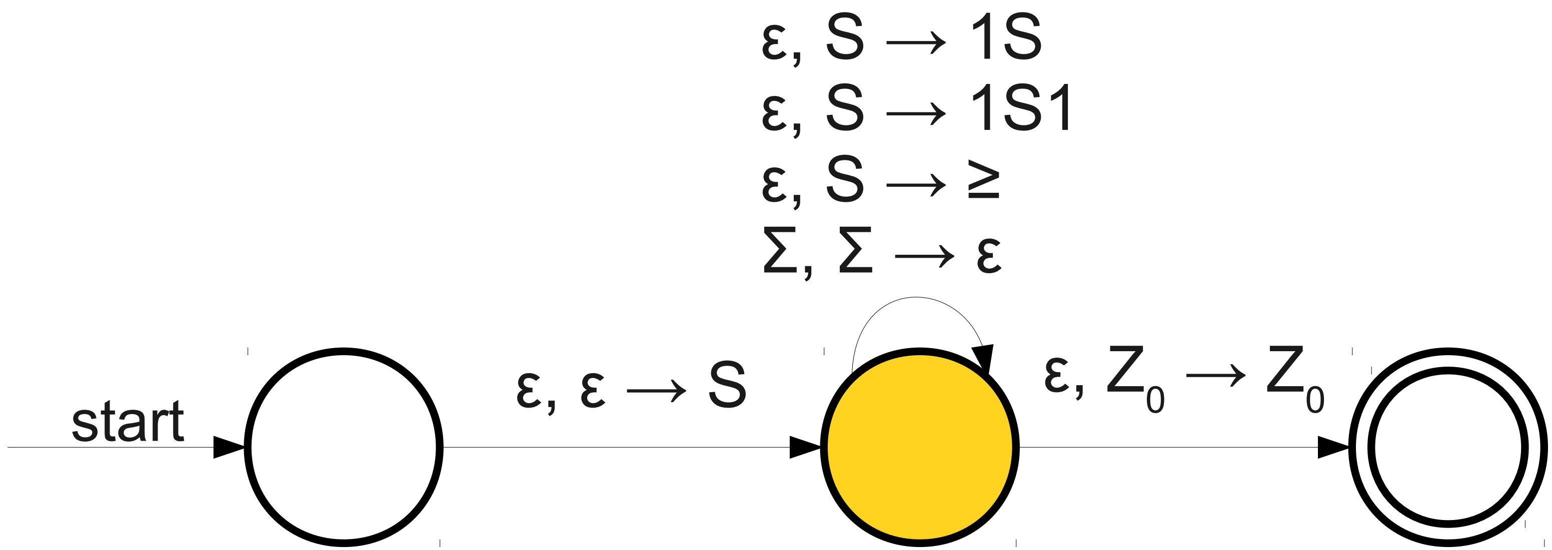
From CFGs to PDAs

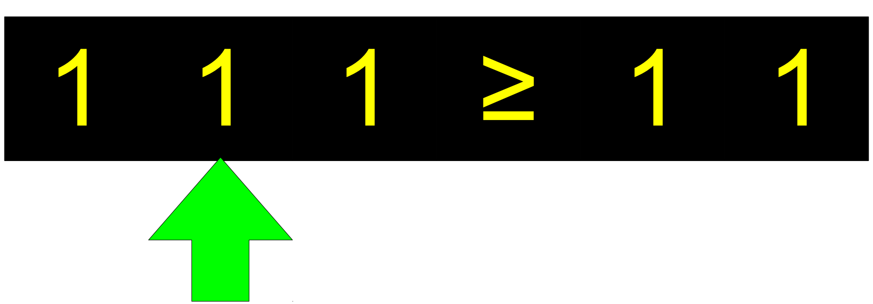
$\mathbf{S} \rightarrow \mathbf{1S1}$
$\mathbf{S} \rightarrow \mathbf{1S}$
$\mathbf{S} \rightarrow \geq$



From CFGs to PDAs

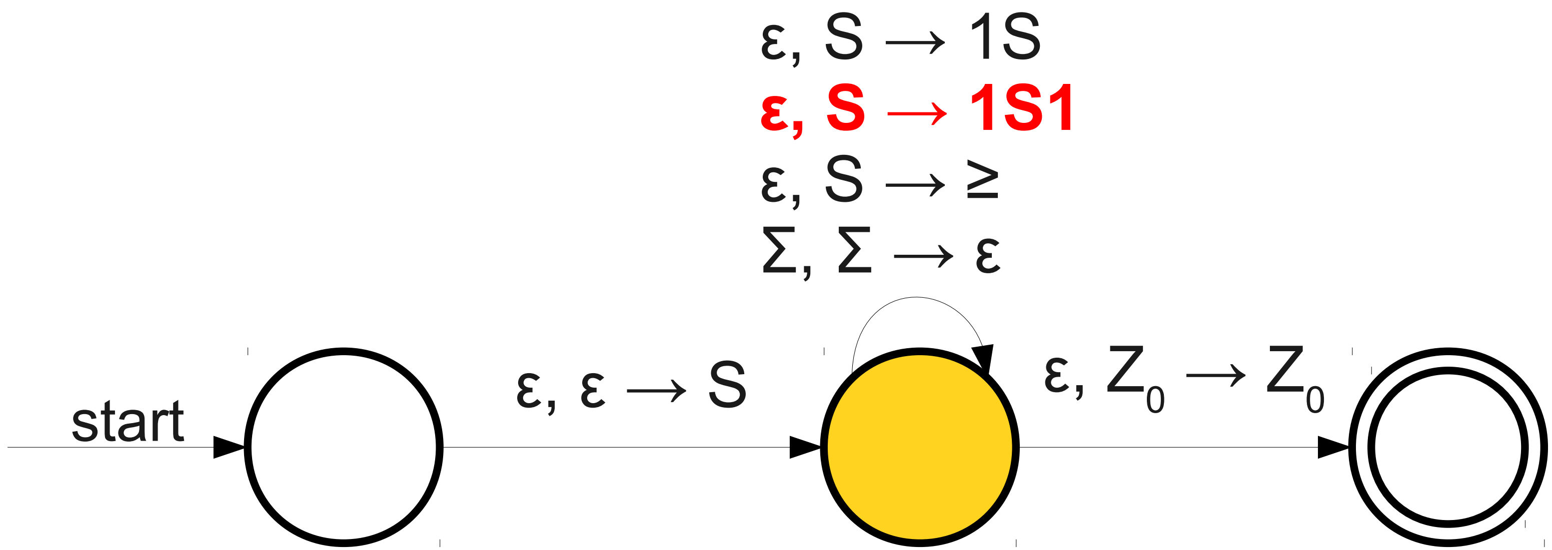
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$

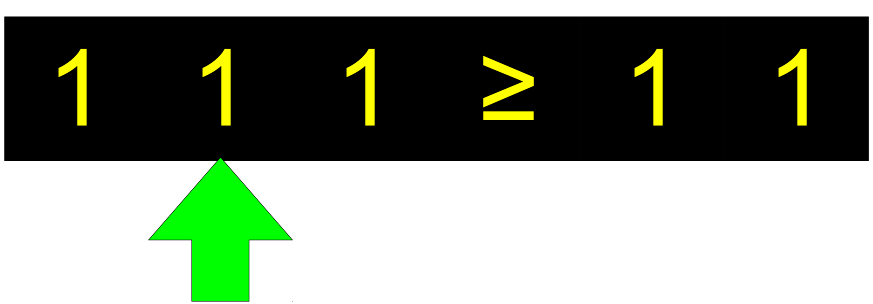




From CFGs to PDAs

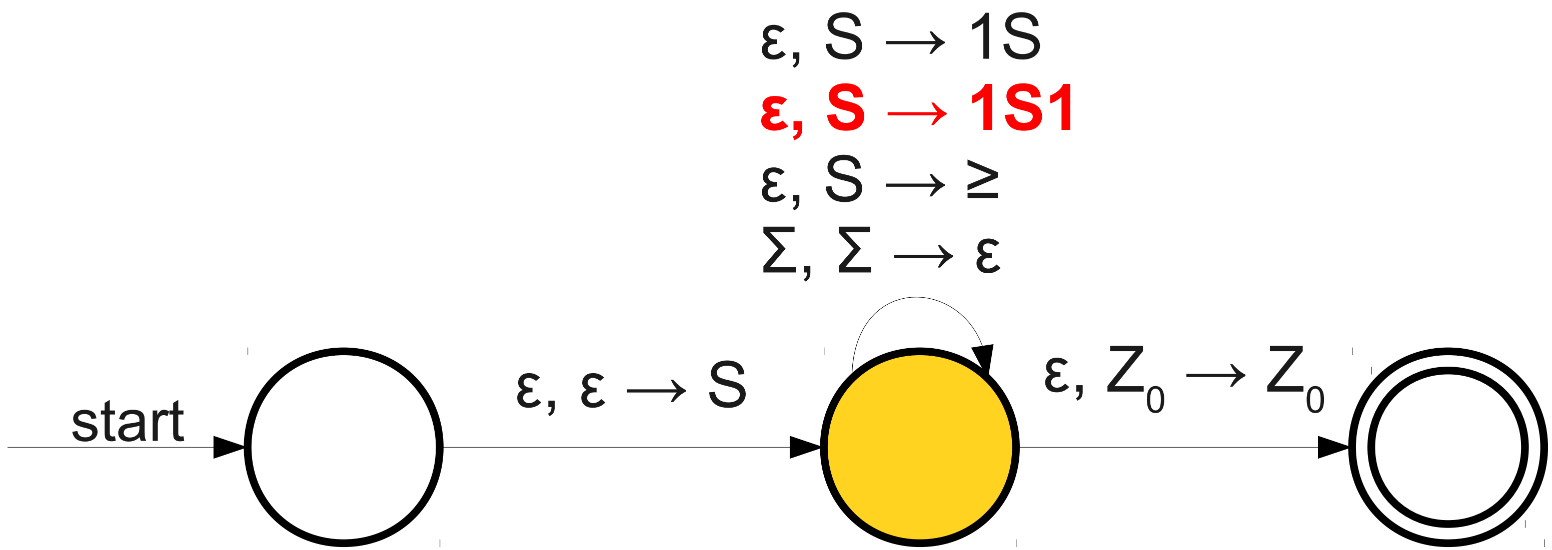
S → **1S1**
S → **1S**
S → **≥**

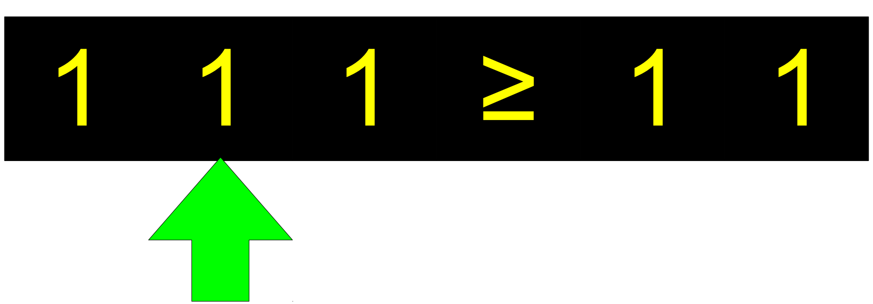


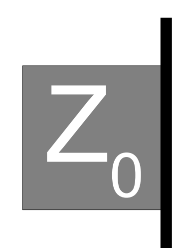


From CFGs to PDAs

S → 1S1
S → 1S
S → ≥

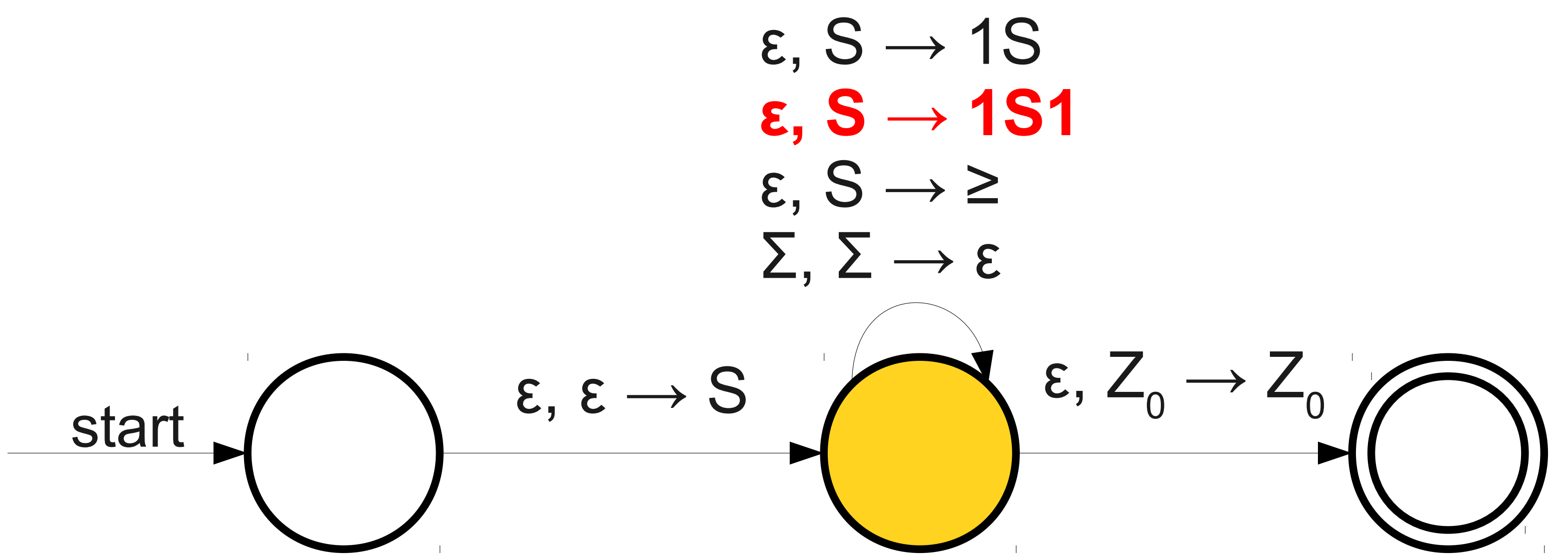






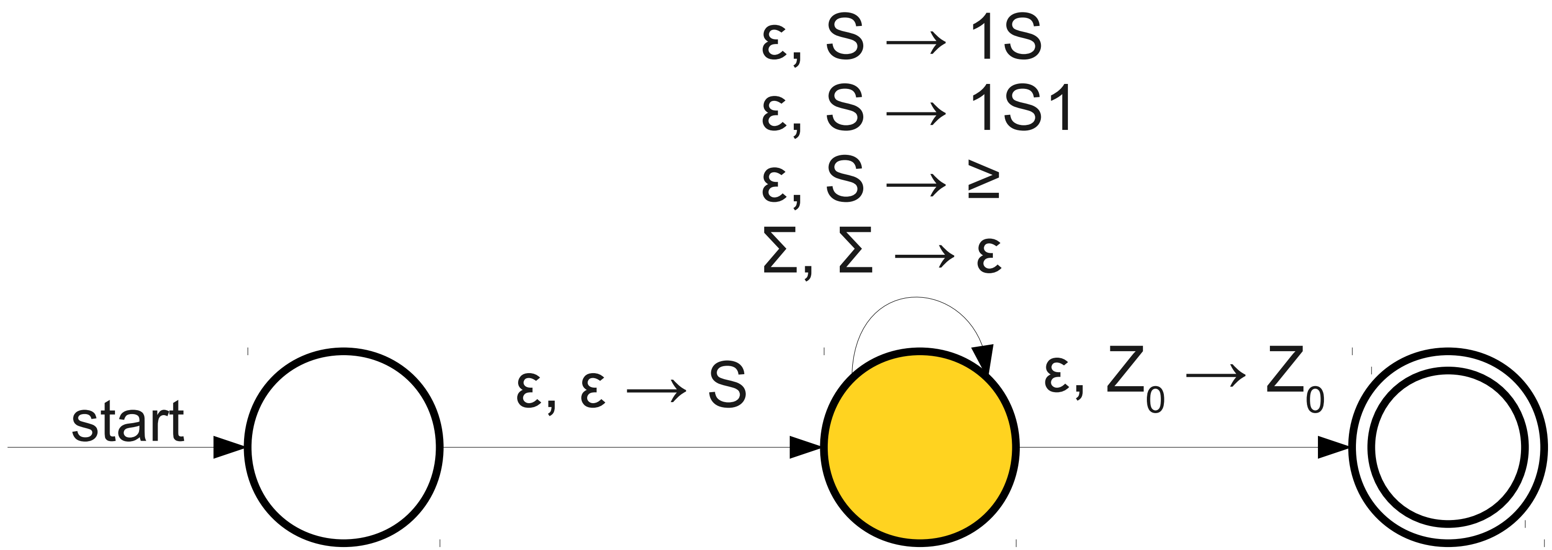
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



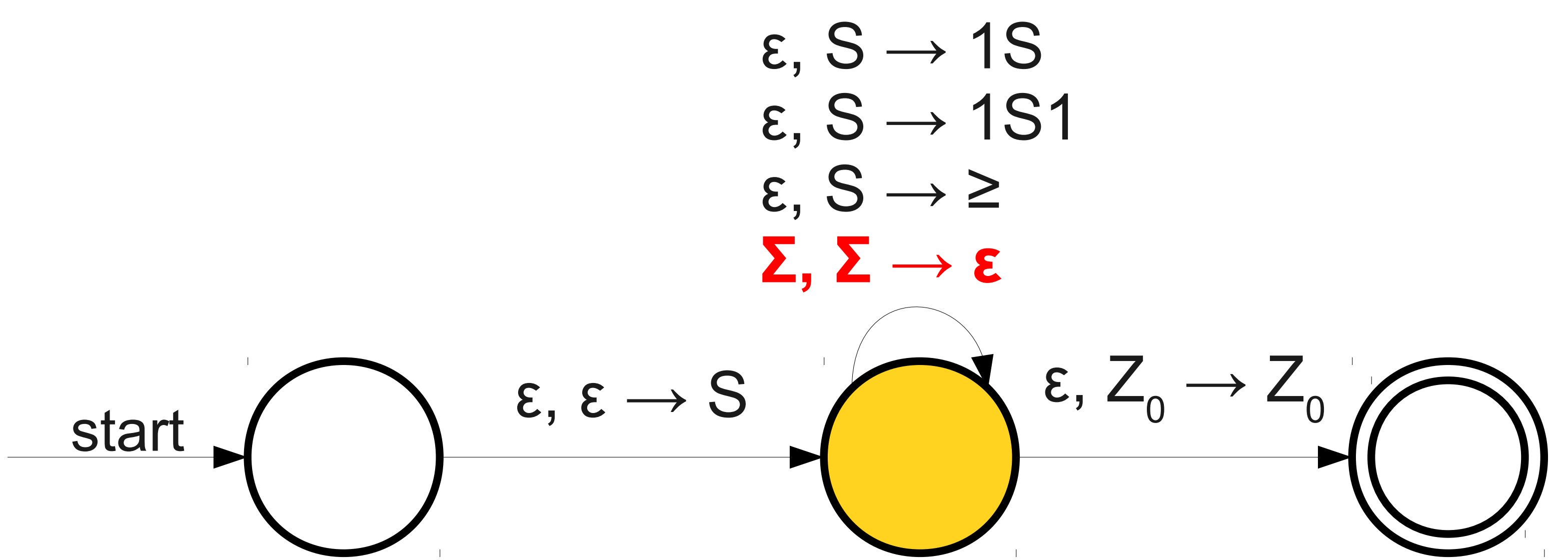
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



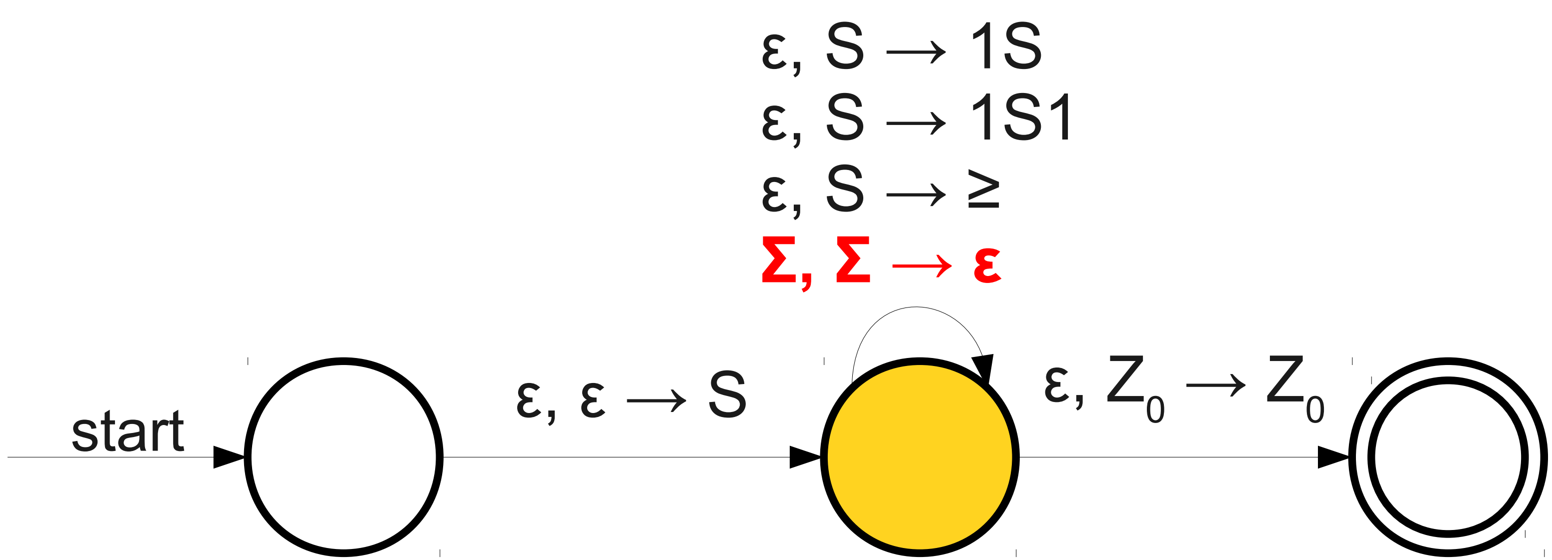
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to \geq$



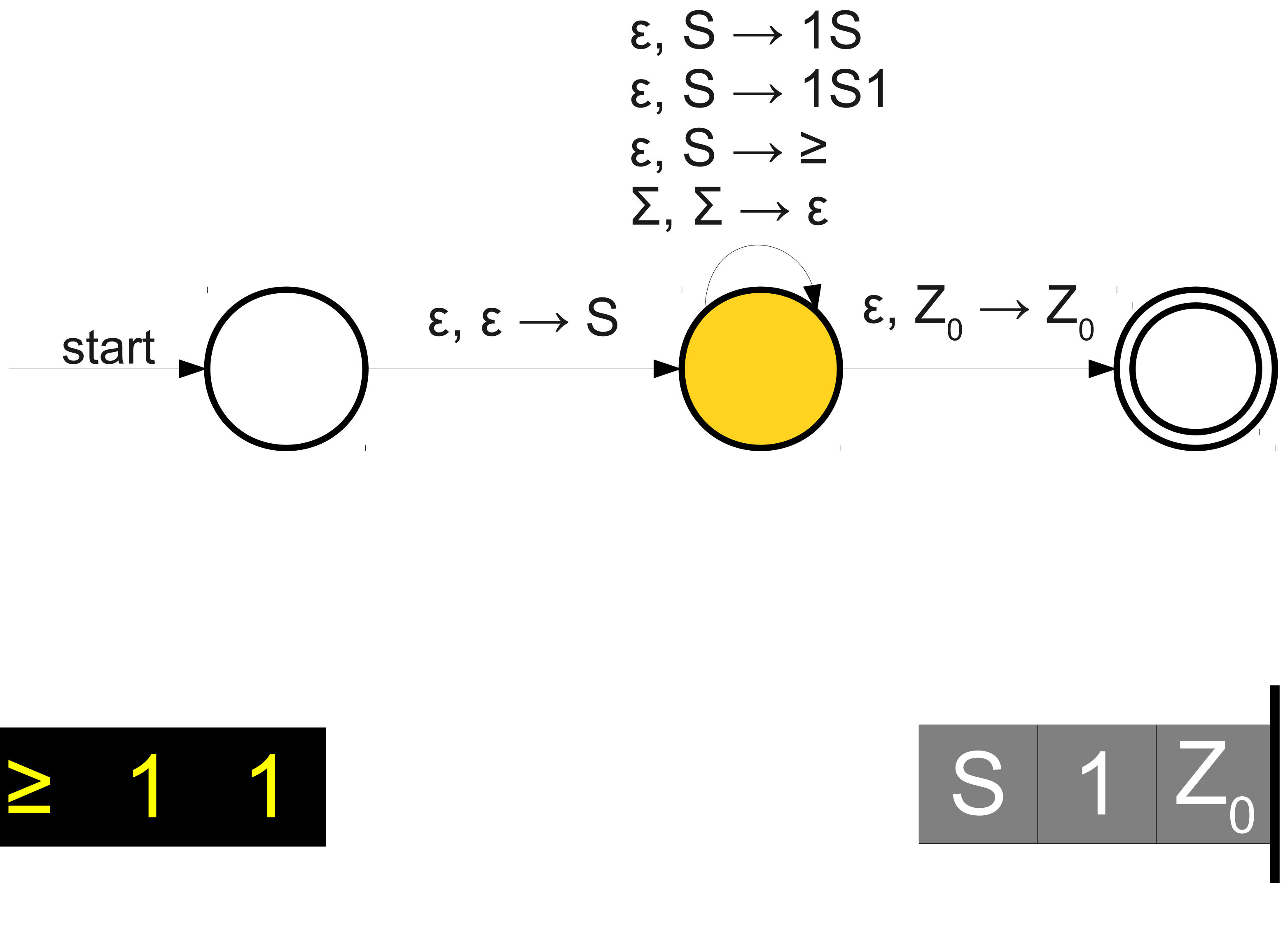
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to \geq$



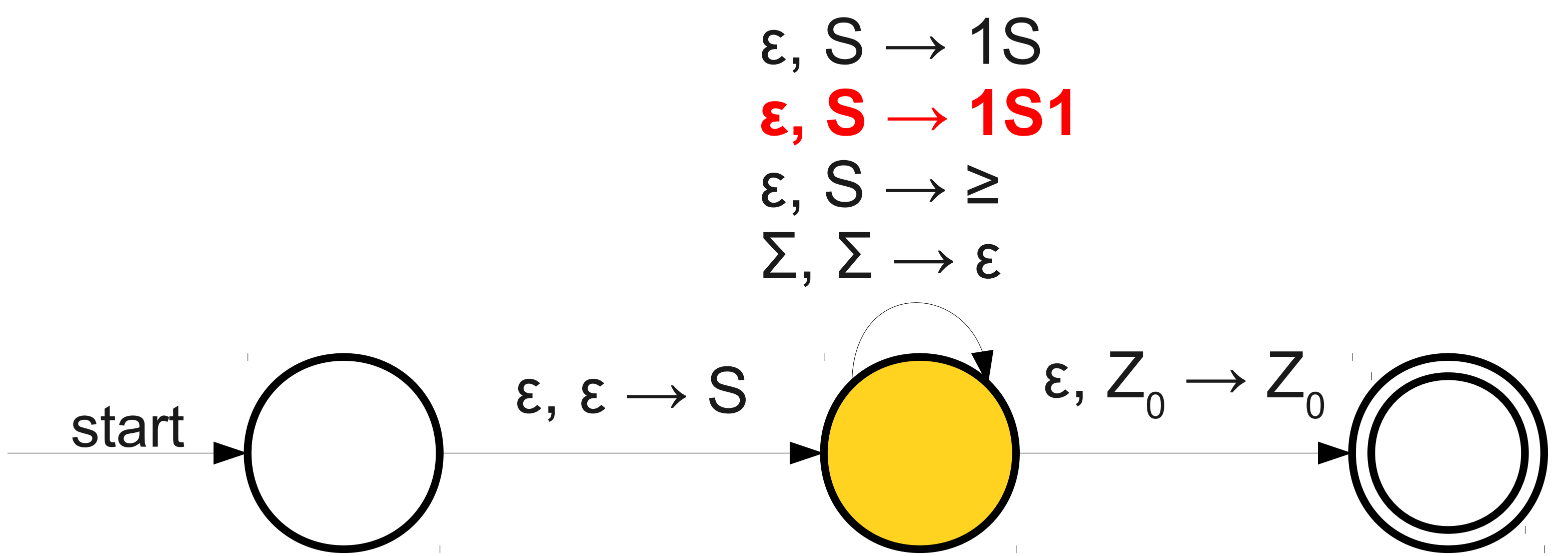
From CFGs to PDAs

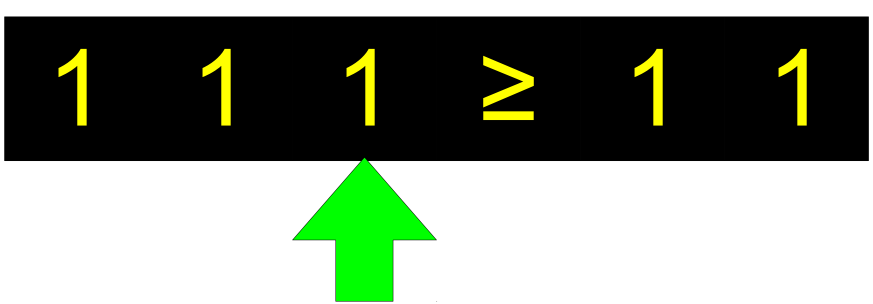
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

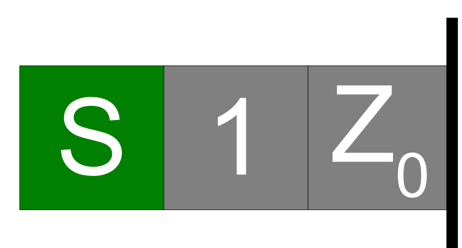


From CFGs to PDAs

S → 1S1
S → 1S
S → ≥





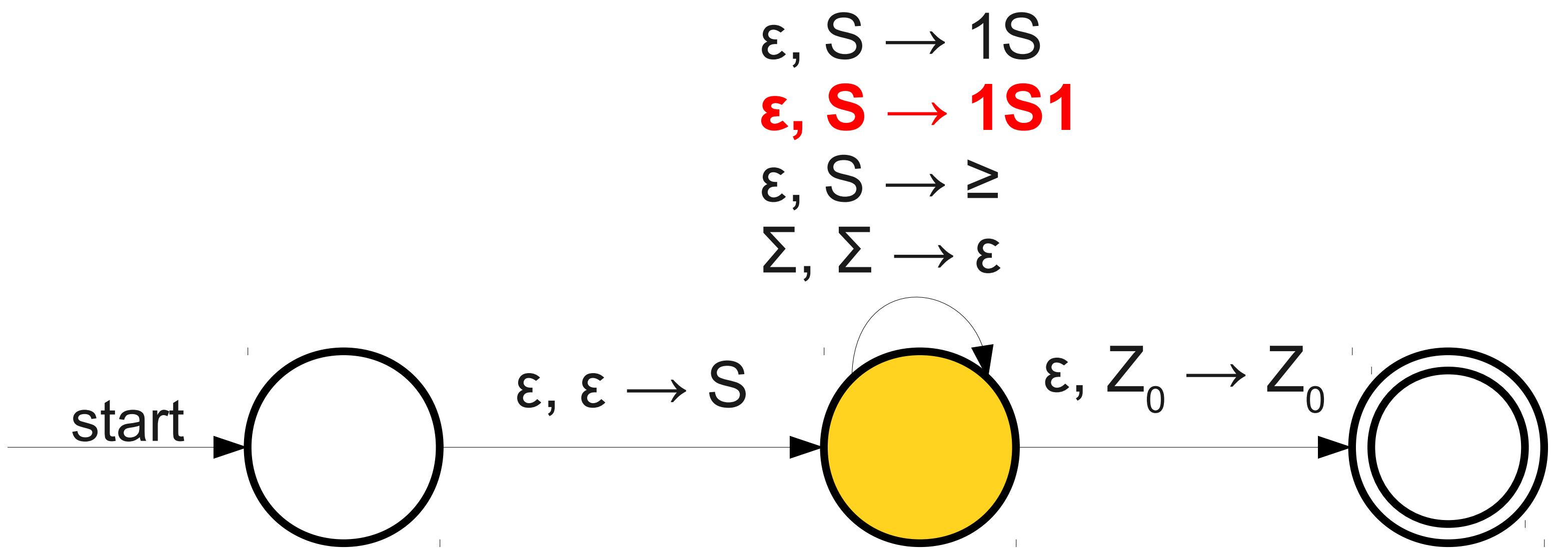


From CFGs to PDAs

$S \rightarrow 1S1$

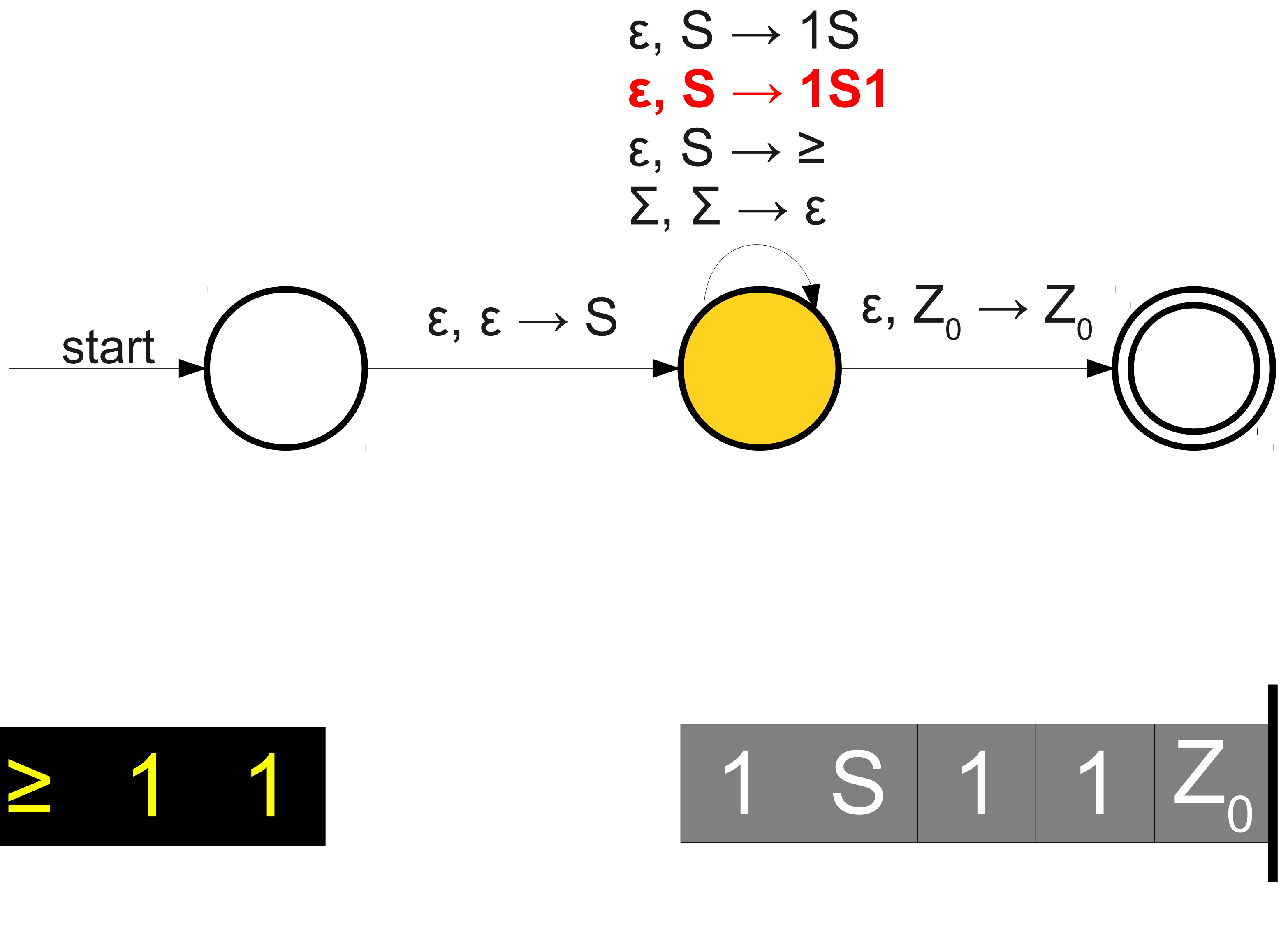
$S \rightarrow 1S$

$S \rightarrow$ ≥



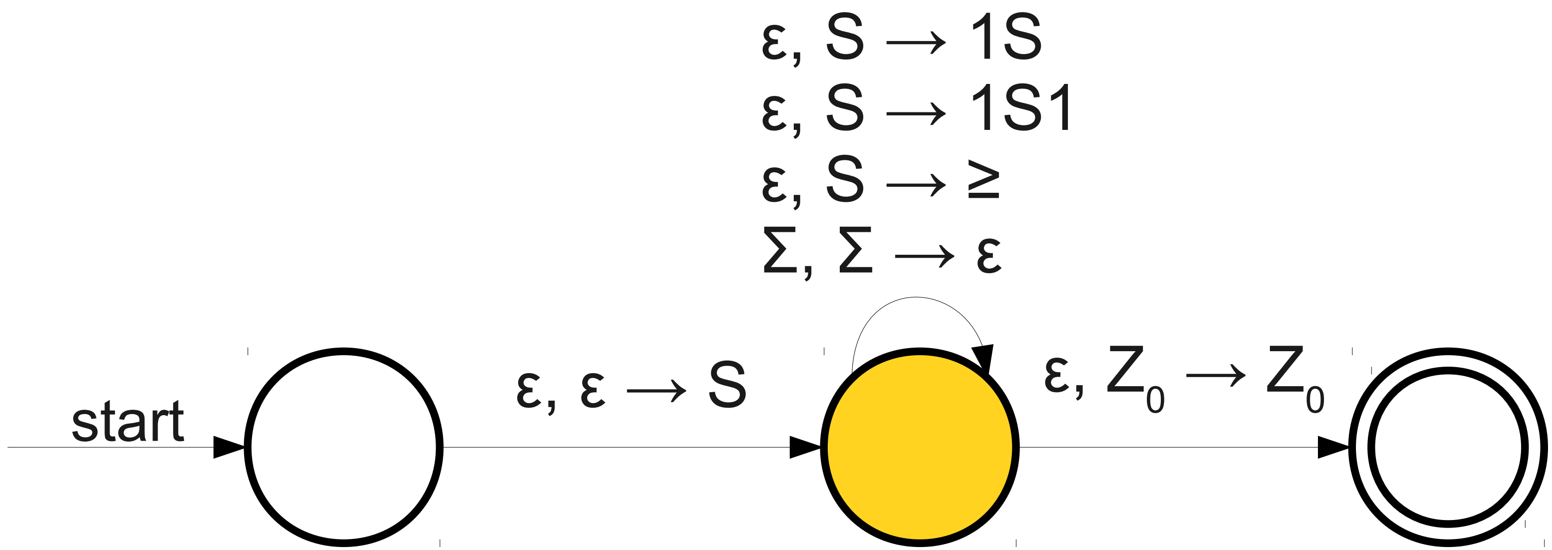
From CFGs to PDAs

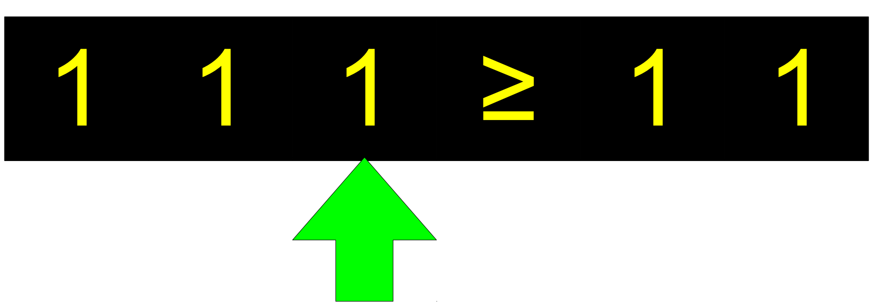
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



From CFGs to PDAs

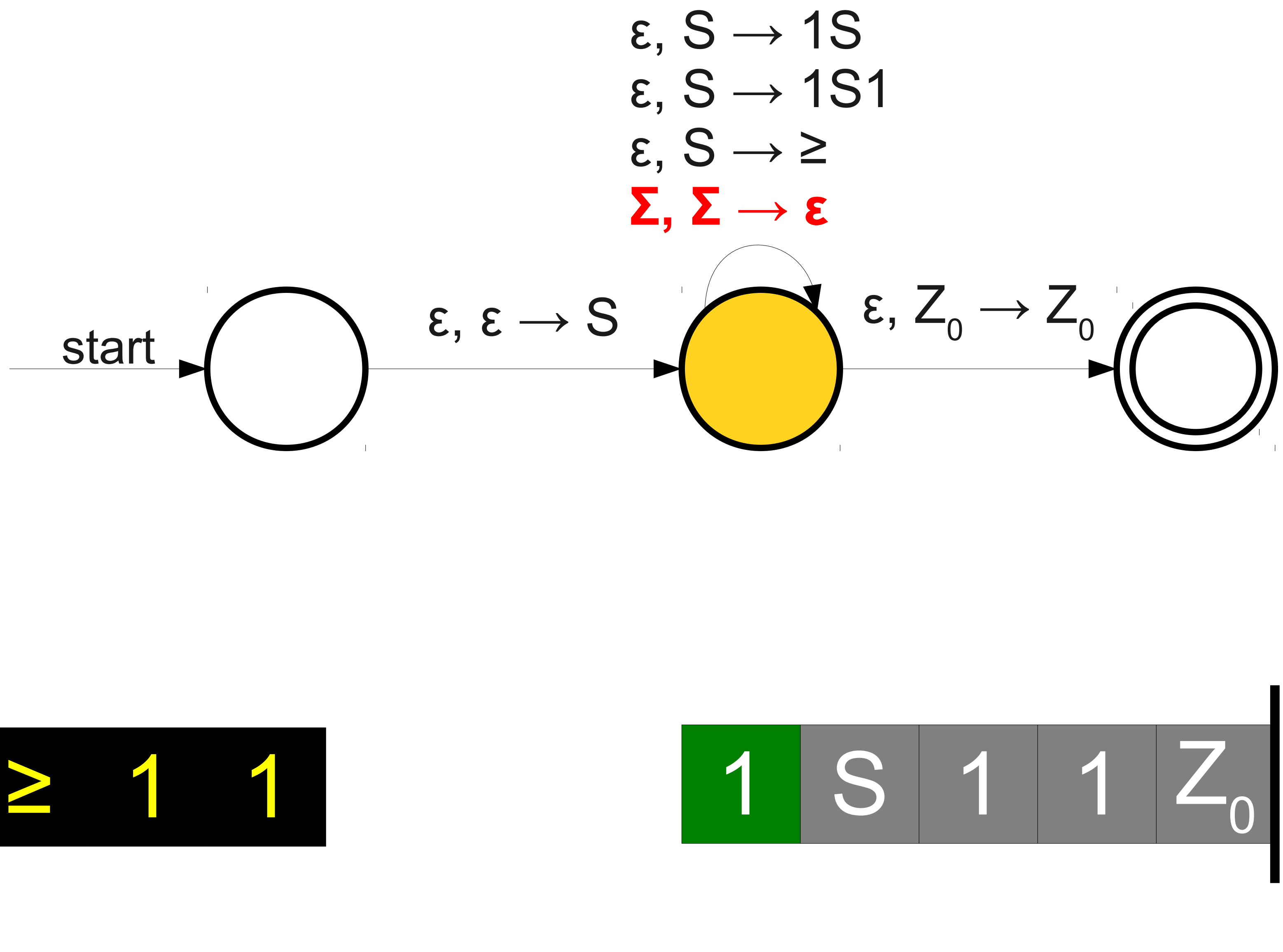
S → 1S1
S → 1S
S → ≥





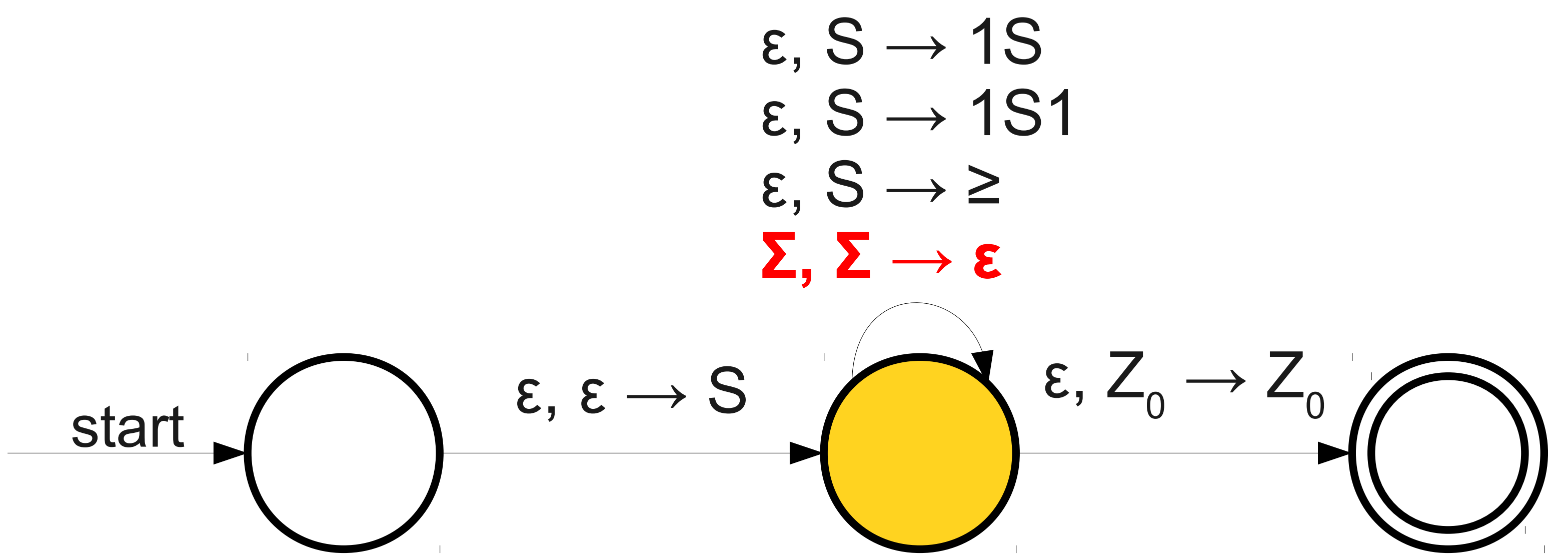
From CFGs to PDAs

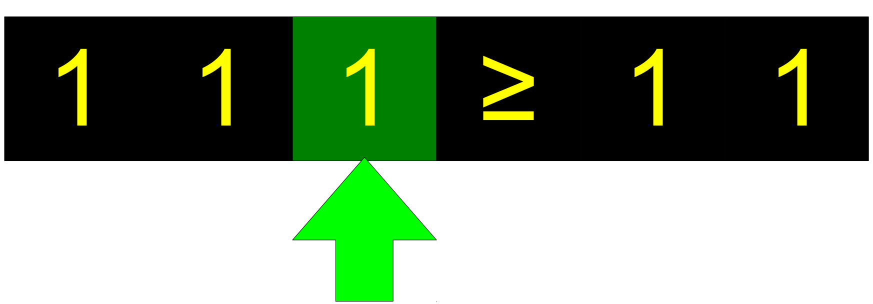
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \varepsilon$



From CFGs to PDAs

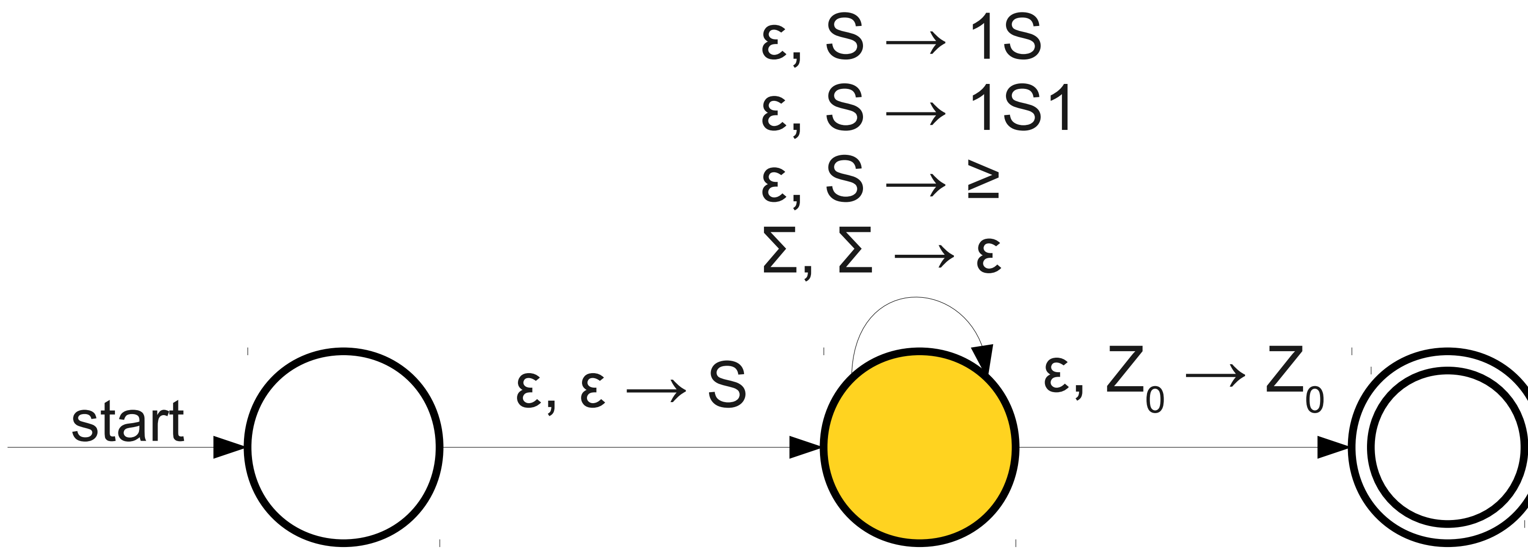
S → 1S1
S → 1S
S → ≥

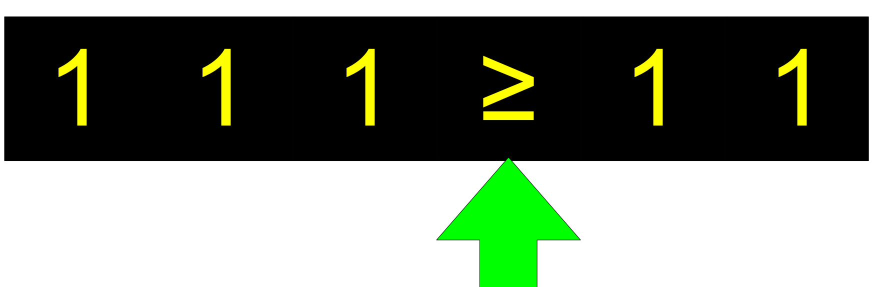




From CFGs to PDAs

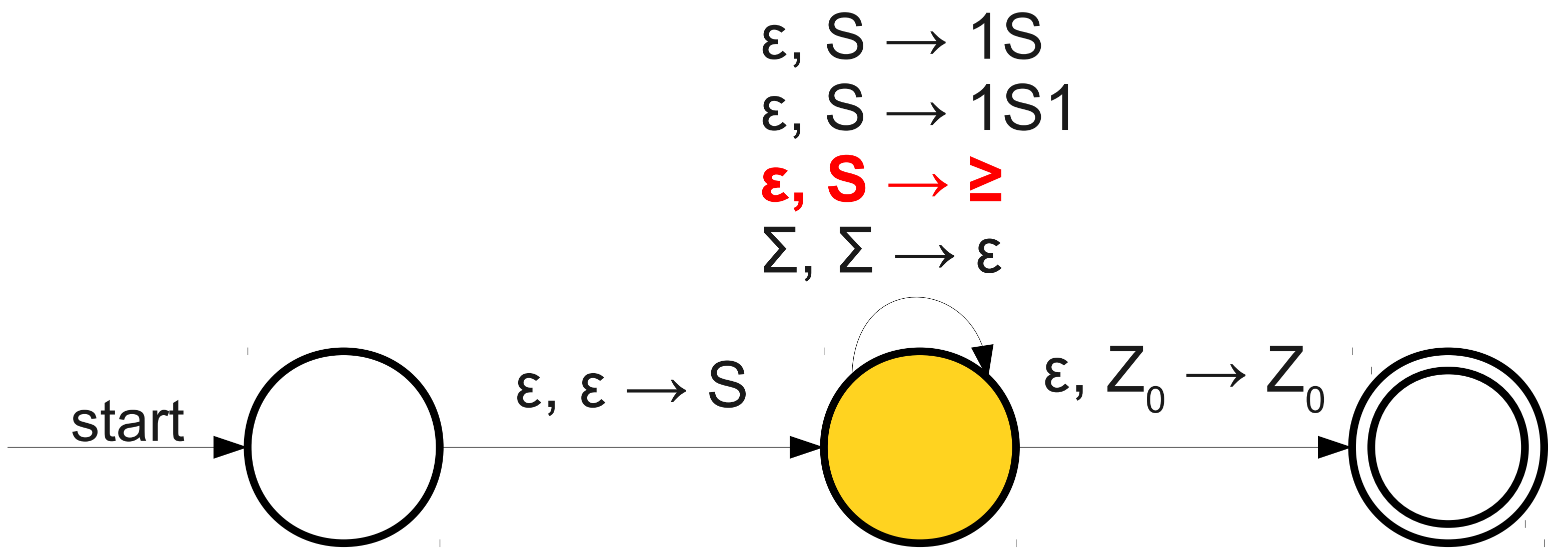
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

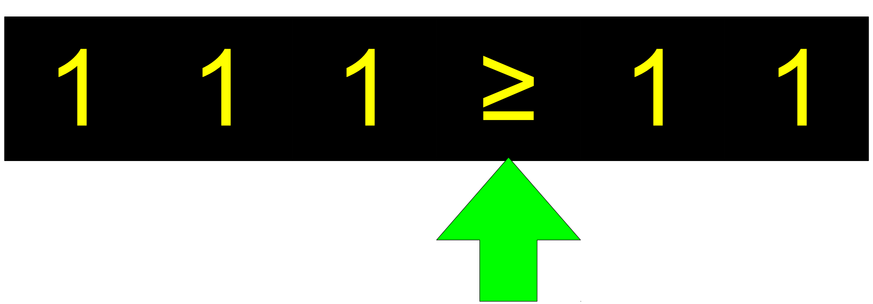




From CFGs to PDAs

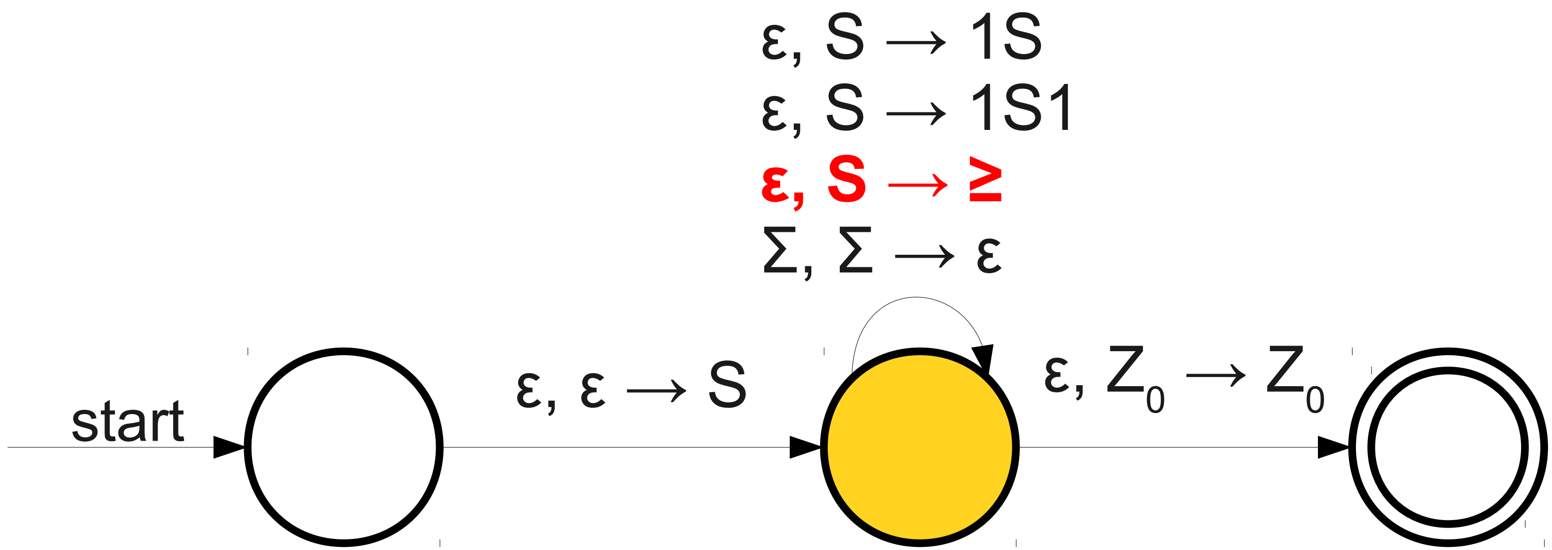
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

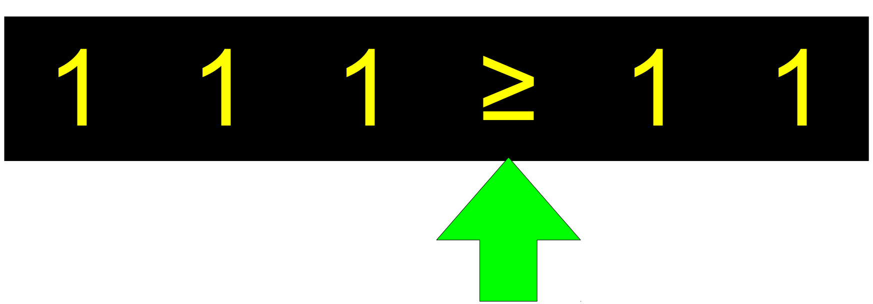




From CFGs to PDAs

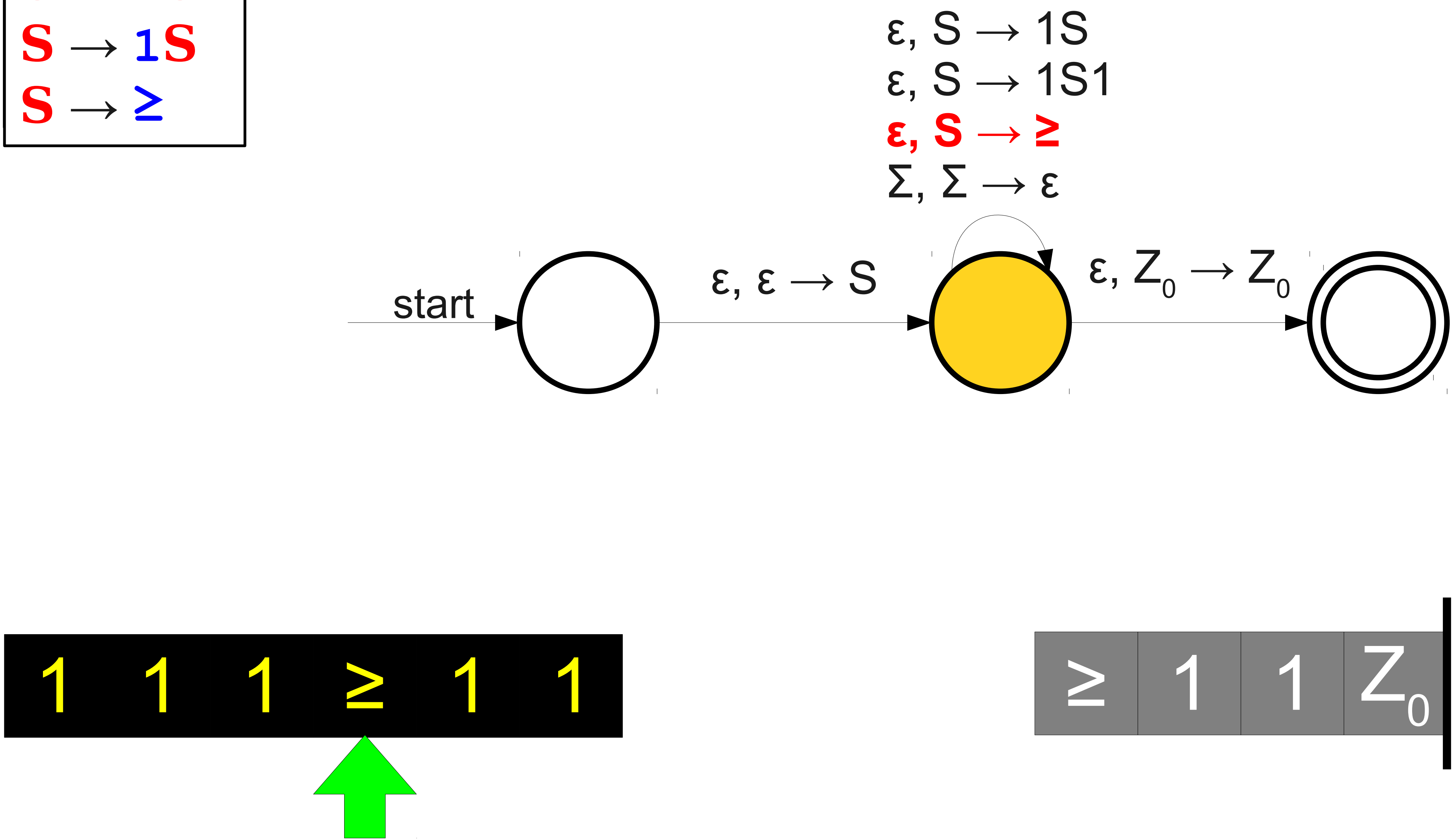
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$





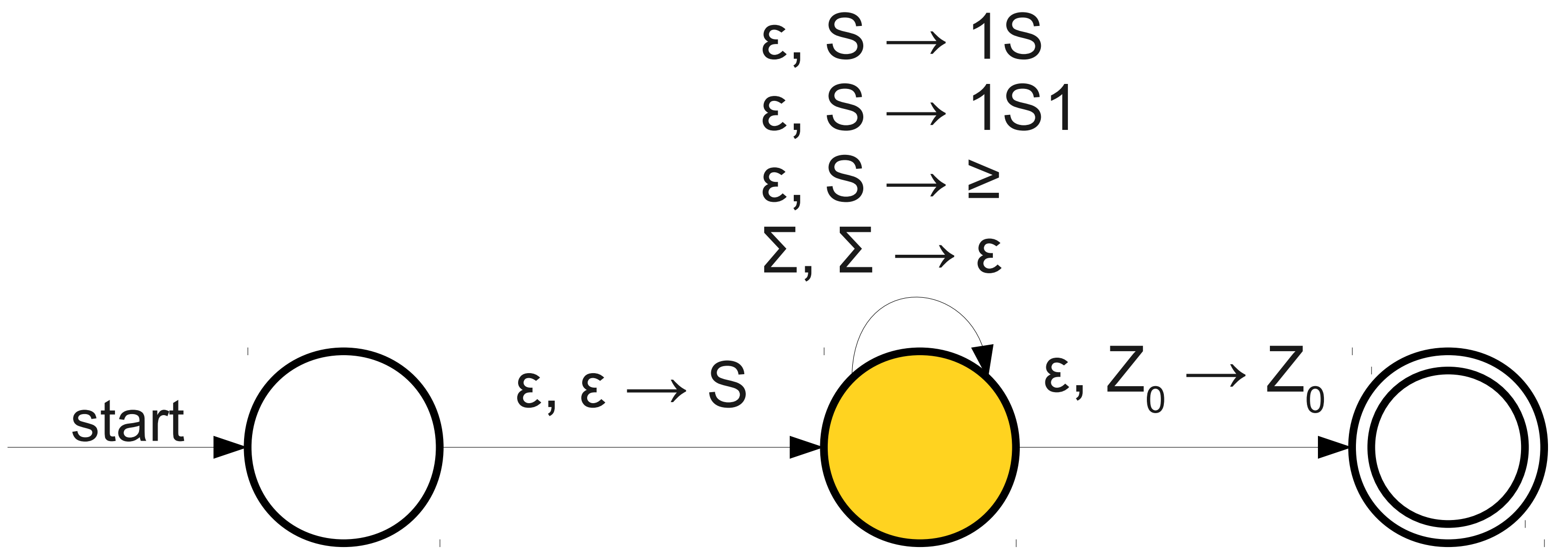
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to \geq$



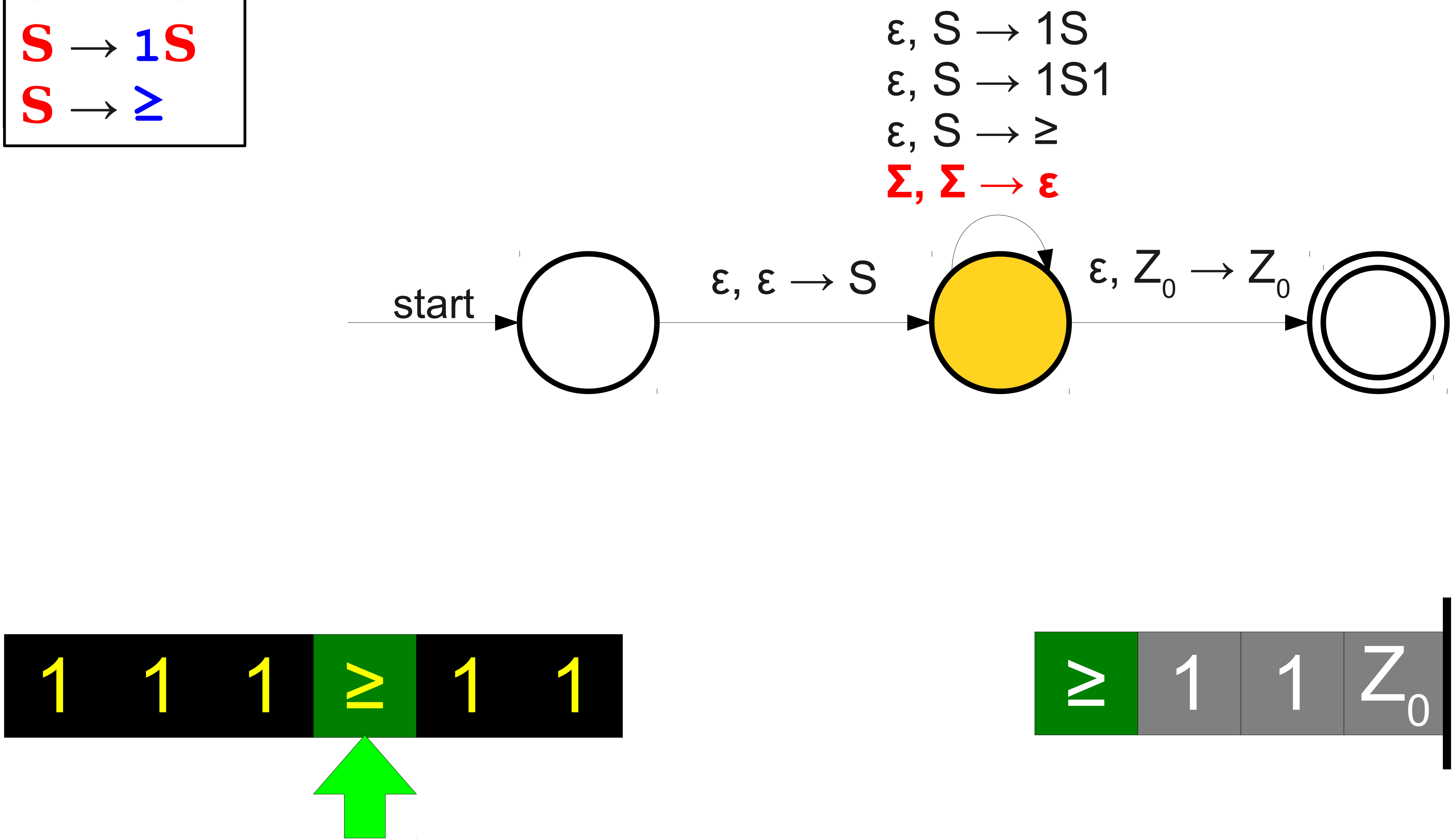
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$



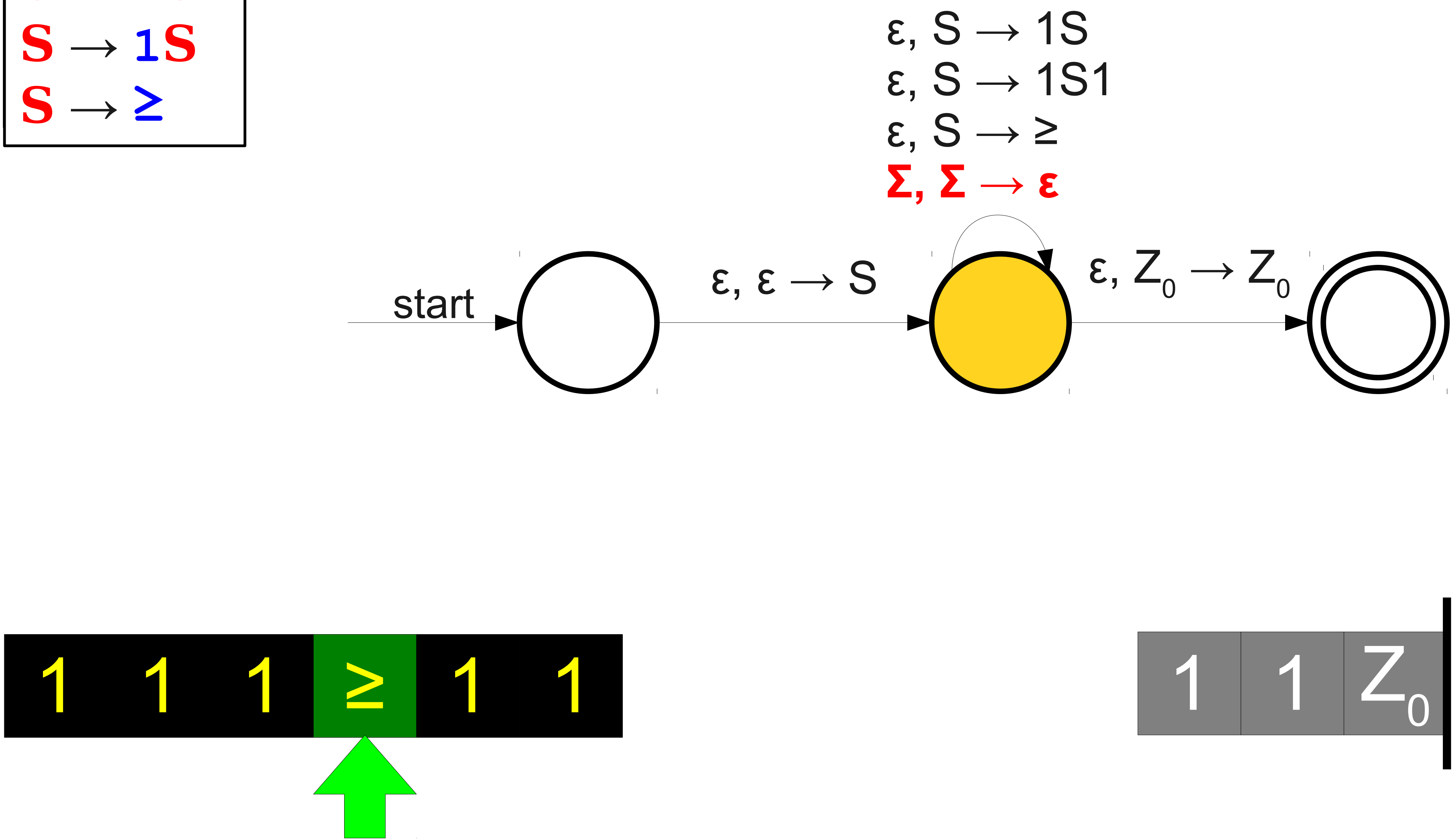
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



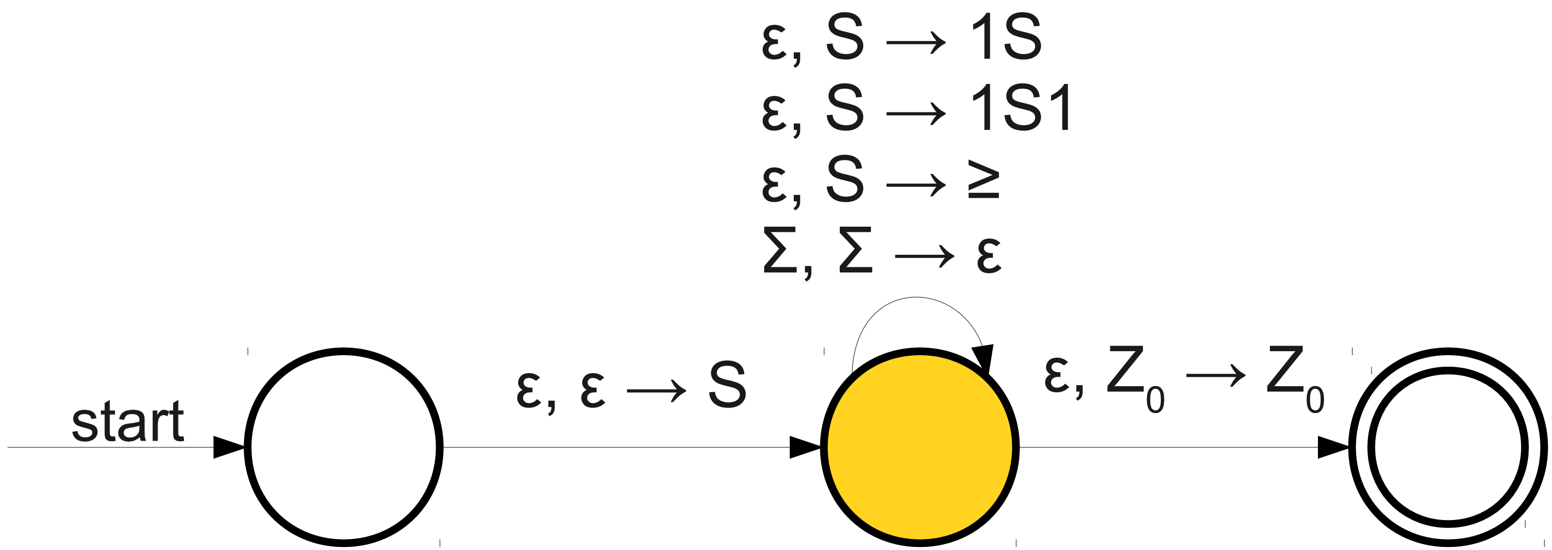
From CFGs to PDAs

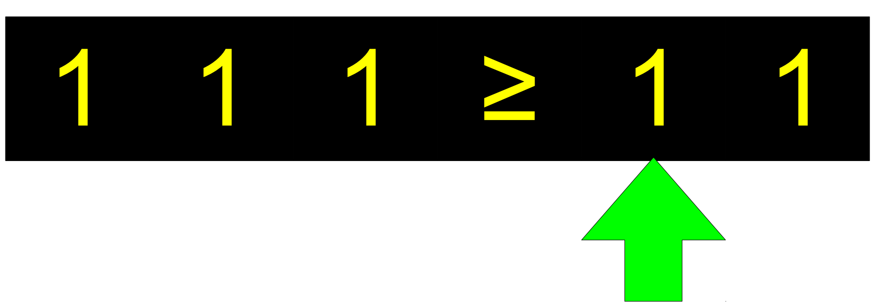
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

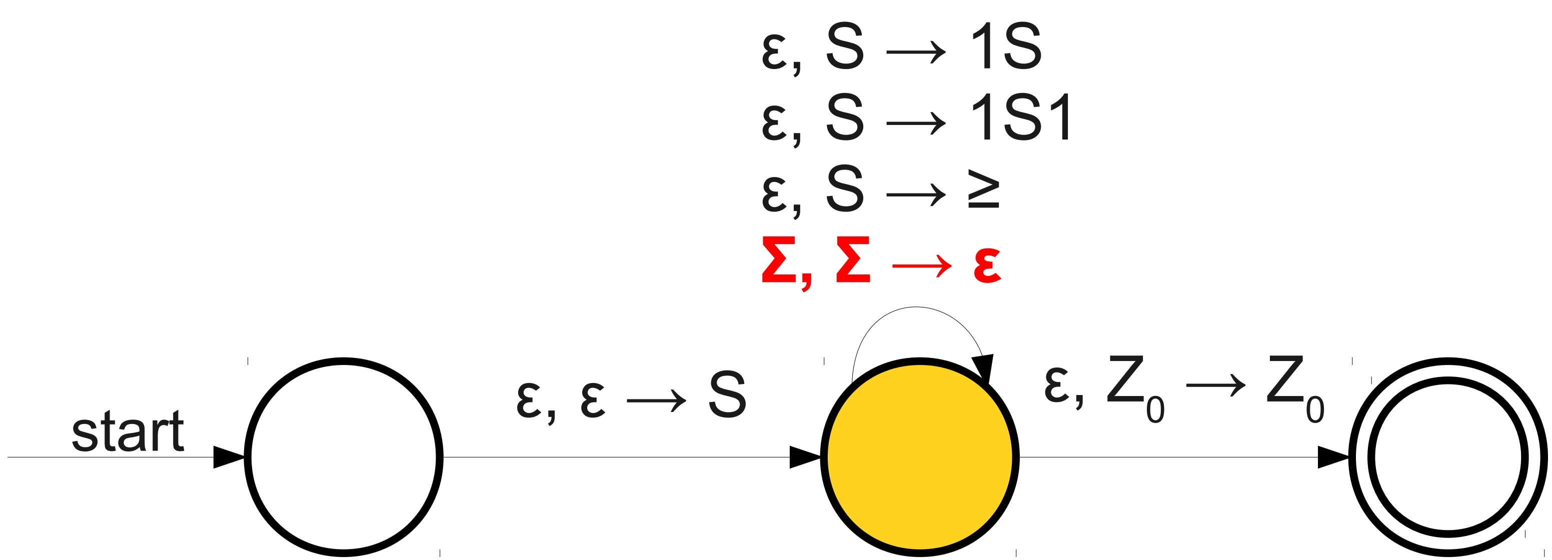
S → 1S1
S → 1S
S → ≥





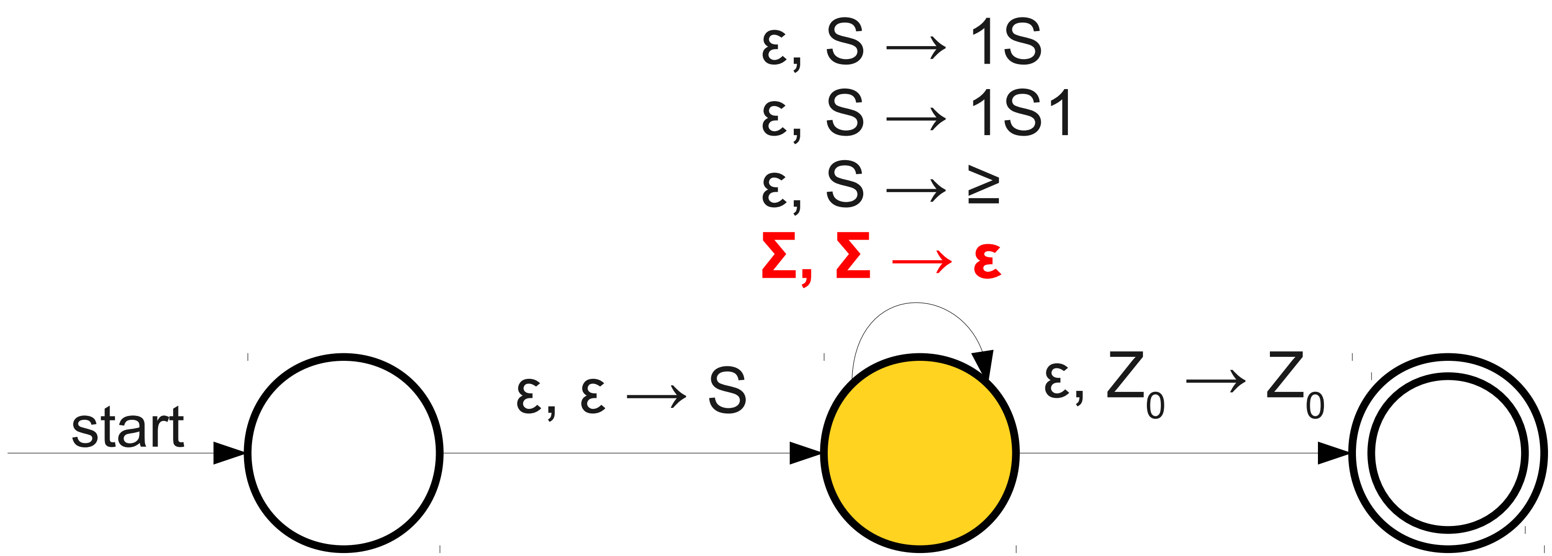
From CFGs to PDAs

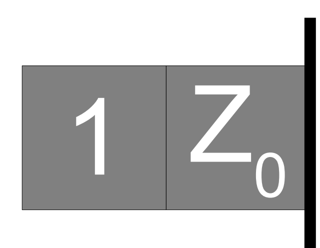
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

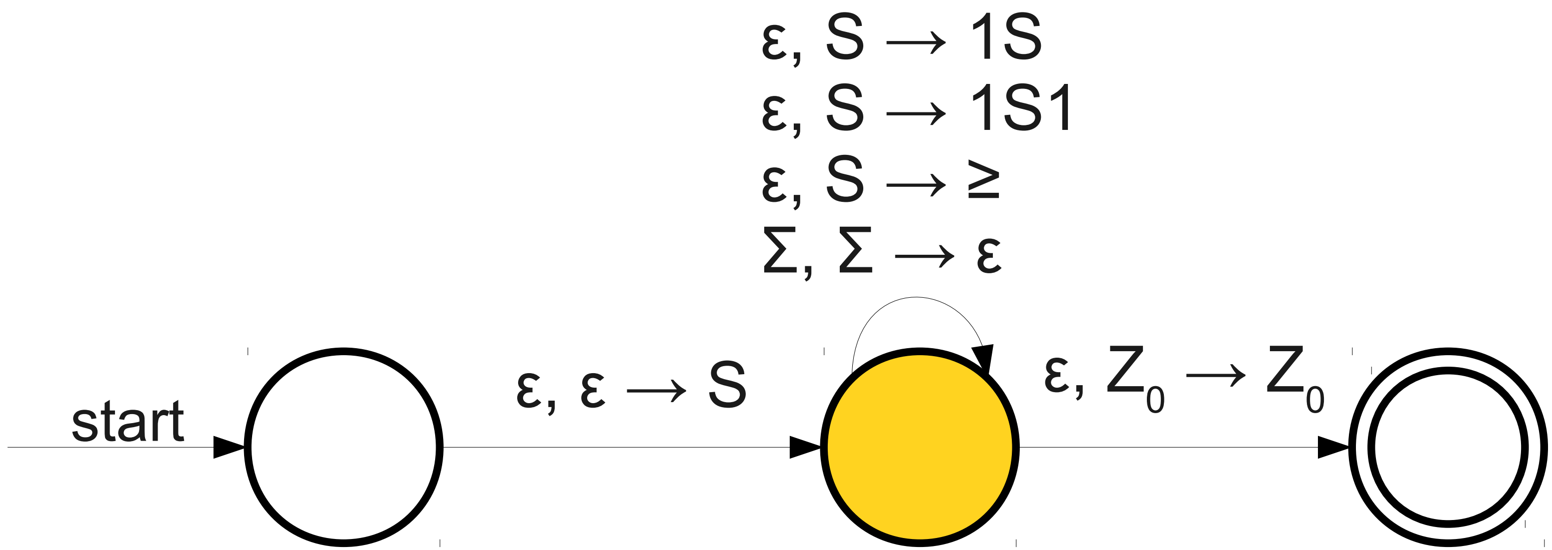
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥

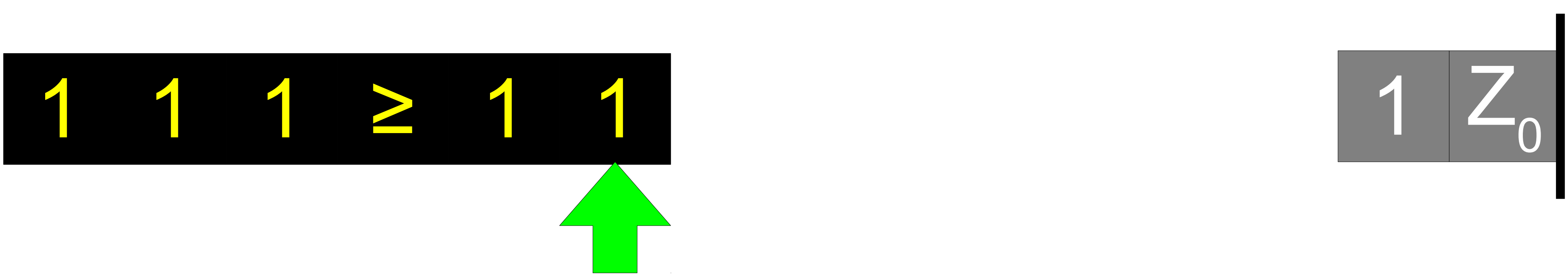




From CFGs to PDAs

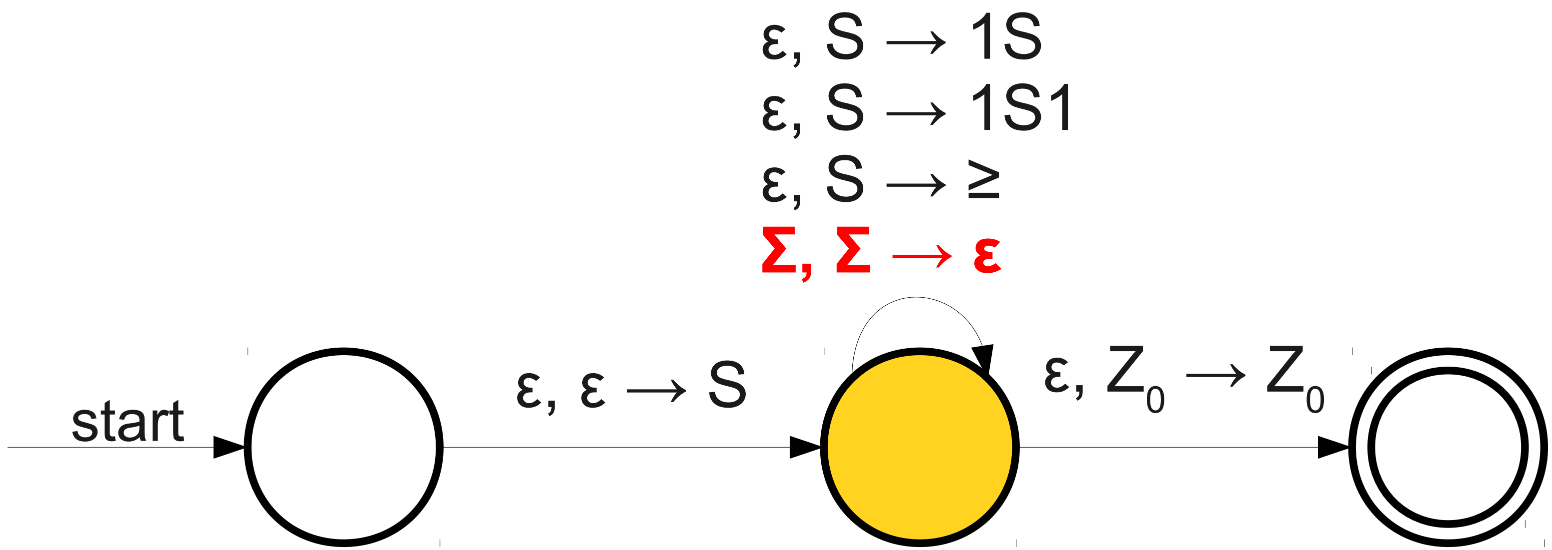
S → **1S1**
S → **1S**
S → **≥**

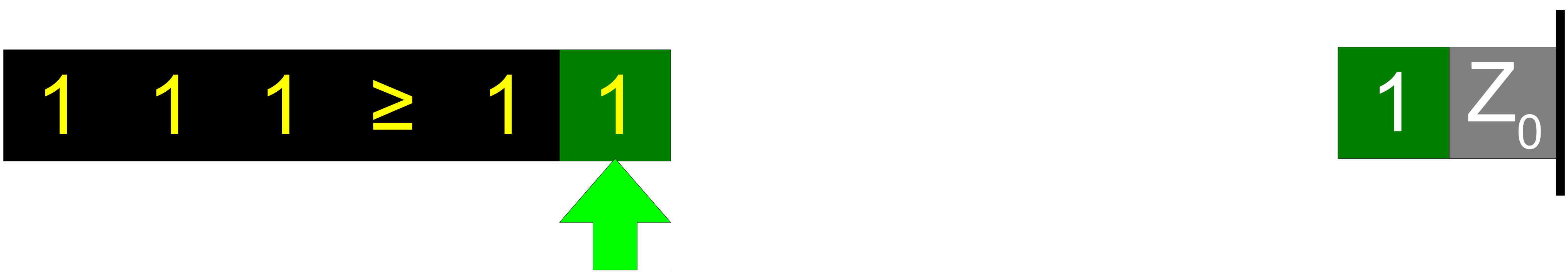




From CFGs to PDAs

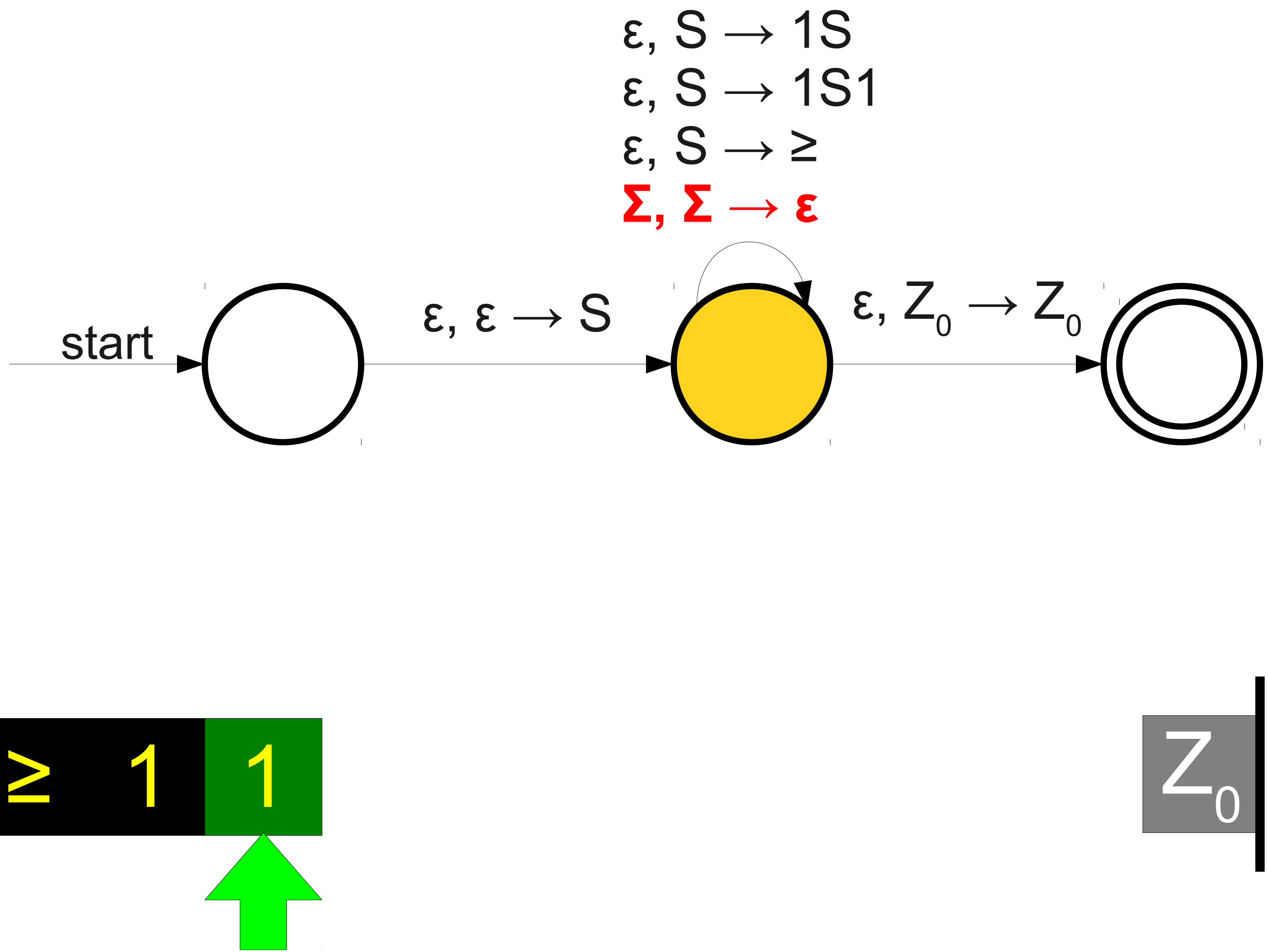
S → 1S1
S → 1S
S → ≥





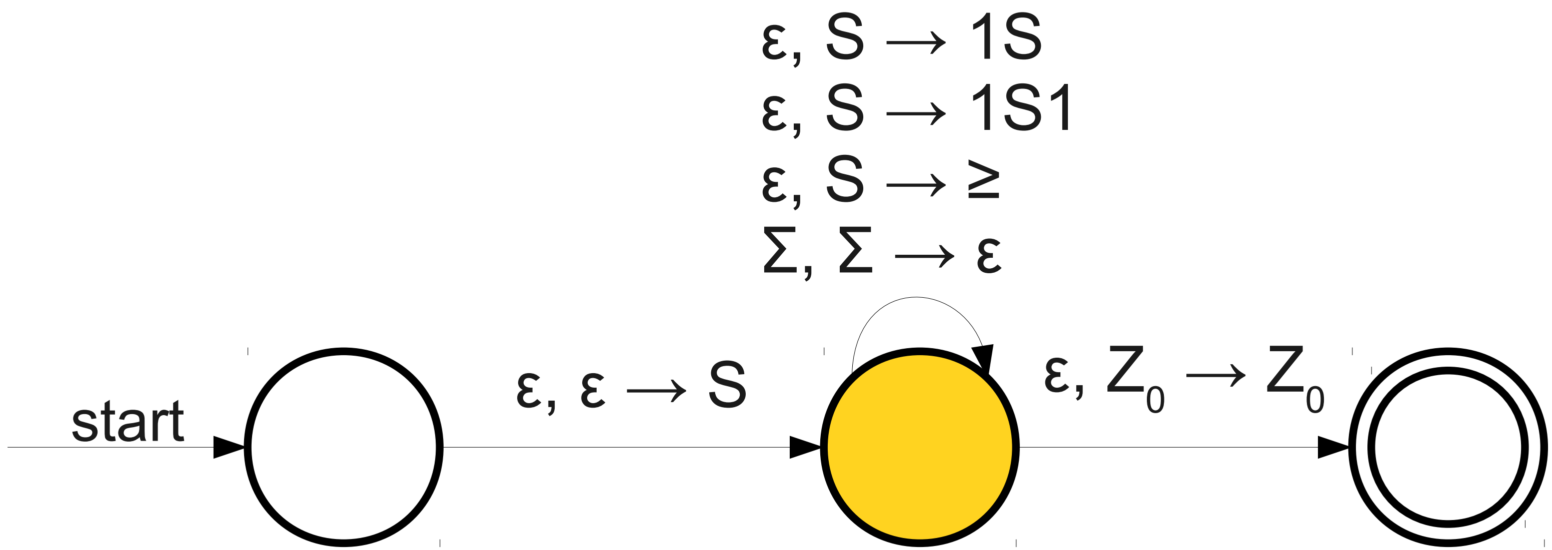
From CFGs to PDAs

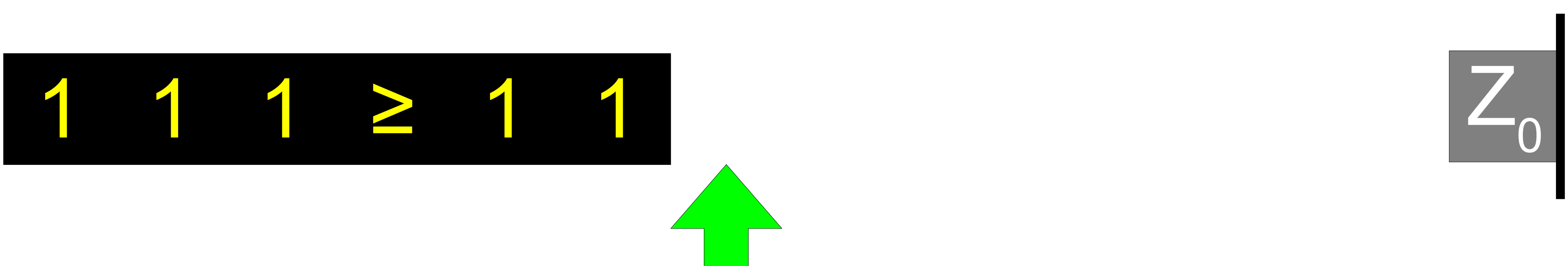
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



From CFGs to PDAs

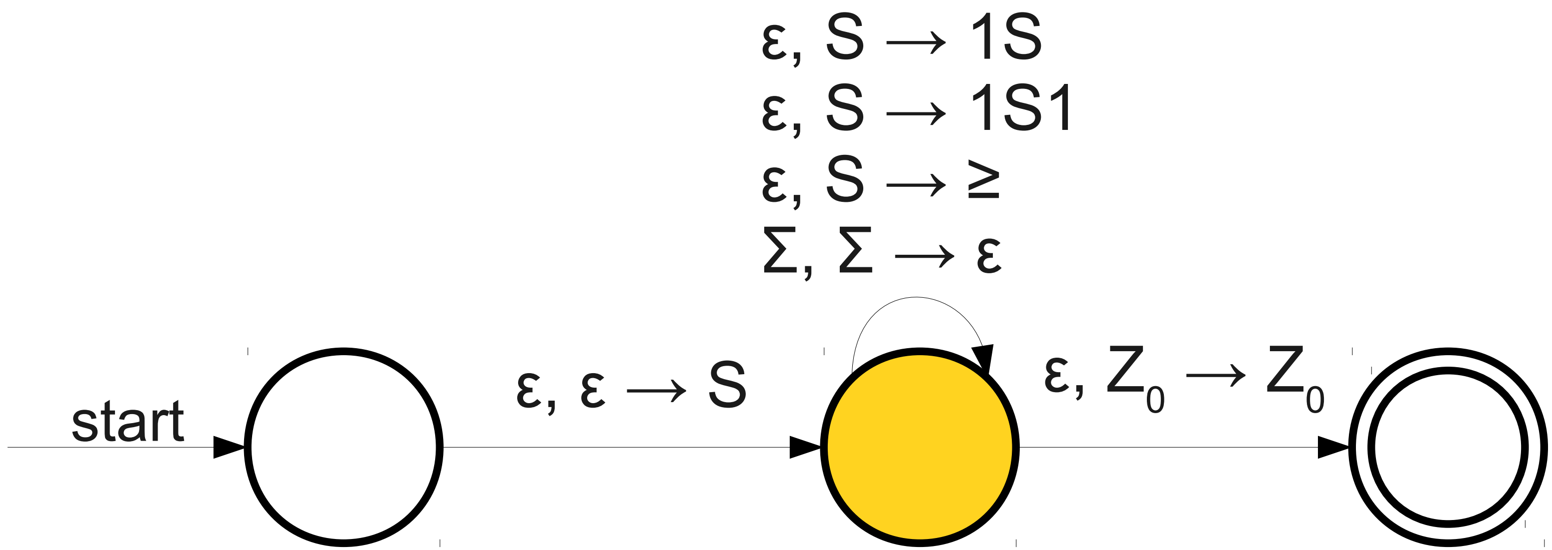
$S \to 1S1$
$S \to 1S$
$S \to ≥$



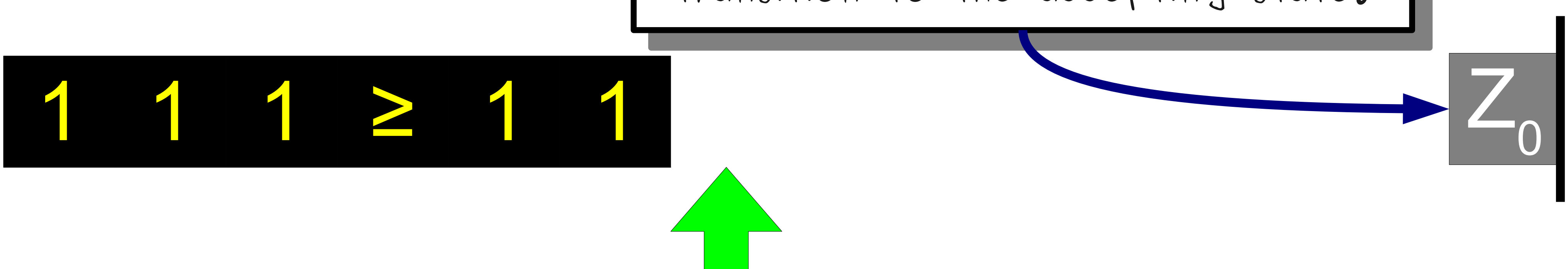


From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to$ ≥

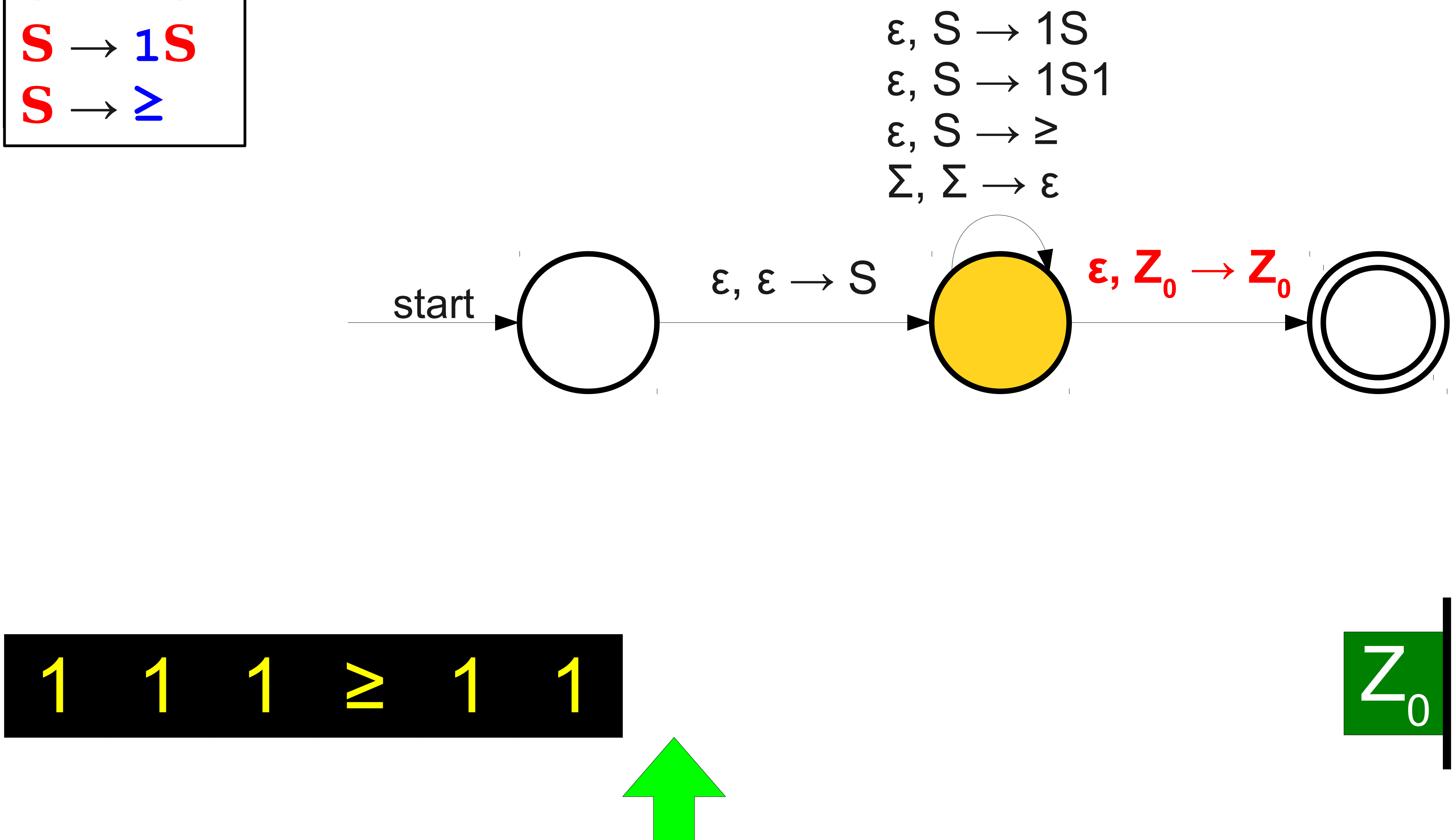


At this point we've completely matched the string, so it's time to transition to the accepting state.



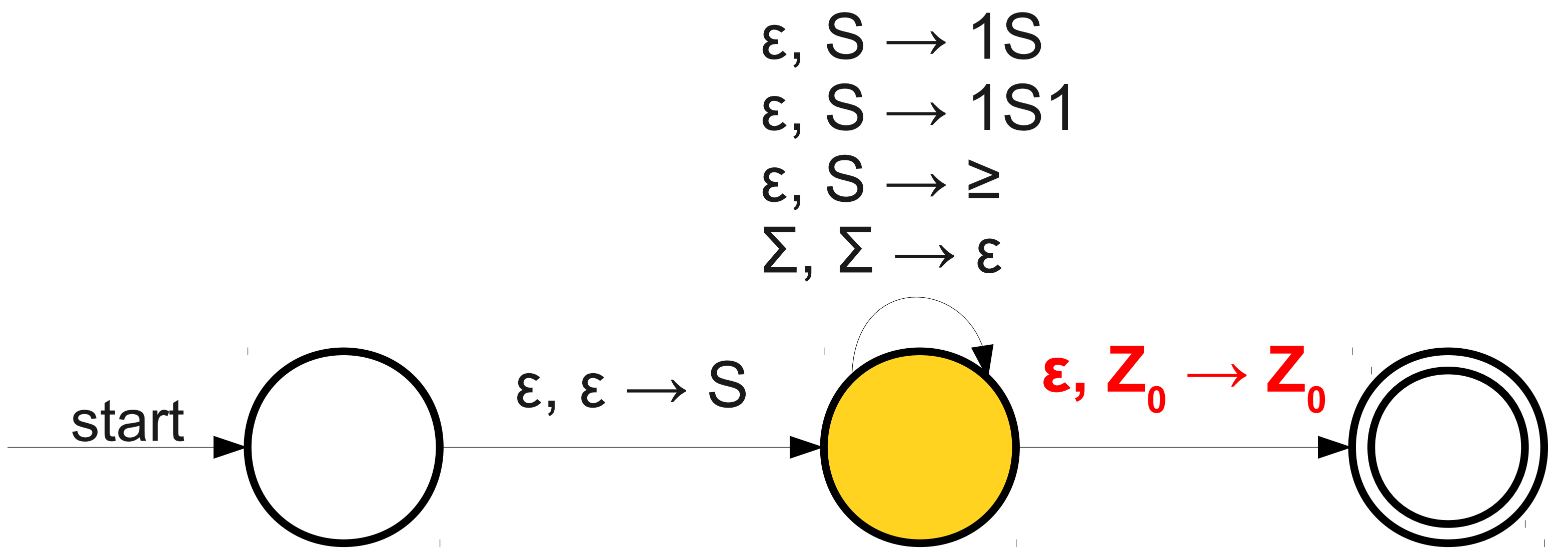
From CFGs to PDAs

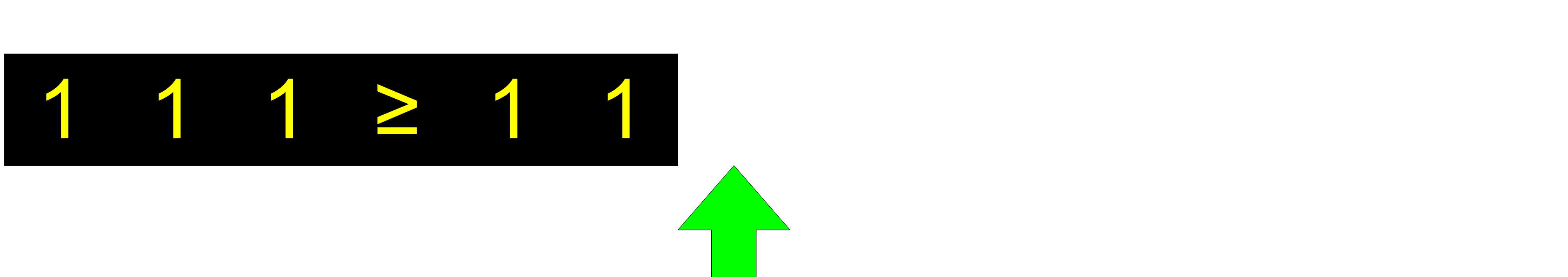
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



From CFGs to PDAs

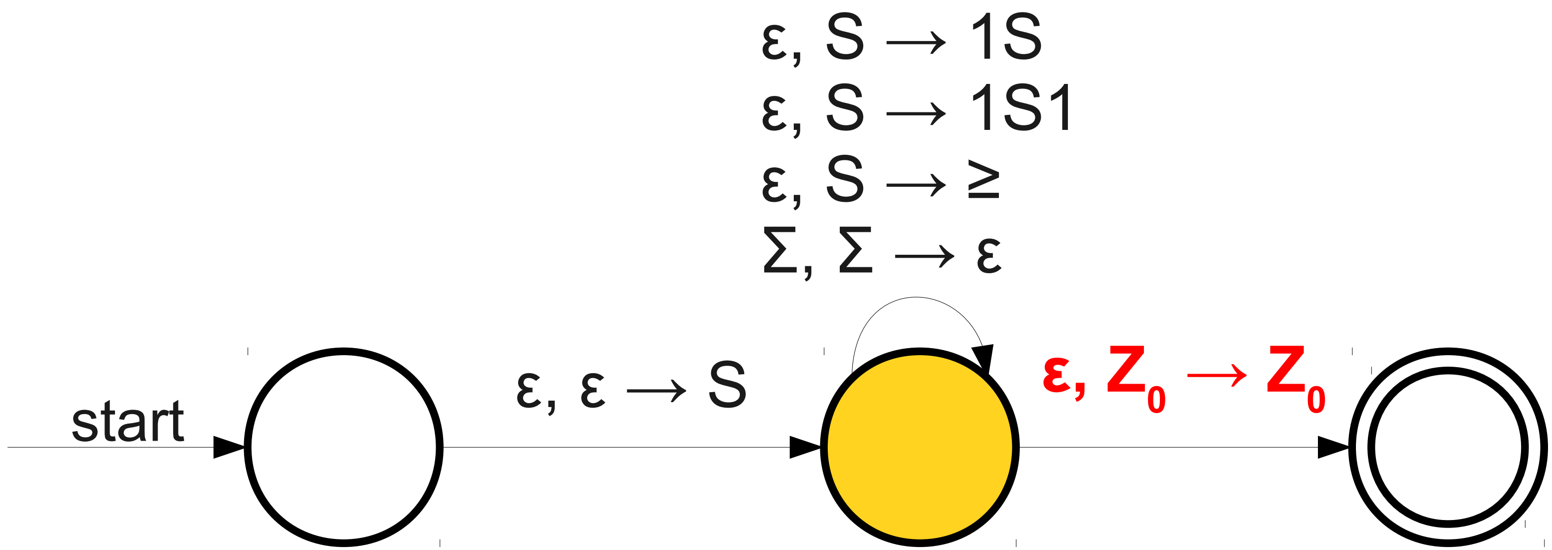
S → 1S1
S → 1S
S → ≥

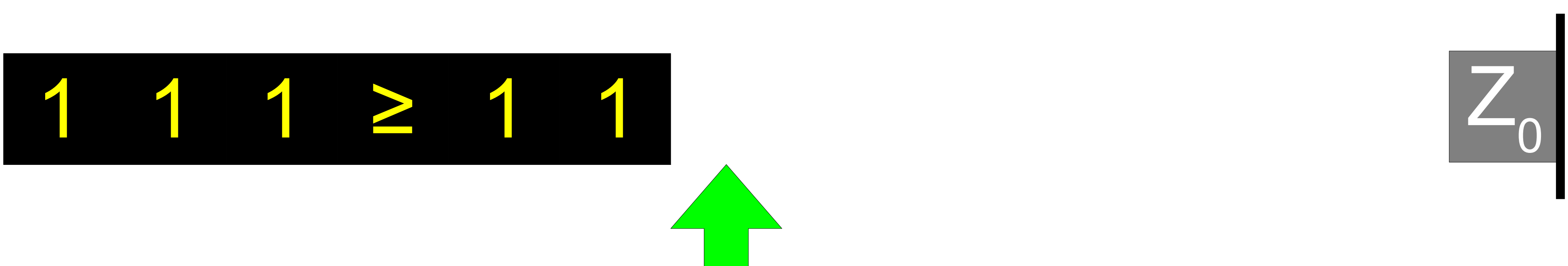




From CFGs to PDAs

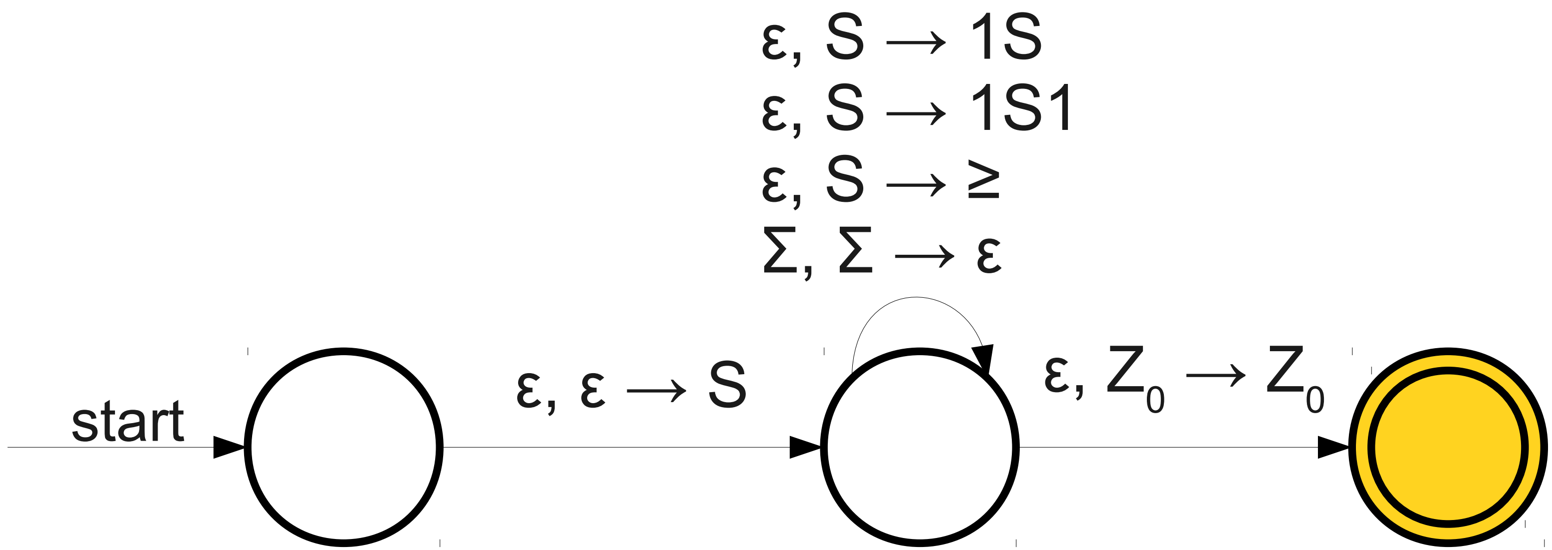
$S \to 1S1$
$S \to 1S$
$S \to ≥$





From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

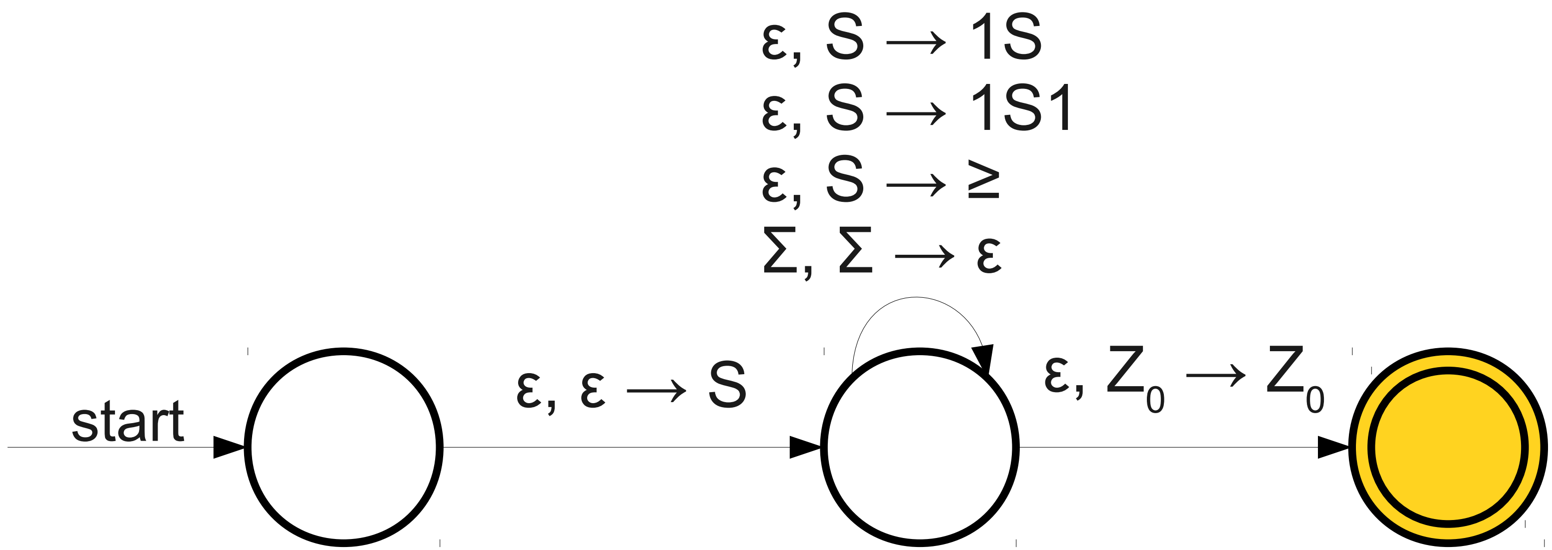


1 1 1 ≥ 1 1

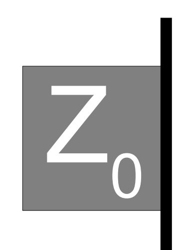
Z_0

From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$

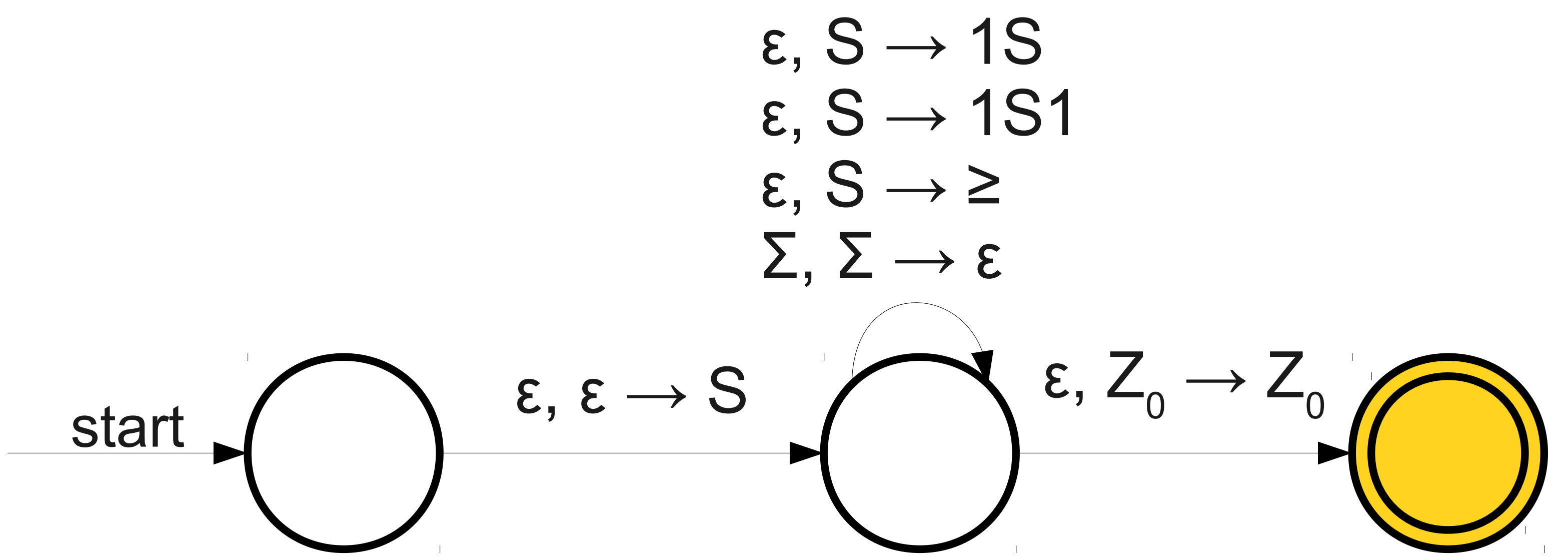


1 1 1 ≥ 1 1

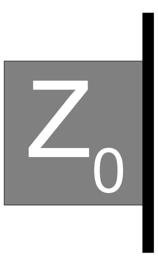


From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



1 1 1 ≥ 1 1



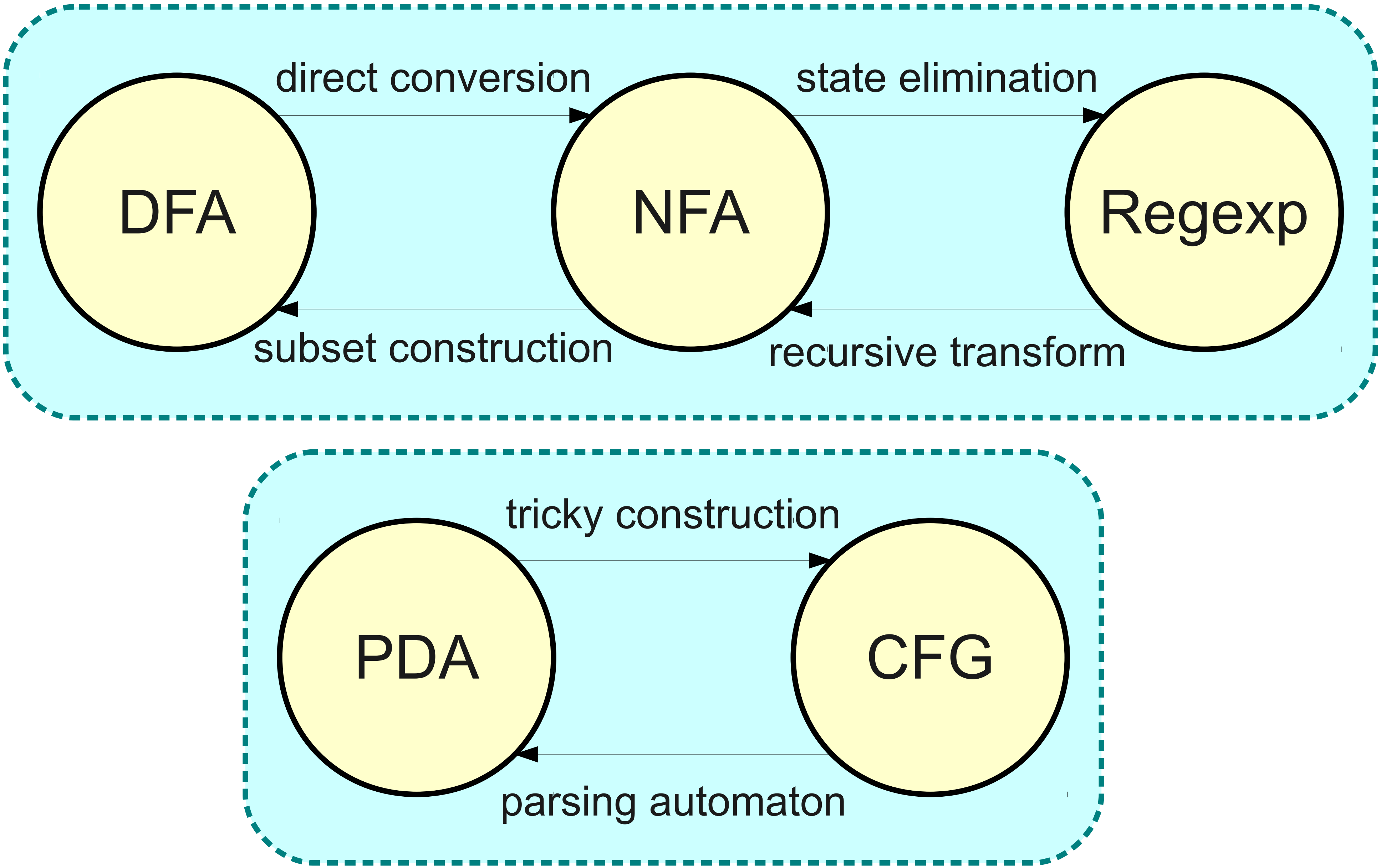
From CFGs to PDAs

- Make three states: **start**, **parsing**, and **accepting**.
- There is a transition ε, ε → **S** from **start** to **parsing**.
  - Corresponds to starting off with the start symbol S.
- There is a transition ε, **A** → ***ω*** from **parsing** to itself for each production **A** → ***ω***.
  - Corresponds to predicting which production to use.
- There is a transition Σ, Σ → ε from **parsing** to itself.
  - Corresponds to matching a character of the input.
- There is a transition ε, Z_0 → Z_0 from **parsing** to **accepting**.
  - Corresponds to completely matching the input.

From PDAs to CFGs

- The other direction of the proof (converting a PDA to a CFG) is much harder.
- Intuitively, create a CFG representing paths between states in the PDA.
- Lots of tricky details, but a marvelous proof.
  - It's just too large to fit into the margins of this slide.
- Read Sipser for more details.

Our Transformations



The Limits of Context-Free Languages

The Pumping Lemma for Regular Languages

- Let L be a regular language, so there is a DFA D for L.
- A sufficiently long string $w \in L$ must visit some state in D twice.
- This means w went through a loop in the D.
- By replicating the characters that went through the loop in the D, we can “pump” a portion of w to produce new strings in the language.

The Pumping Lemma Intuition

- The model of computation used has a finite description.
- For sufficiently long strings, the model of computation must repeat some step of the computation to recognize the string.
- Under the right circumstances, we can iterate this repeated step zero or more times to produce more and more strings.

Parse Trees

E

E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

E

E

E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

E

E

⇒ E Op E

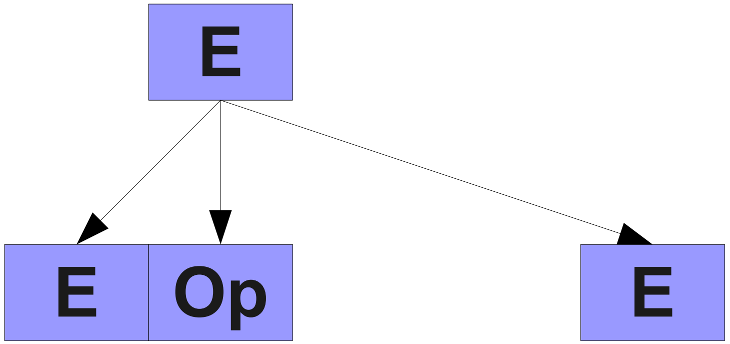
E → E Op E | `int` | (E)

Op → `+` | `*` | `-` | `/`

Parse Trees

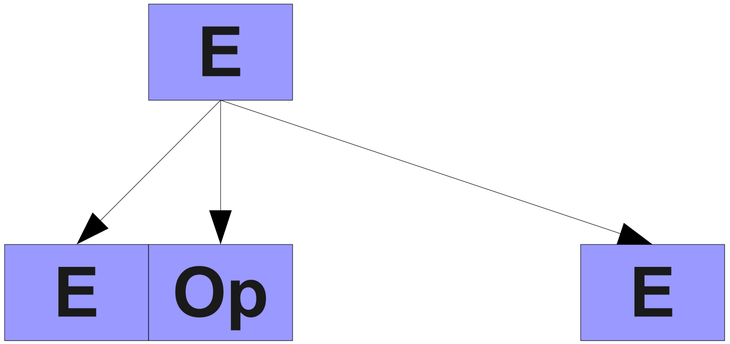
E

⇒ E Op E



$E \rightarrow E\ Op\ E$ | `int` | (E)
$Op \rightarrow$ `+` | `*` | `-` | `/`

Parse Trees



E

⇒ E Op E

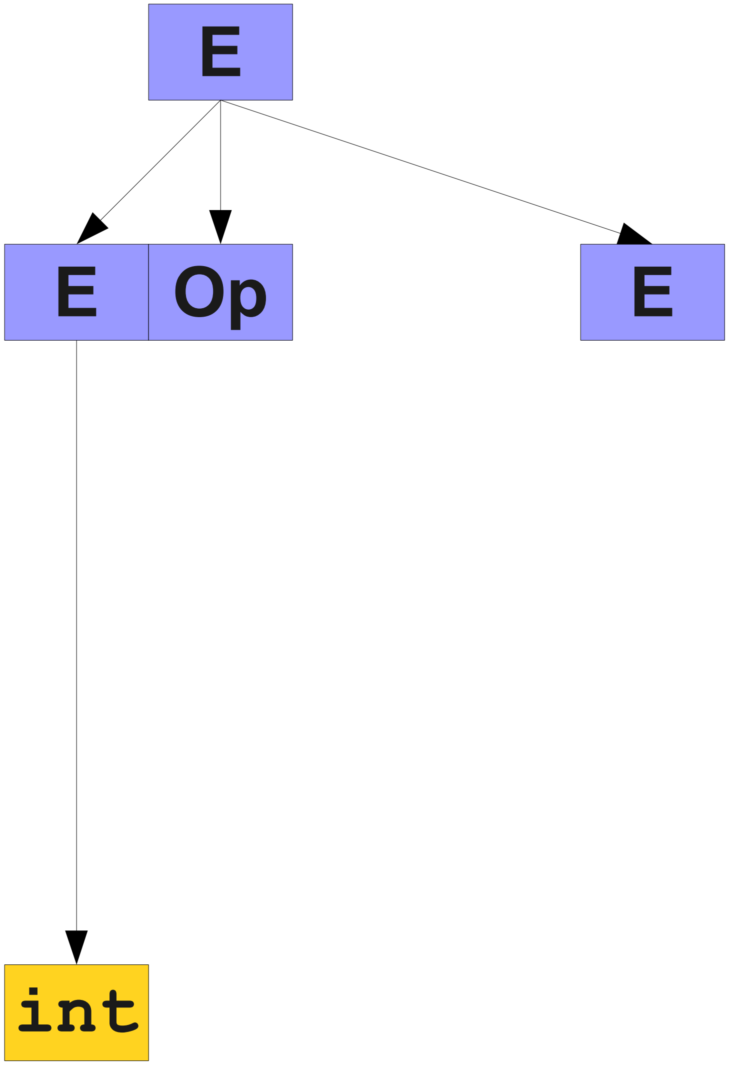
⇒ `int` Op E

E → E Op E | `int` | (E)

Op → `+` | `*` | `-` | `/`

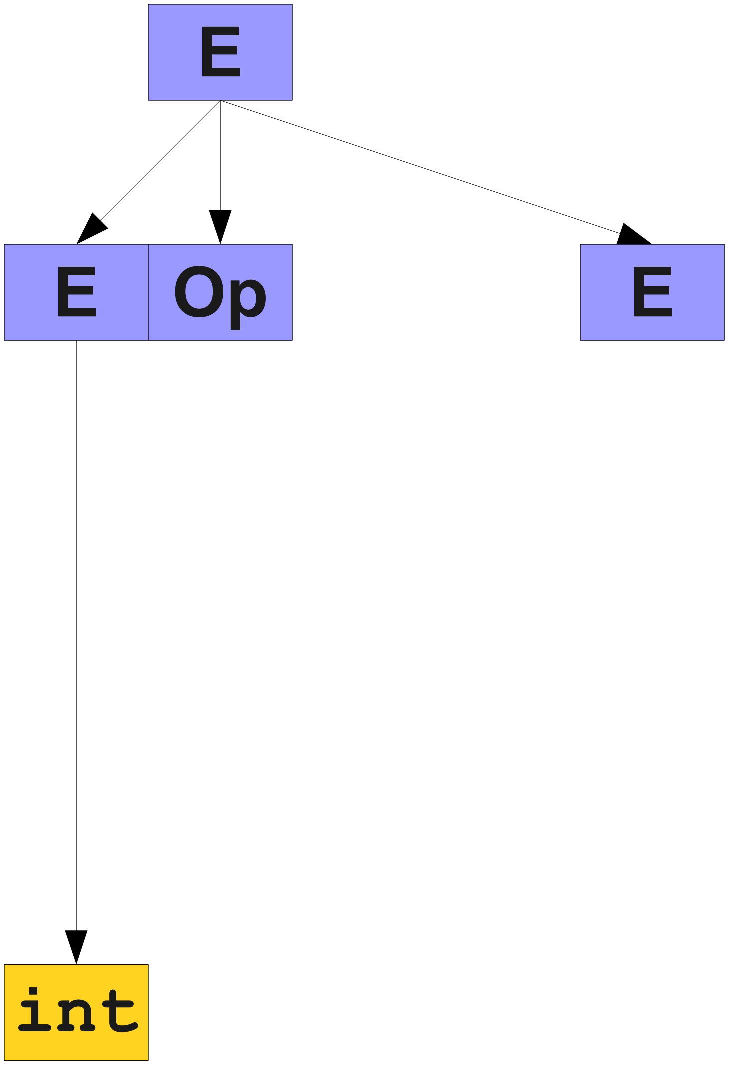
Parse Trees

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \texttt{int}\ \mathbf{Op\ E}
\end{aligned}
$$



$$
\begin{aligned}
\mathbf{E} &\to \mathbf{E\ Op\ E} \mid \texttt{int} \mid (\mathbf{E}) \\
\mathbf{Op} &\to \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/}
\end{aligned}
$$

Parse Trees



E

⇒ **E Op E**

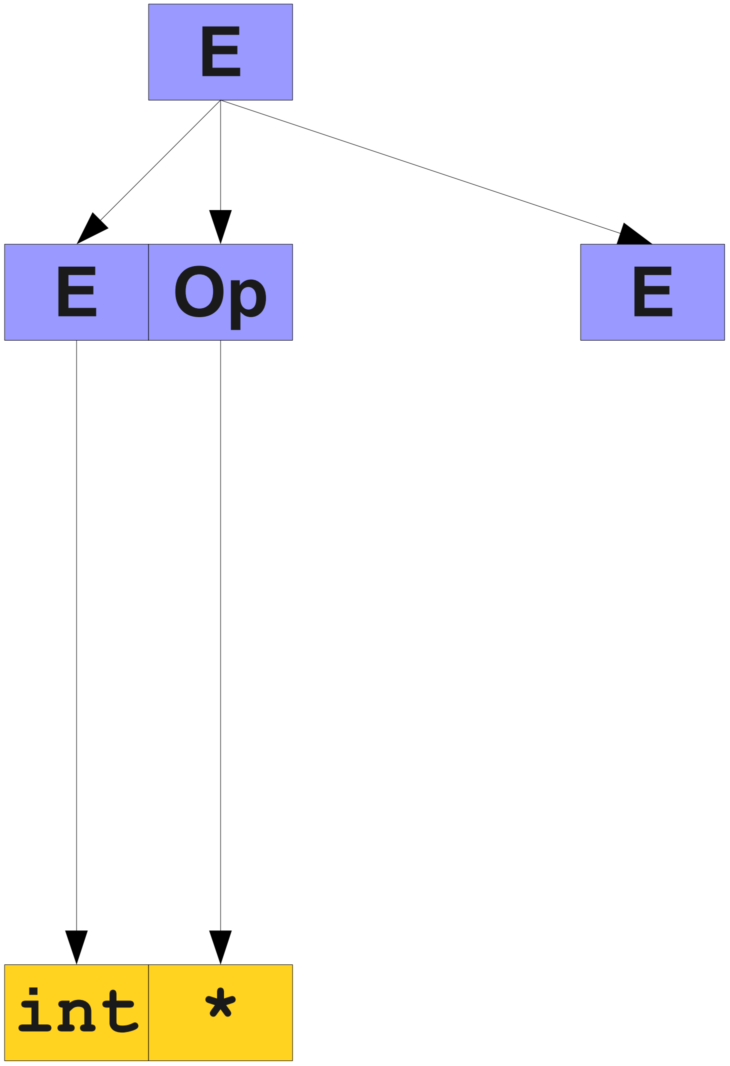
⇒ `int` **Op E**

⇒ `int` `*` **E**

E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

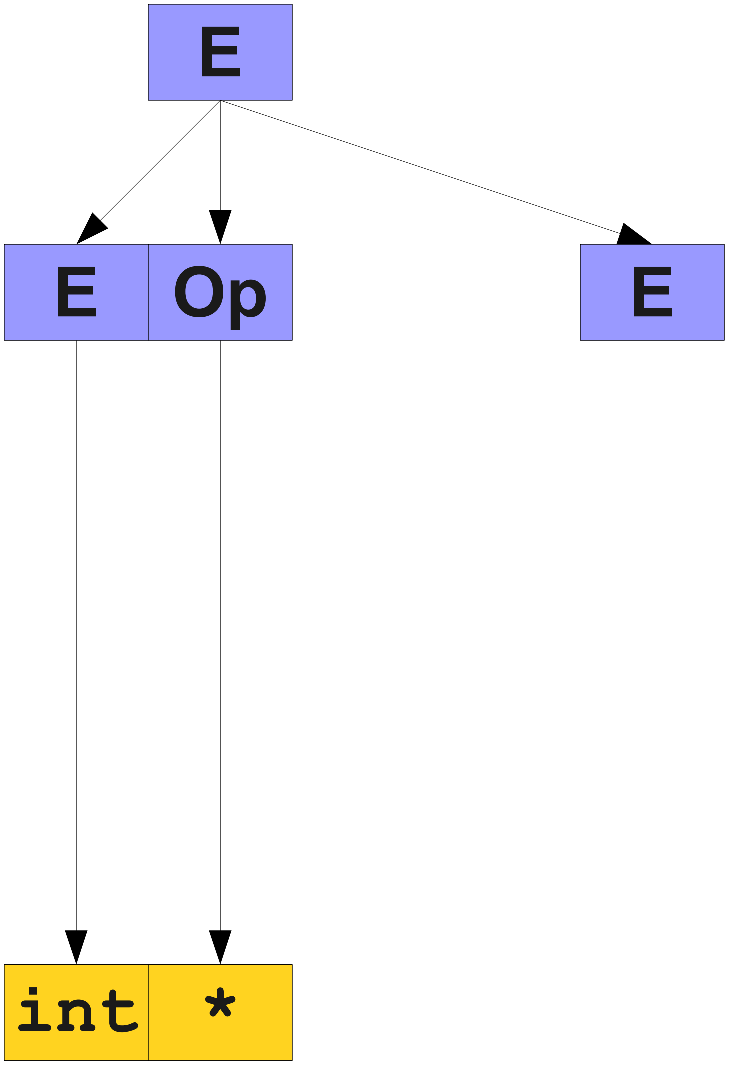
$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \mathtt{int}\ \mathbf{Op\ E} \\
\Rightarrow\ & \mathtt{int}\ \mathtt{*}\ \mathbf{E}
\end{aligned}
$$



$\mathbf{E} \to \mathbf{E\ Op\ E} \mid \mathtt{int} \mid (\mathbf{E})$

$\mathbf{Op} \to \mathtt{+} \mid \mathtt{*} \mid \mathtt{-} \mid \mathtt{/}$

Parse Trees



E

⇒ **E Op E**

⇒ `int` **Op E**

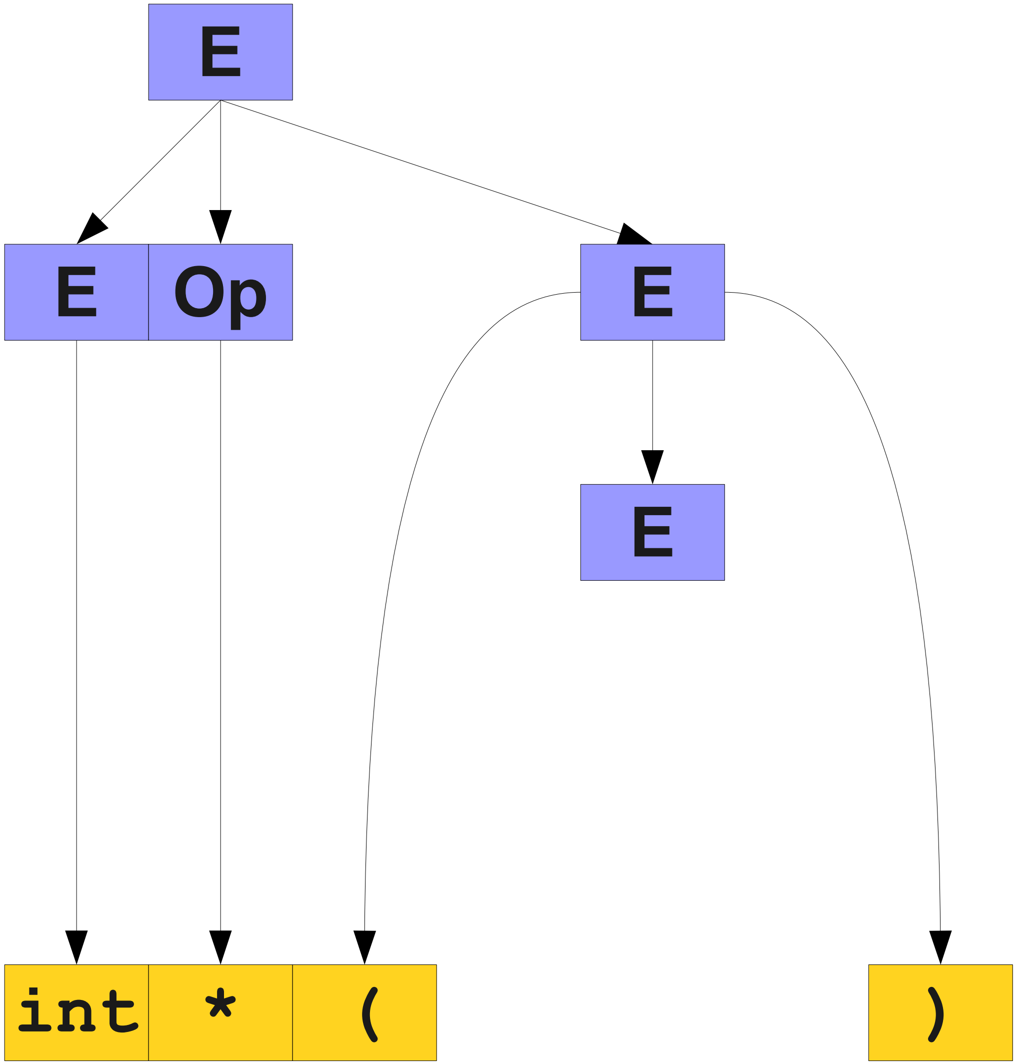
⇒ `int * ` **E**

⇒ `int * (`**E**`)`

E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \texttt{int}\ \mathbf{Op\ E} \\
\Rightarrow\ & \texttt{int * } \mathbf{E} \\
\Rightarrow\ & \texttt{int * (}\mathbf{E}\texttt{)}
\end{aligned}
$$



$\mathbf{E} \rightarrow \mathbf{E\ Op\ E} \mid \texttt{int} \mid (\mathbf{E})$

$\mathbf{Op} \rightarrow \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/}$

Parse Trees

E

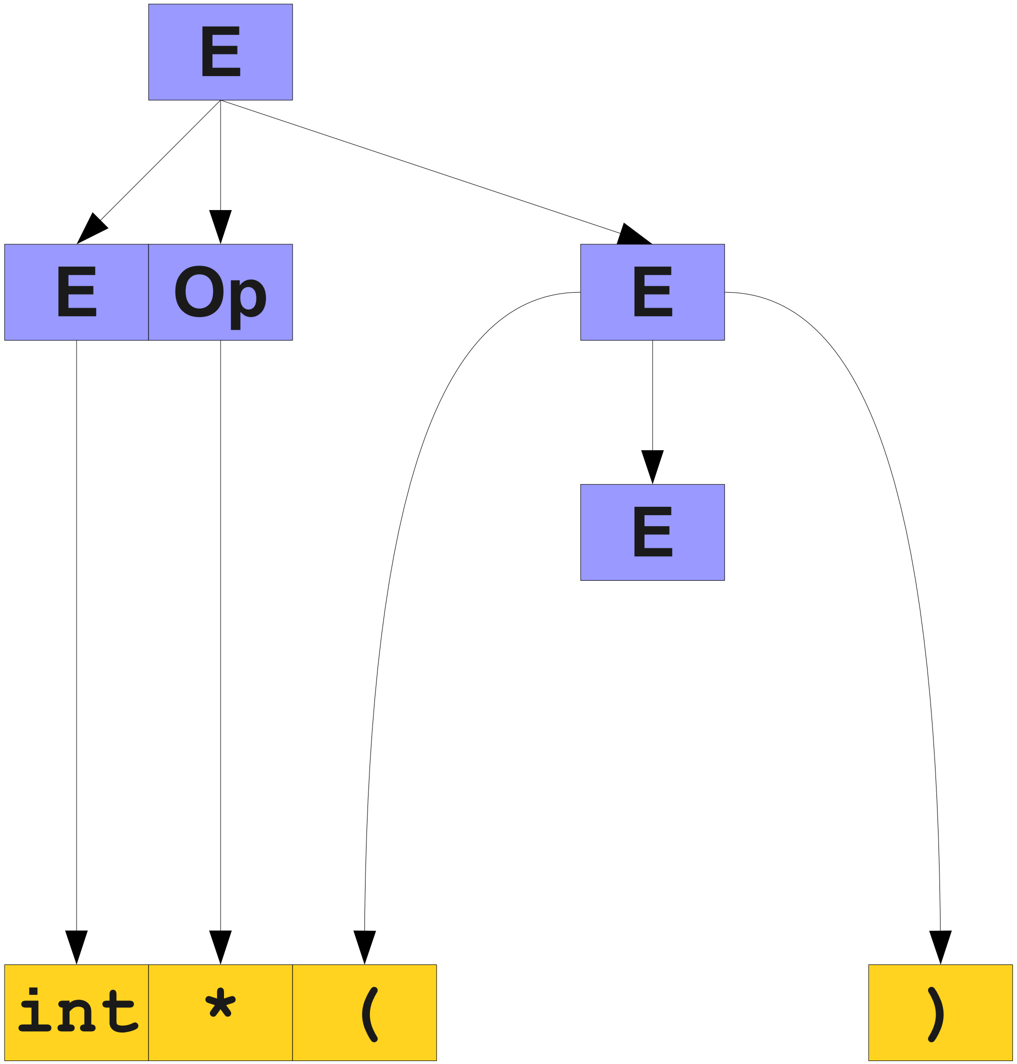
⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int * ` **E**

⇒ `int * (`**E**`)`

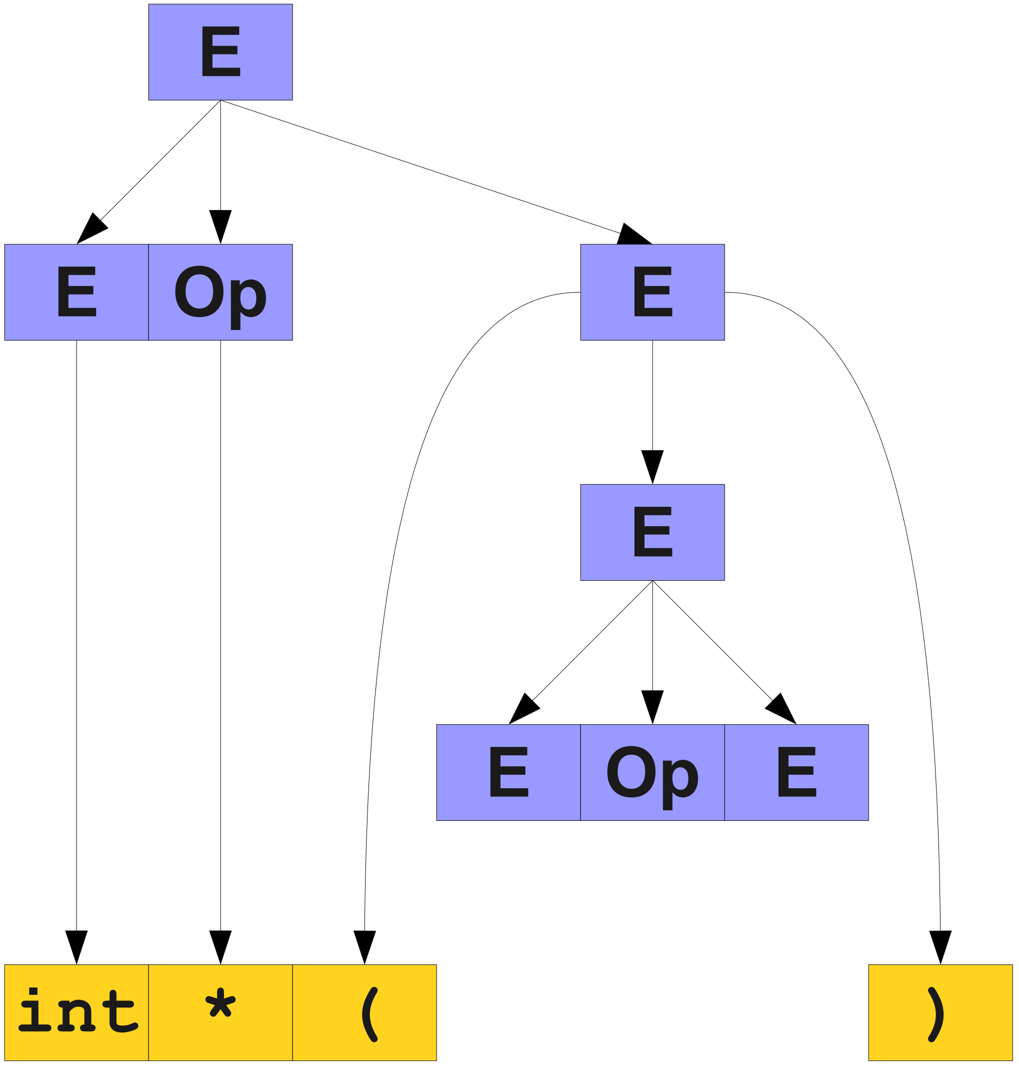
⇒ `int * (`**E Op E**`)`



E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\; & \mathbf{E\ Op\ E} \\
\Rightarrow\; & \texttt{int}\ \mathbf{Op\ E} \\
\Rightarrow\; & \texttt{int * }\mathbf{E} \\
\Rightarrow\; & \texttt{int * (}\mathbf{E}\texttt{)} \\
\Rightarrow\; & \texttt{int * (}\mathbf{E\ Op\ E}\texttt{)}
\end{aligned}
$$



$\mathbf{E} \to \mathbf{E\ Op\ E} \mid \texttt{int} \mid (\mathbf{E})$

$\mathbf{Op} \to \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/}$

Parse Trees

E

⇒ E Op E

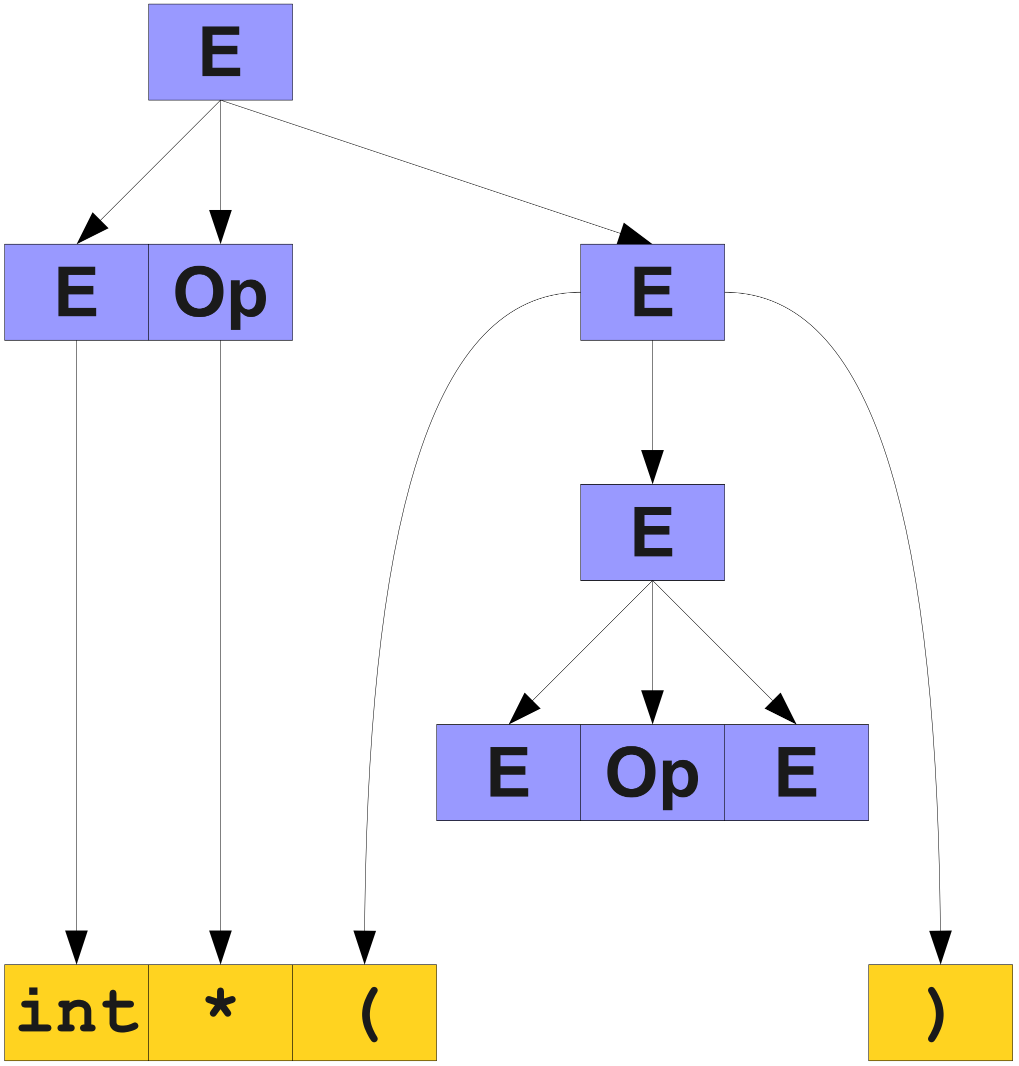
⇒ `int` Op E

⇒ `int * ` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`



E → E Op E | `int` | (E)
Op → `+` | `*` | `-` | `/`

Parse Trees

E

⇒ E Op E

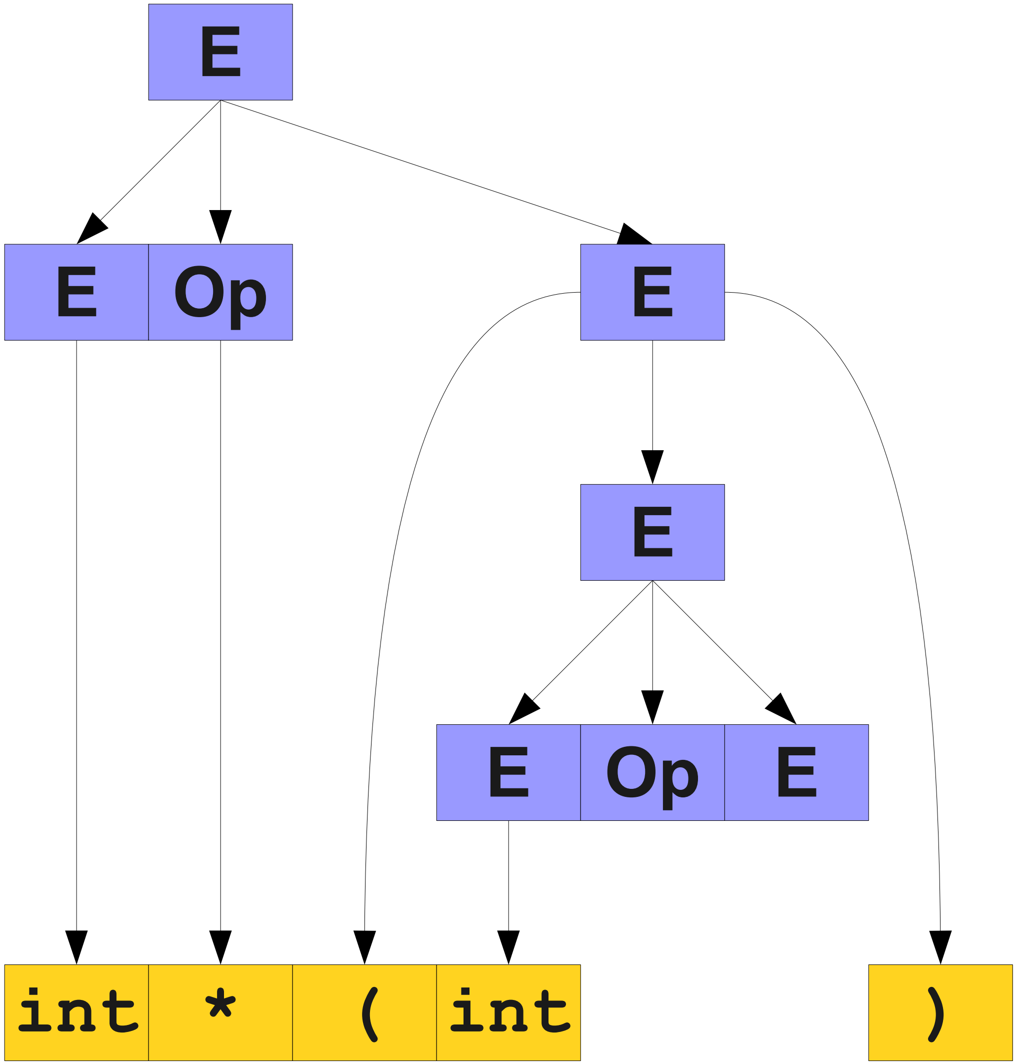
⇒ `int` Op E

⇒ `int * ` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

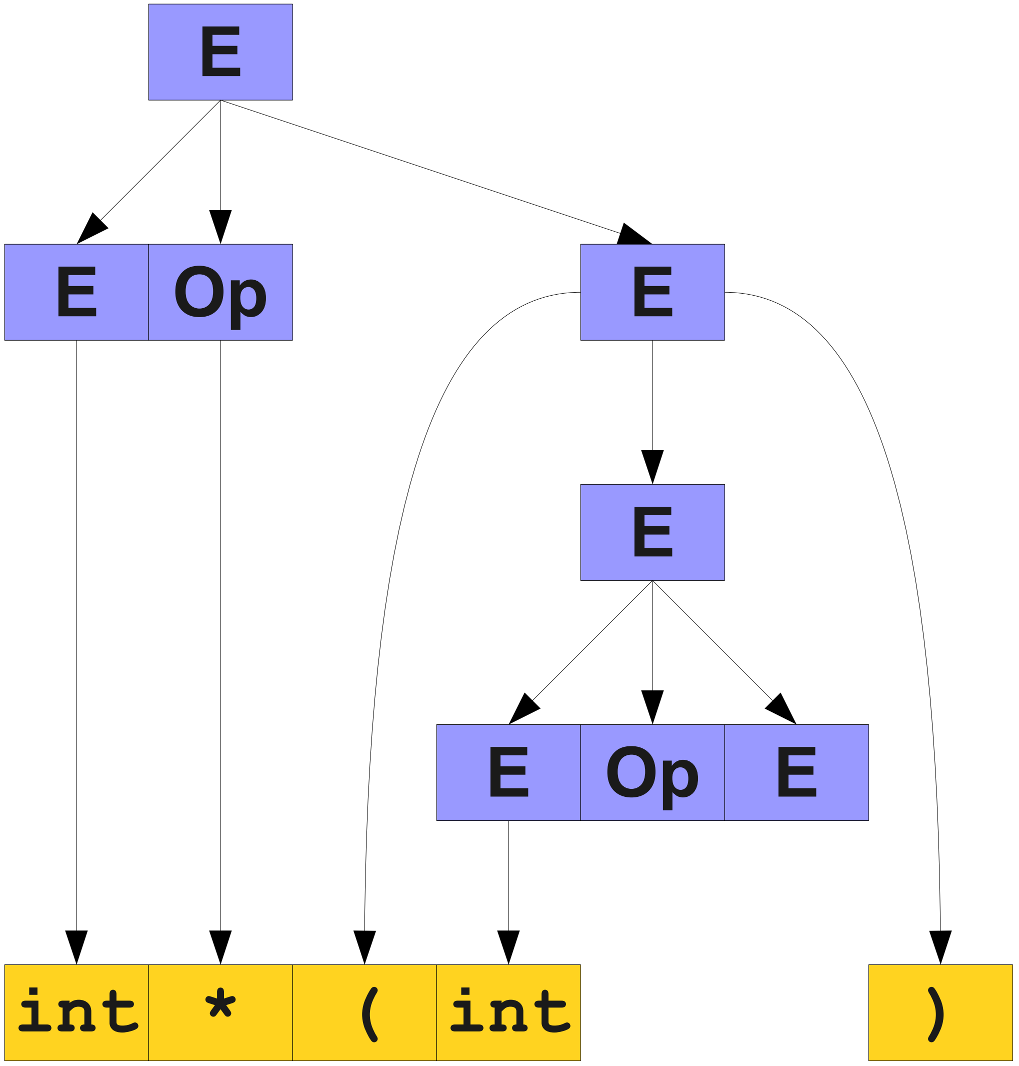
⇒ `int * (int` Op E`)`



E → E Op E | `int` | (E)
Op → `+` | `*` | `-` | `/`

Parse Trees

$$\begin{aligned}
& E \\
\Rightarrow\ & E\ Op\ E \\
\Rightarrow\ & \texttt{int}\ Op\ E \\
\Rightarrow\ & \texttt{int * } E \\
\Rightarrow\ & \texttt{int * (}E\texttt{)} \\
\Rightarrow\ & \texttt{int * (}E\ Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (int }Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (int + }E\texttt{)}
\end{aligned}$$



$$\boxed{\begin{array}{c} E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E) \\ Op \rightarrow \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/} \end{array}}$$

Parse Trees

E

⇒ E Op E

⇒ int Op E

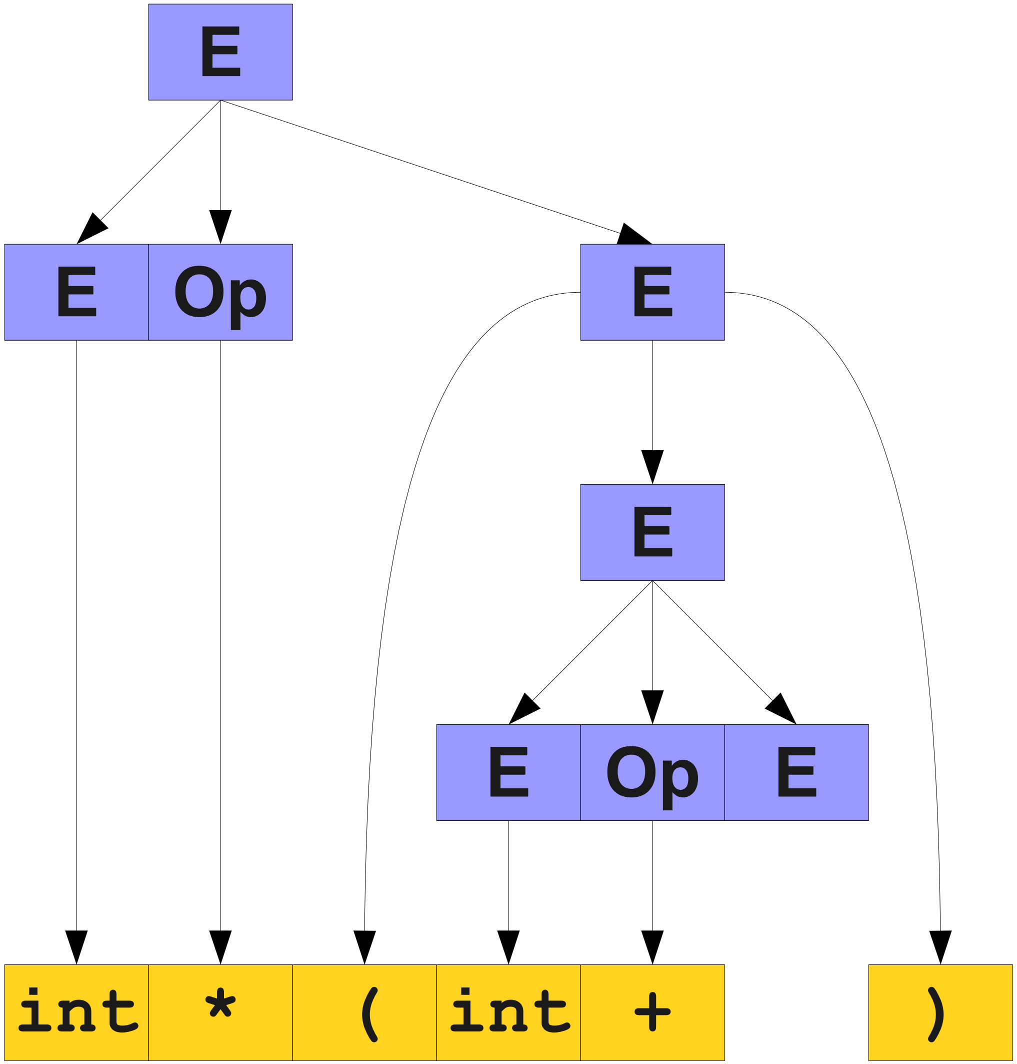
⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

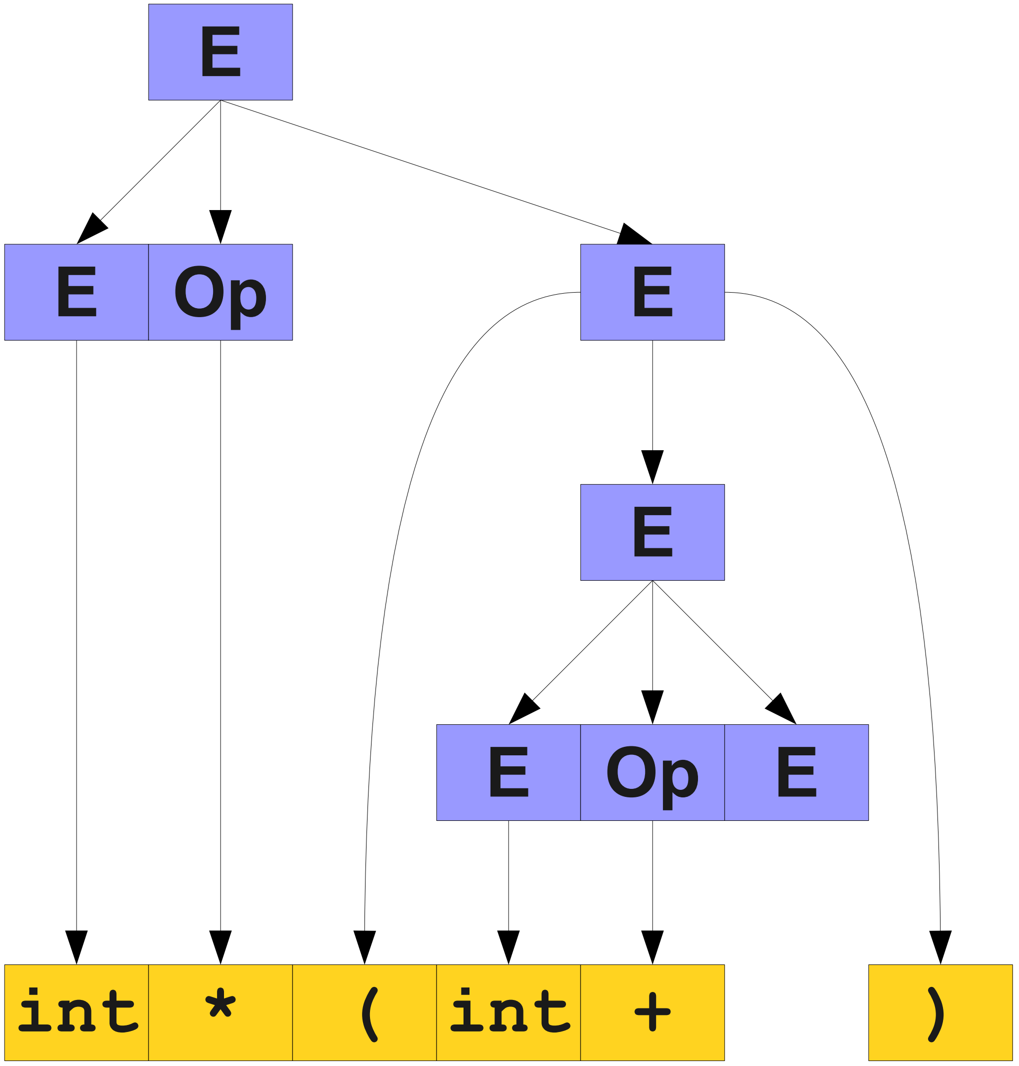
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ `int` Op E

$\Rightarrow$ `int * ` E

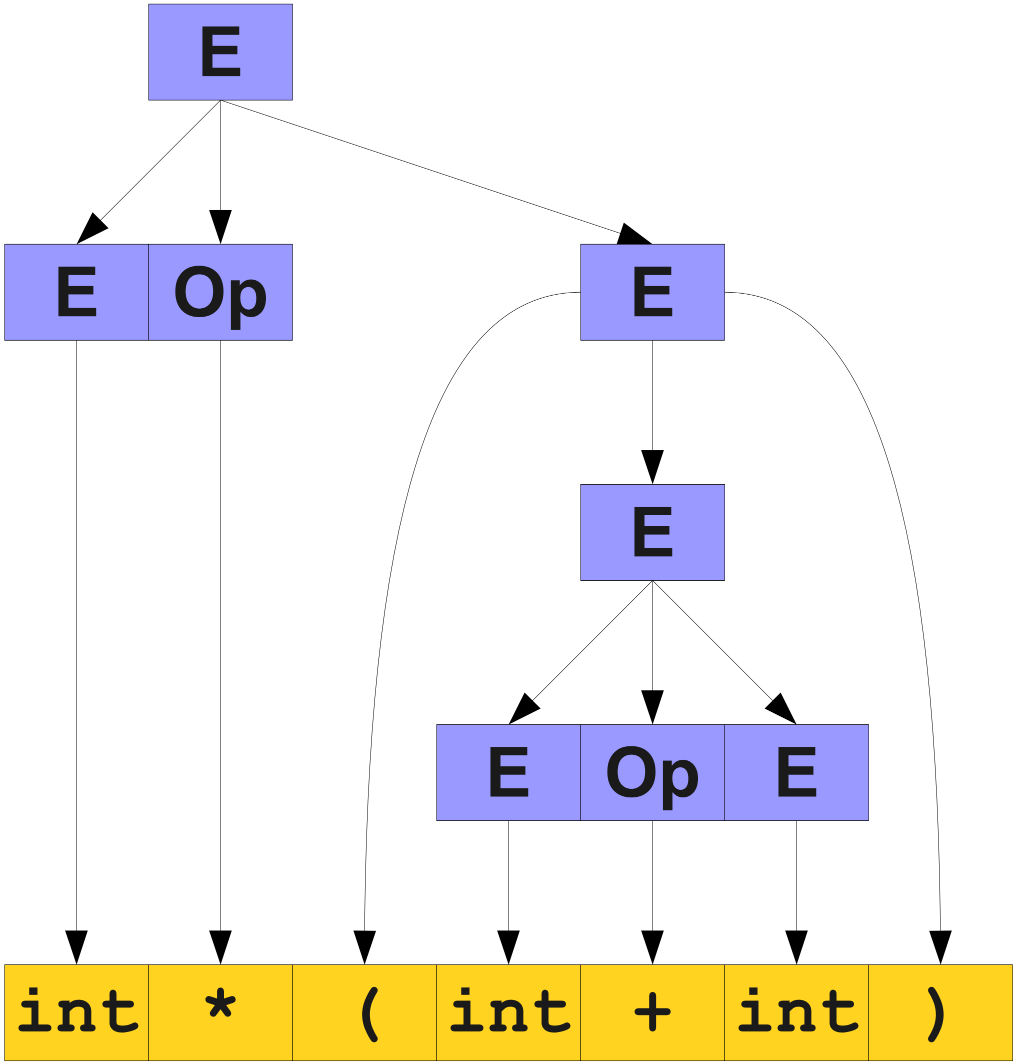
$\Rightarrow$ `int * (`E`)`

$\Rightarrow$ `int * (`E Op E`)`

$\Rightarrow$ `int * (int ` Op E`)`

$\Rightarrow$ `int * (int + `E`)`

$\Rightarrow$ `int * (int + int)`



E → E Op E | `int` | (E)
Op → `+` | `*` | `-` | `/`

Parse Trees

E

E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

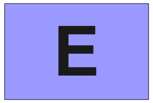
E

E

E → **E Op E** | `int` | (**E**)

Op → `+` | `*` | `-` | `/`

Parse Trees



$\mathbf{E}$

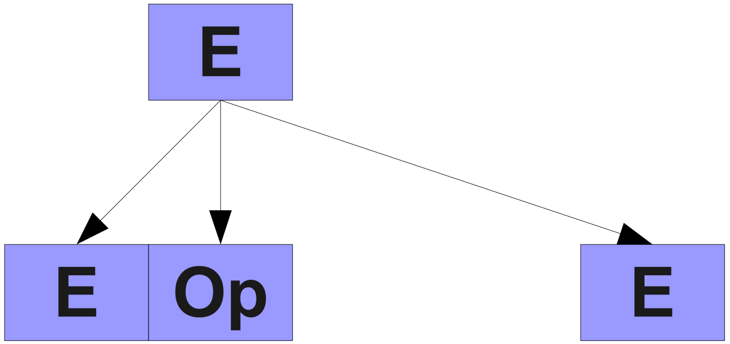
$\Rightarrow \mathbf{E\ Op\ E}$

$\mathbf{E} \rightarrow \mathbf{E\ Op\ E} \mid \texttt{int} \mid (\mathbf{E})$
$\mathbf{Op} \rightarrow \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/}$

Parse Trees

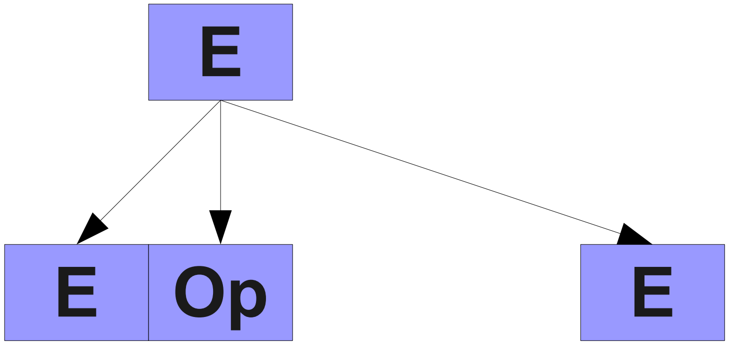
E

⇒ **E Op E**



E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees



$$\mathbf{E}$$
$$\Rightarrow \mathbf{E\ Op\ E}$$
$$\Rightarrow \mathbf{E\ Op\ (E)}$$

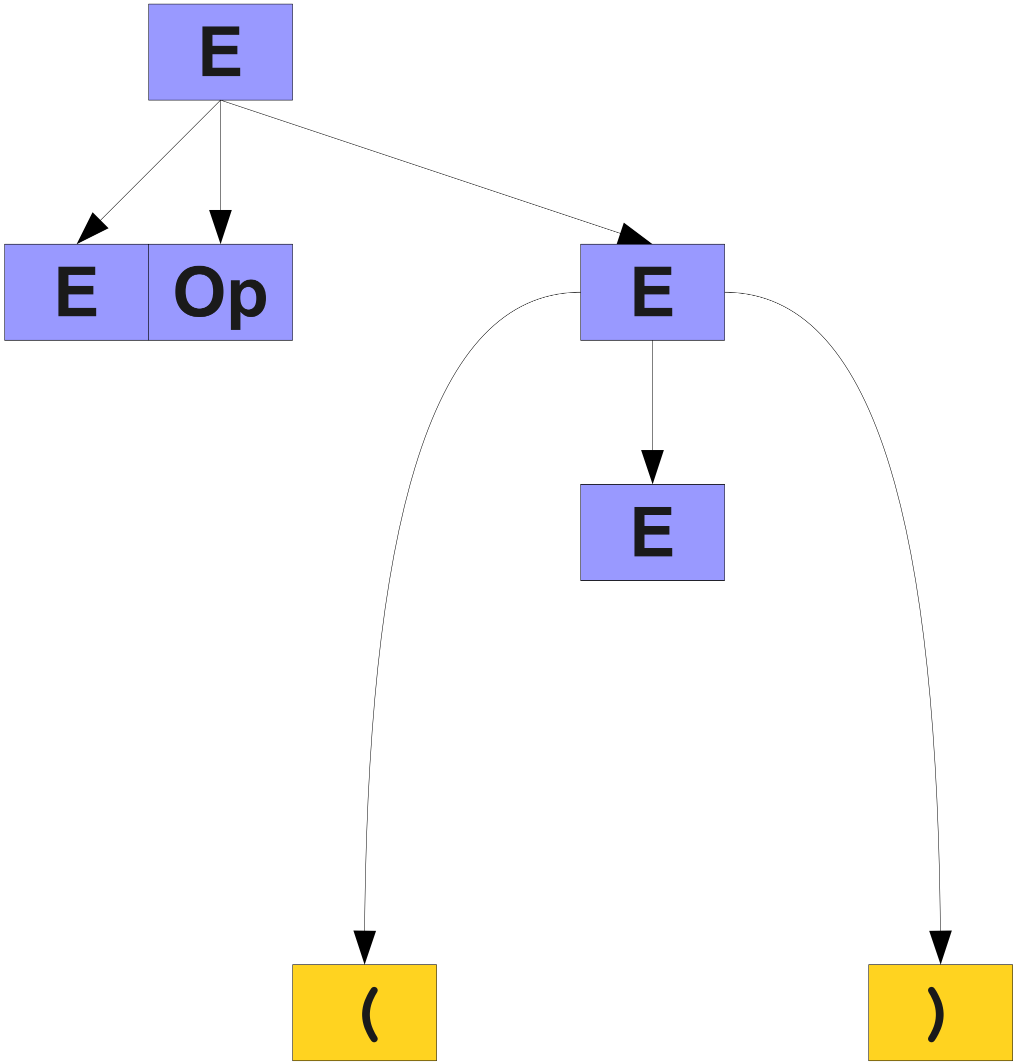
E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

E

⇒ E Op E

⇒ E Op (E)



E → E Op E | `int` | (E)

Op → `+` | `*` | `-` | `/`

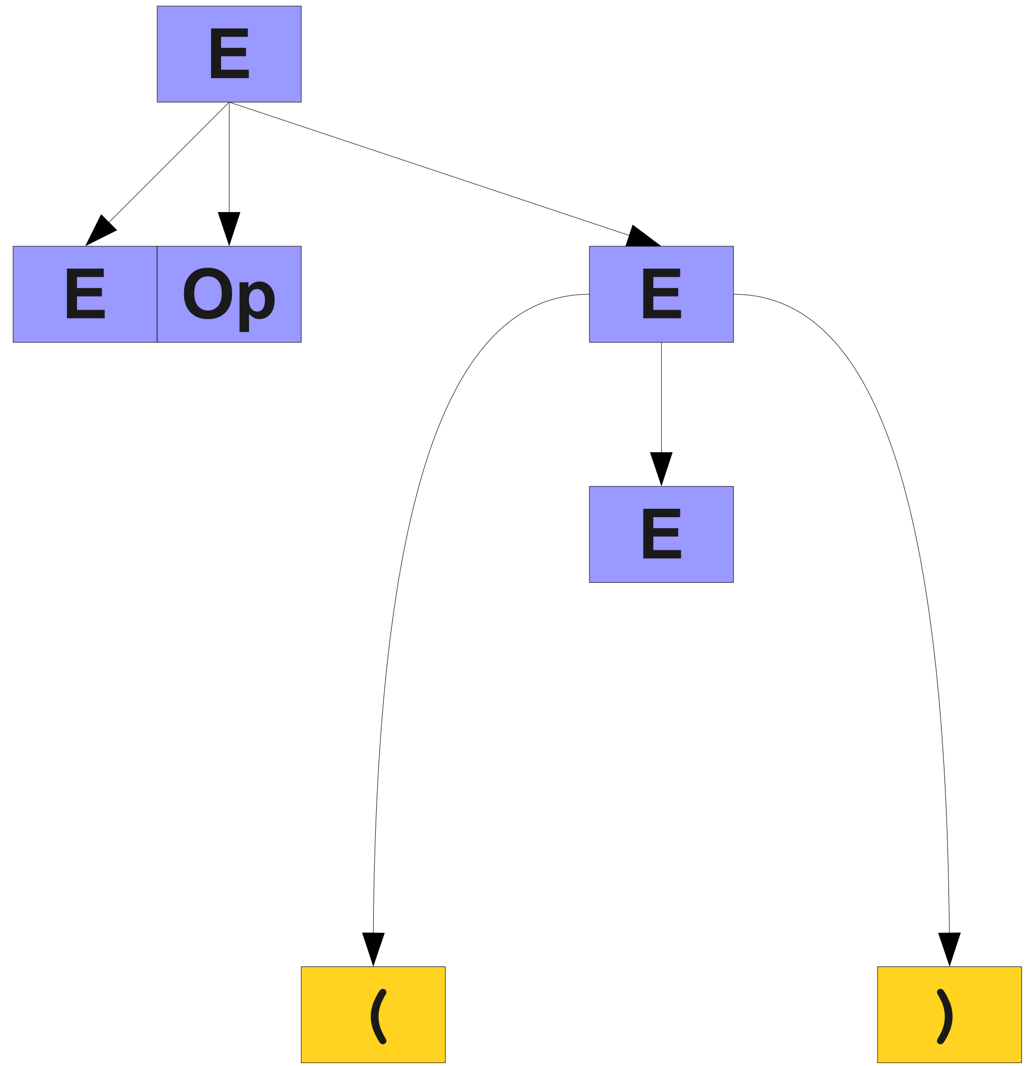
Parse Trees

$\mathbf{E}$

$\Rightarrow$ **E Op E**

$\Rightarrow$ **E Op (E)**

$\Rightarrow$ **E Op (E Op E)**



E → **E Op E** | `int` | (**E**)

Op → `+` | `*` | `-` | `/`

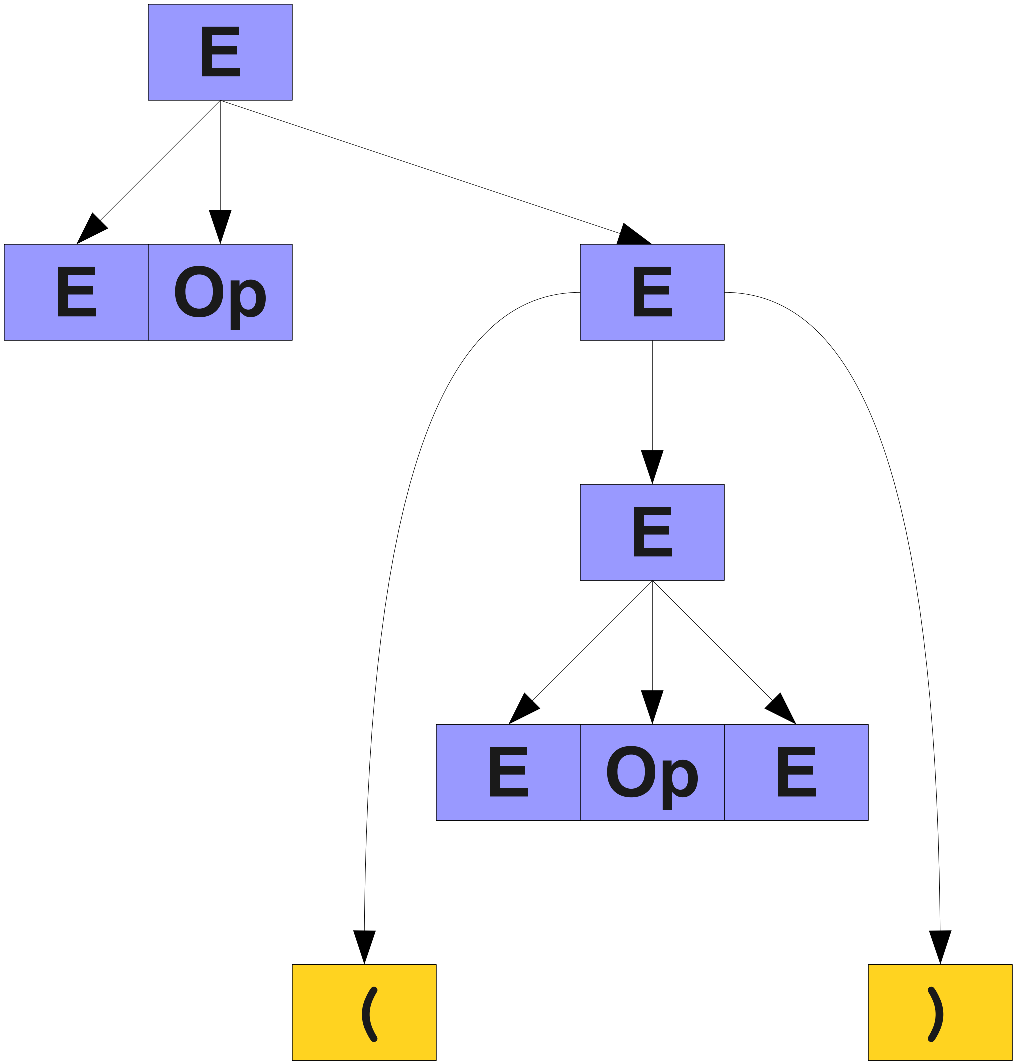
Parse Trees

E

$\Rightarrow$ **E Op E**

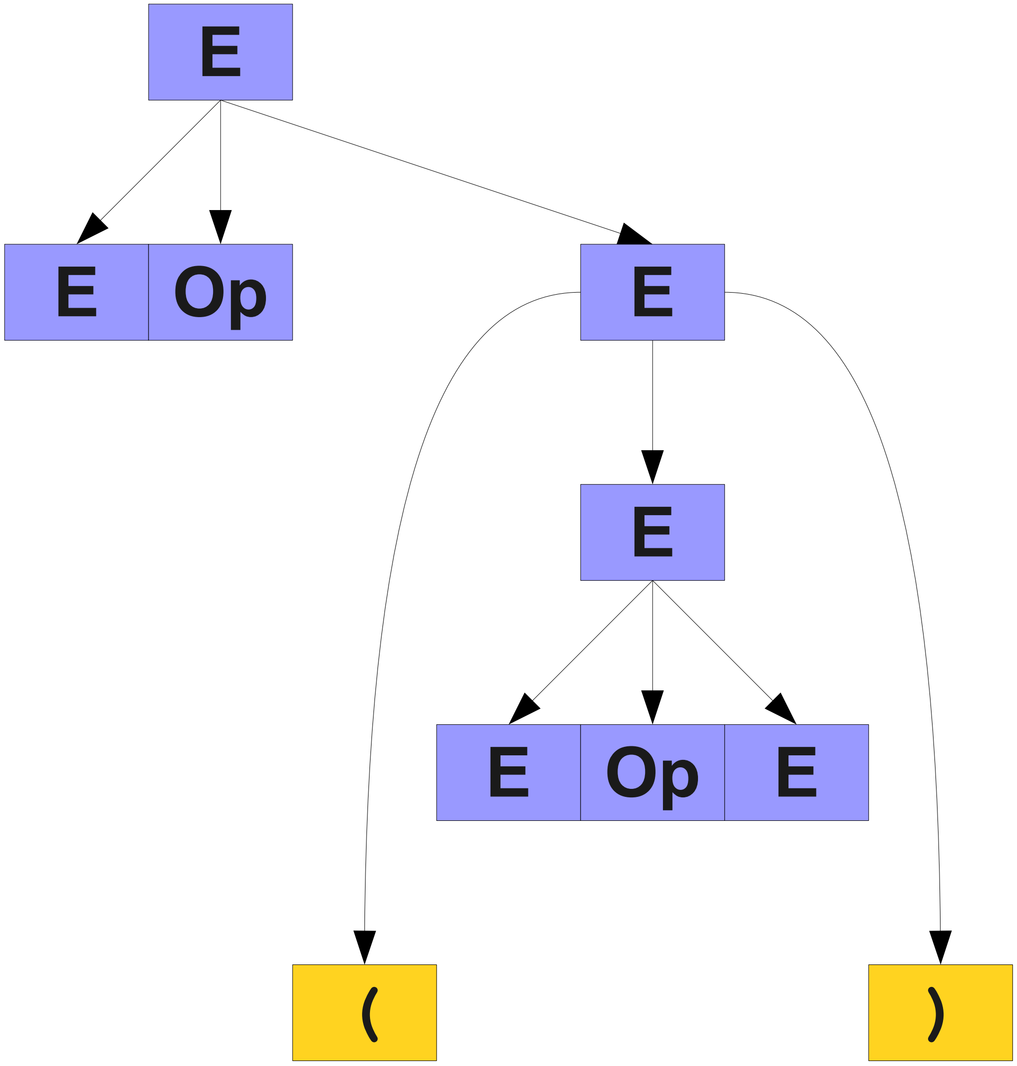
$\Rightarrow$ **E Op (E)**

$\Rightarrow$ **E Op (E Op E)**



E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees



$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \mathbf{E\ Op}\ \texttt{(}\mathbf{E}\texttt{)} \\
\Rightarrow\ & \mathbf{E\ Op}\ \texttt{(}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow\ & \mathbf{E\ Op}\ \texttt{(}\mathbf{E\ Op}\ \texttt{int)}
\end{aligned}
$$

$\mathbf{E} \to \mathbf{E\ Op\ E} \mid \texttt{int} \mid (\mathbf{E})$

$\mathbf{Op} \to \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/}$

Parse Trees

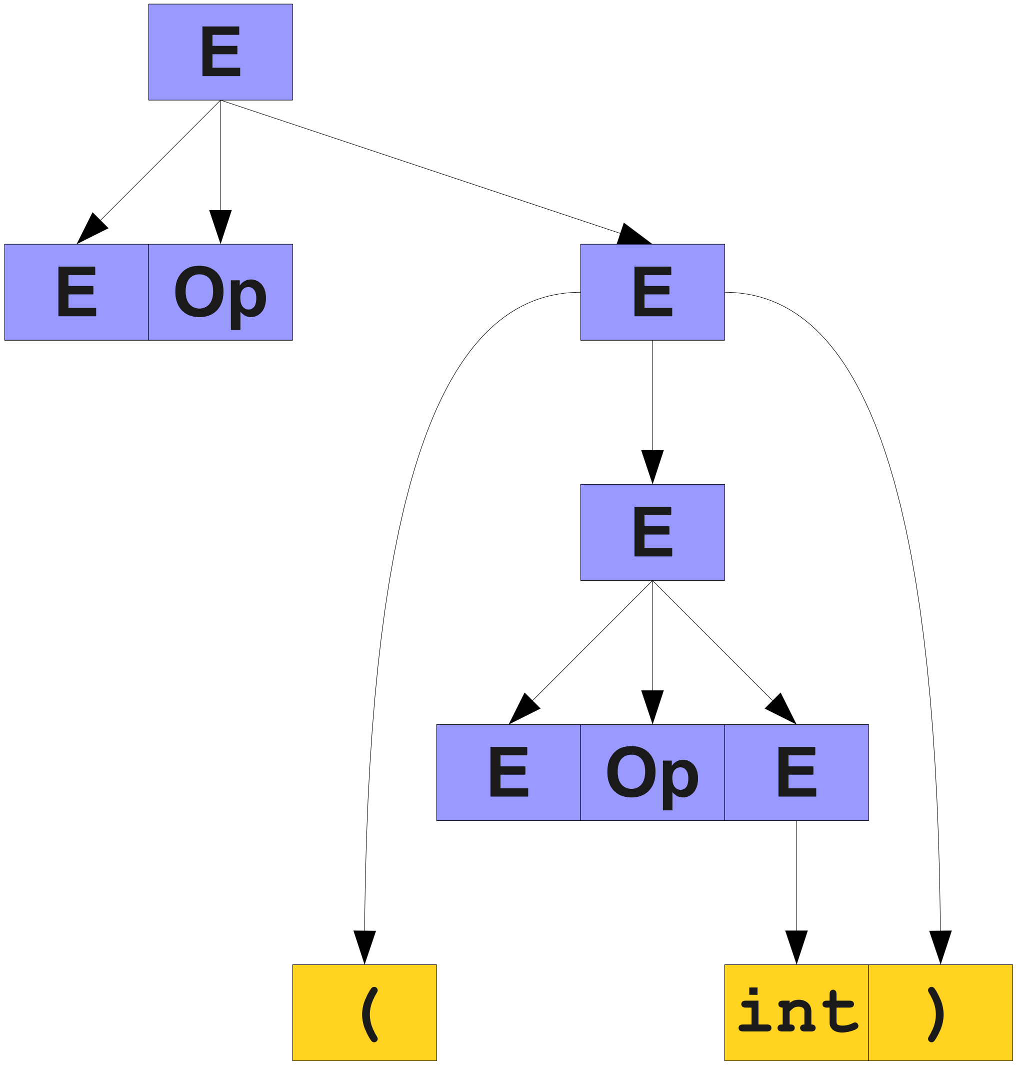
E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op `int`)



$\mathbf{E} \rightarrow \mathbf{E\ Op\ E} \mid \texttt{int} \mid (\mathbf{E})$

$\mathbf{Op} \rightarrow \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/}$

Parse Trees

E

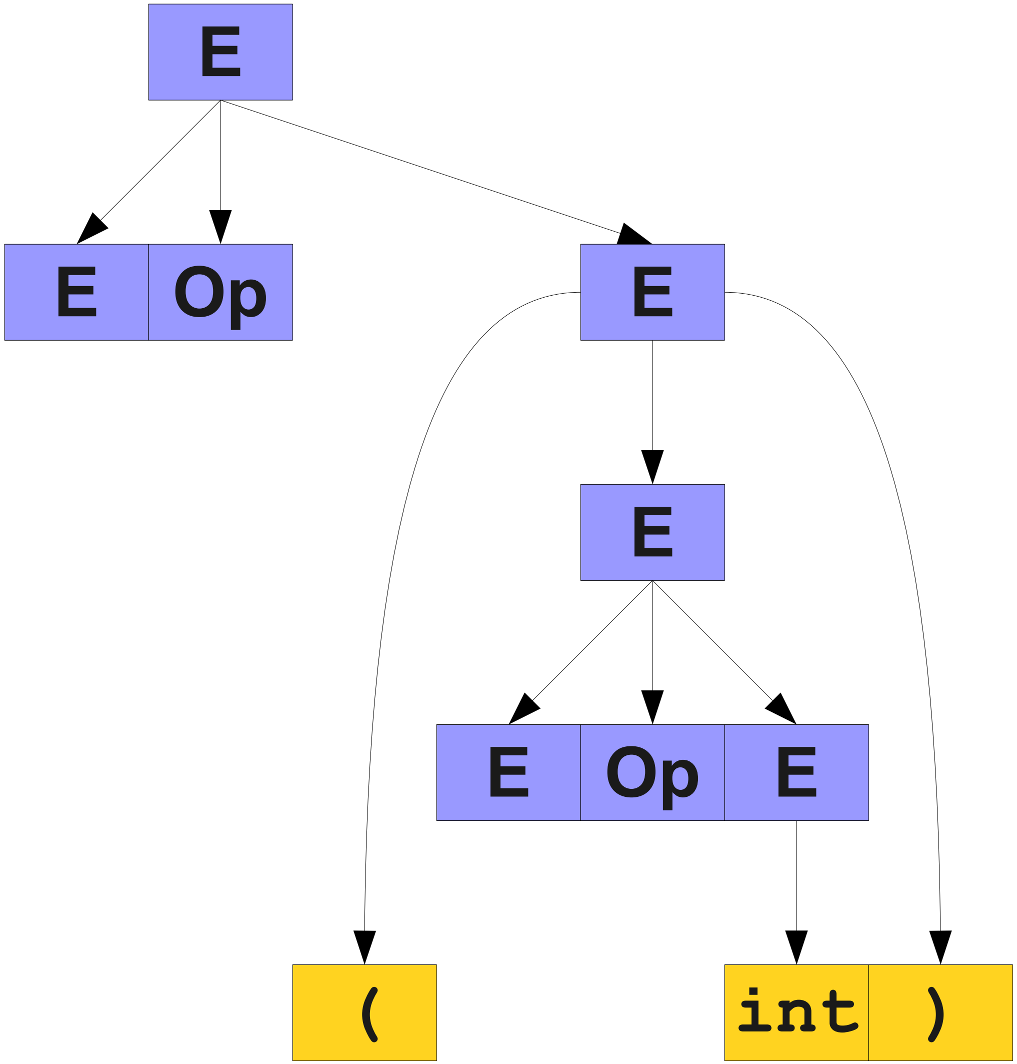
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

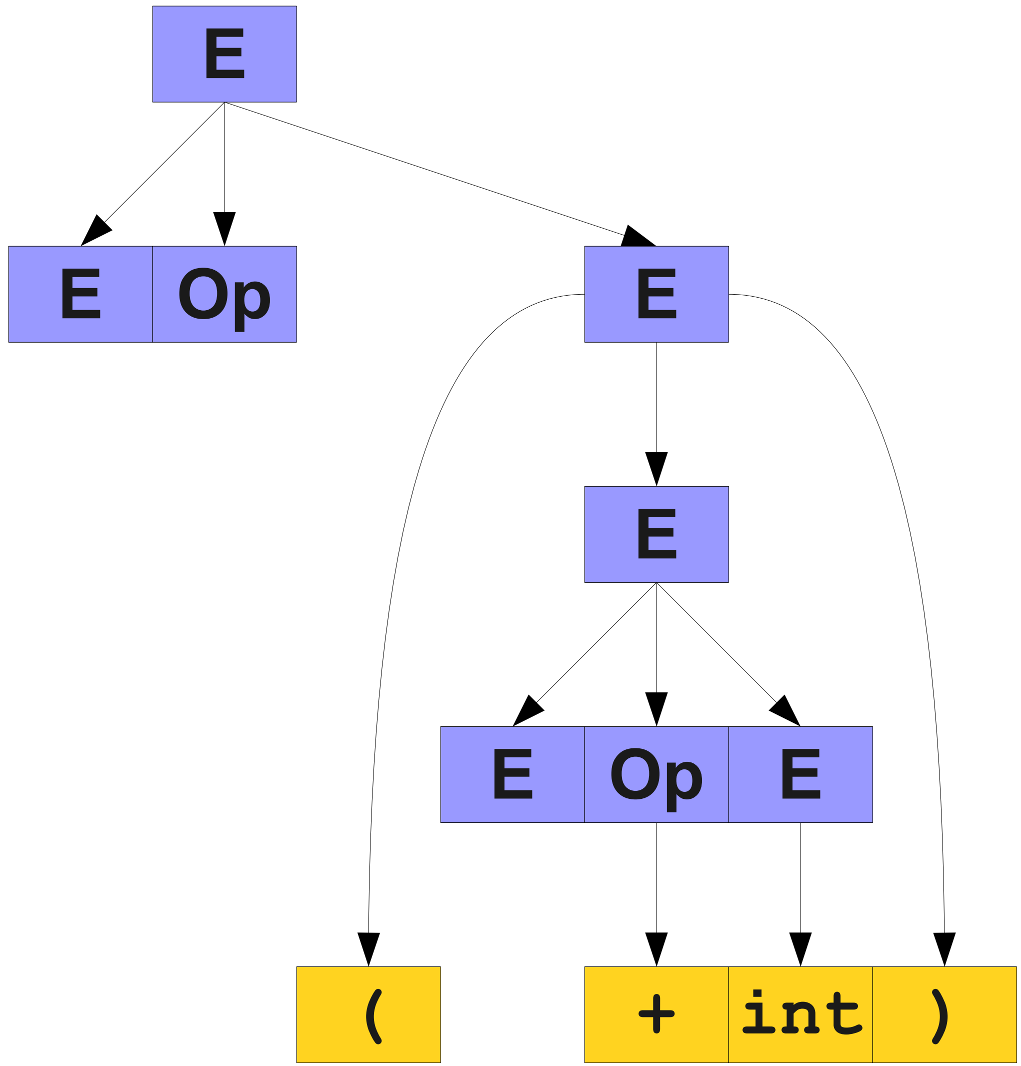
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

⇒ E Op E

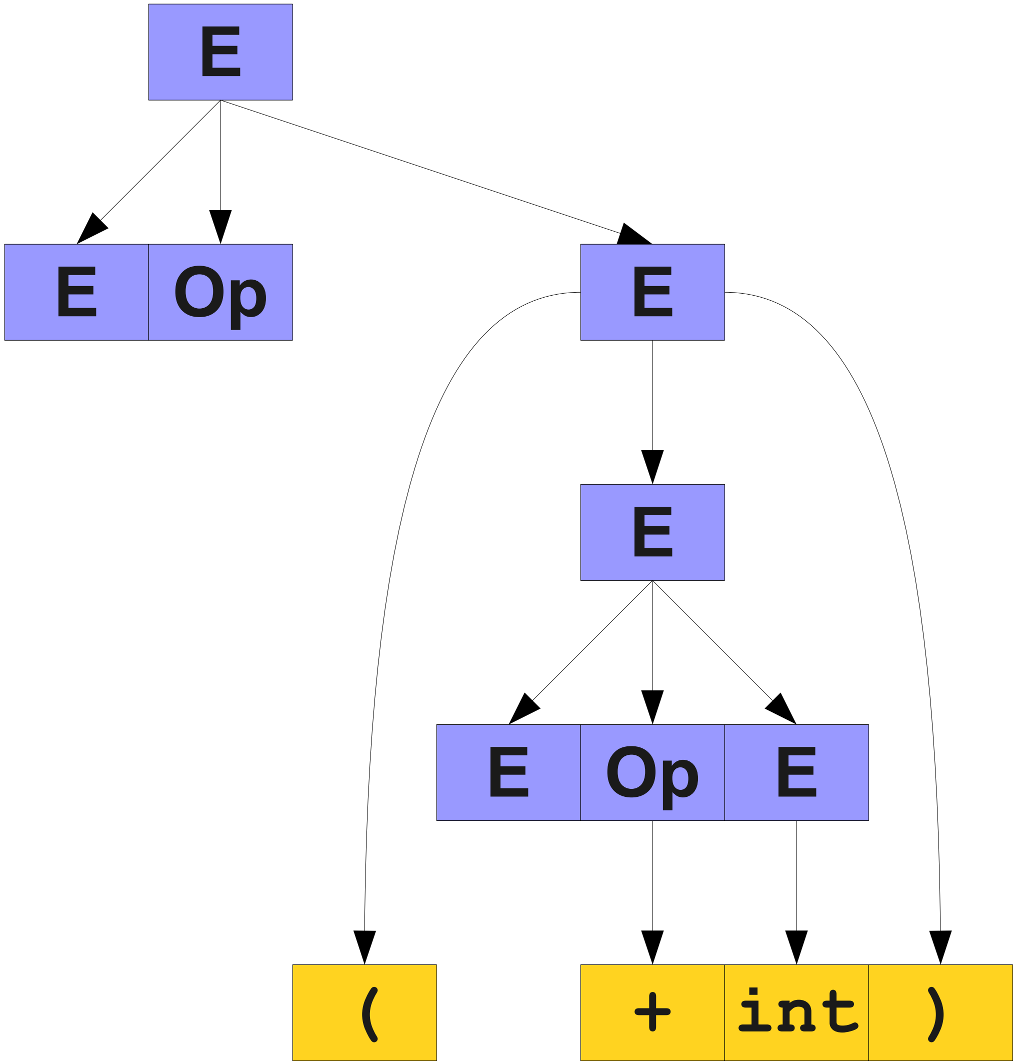
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

⇒ E Op E

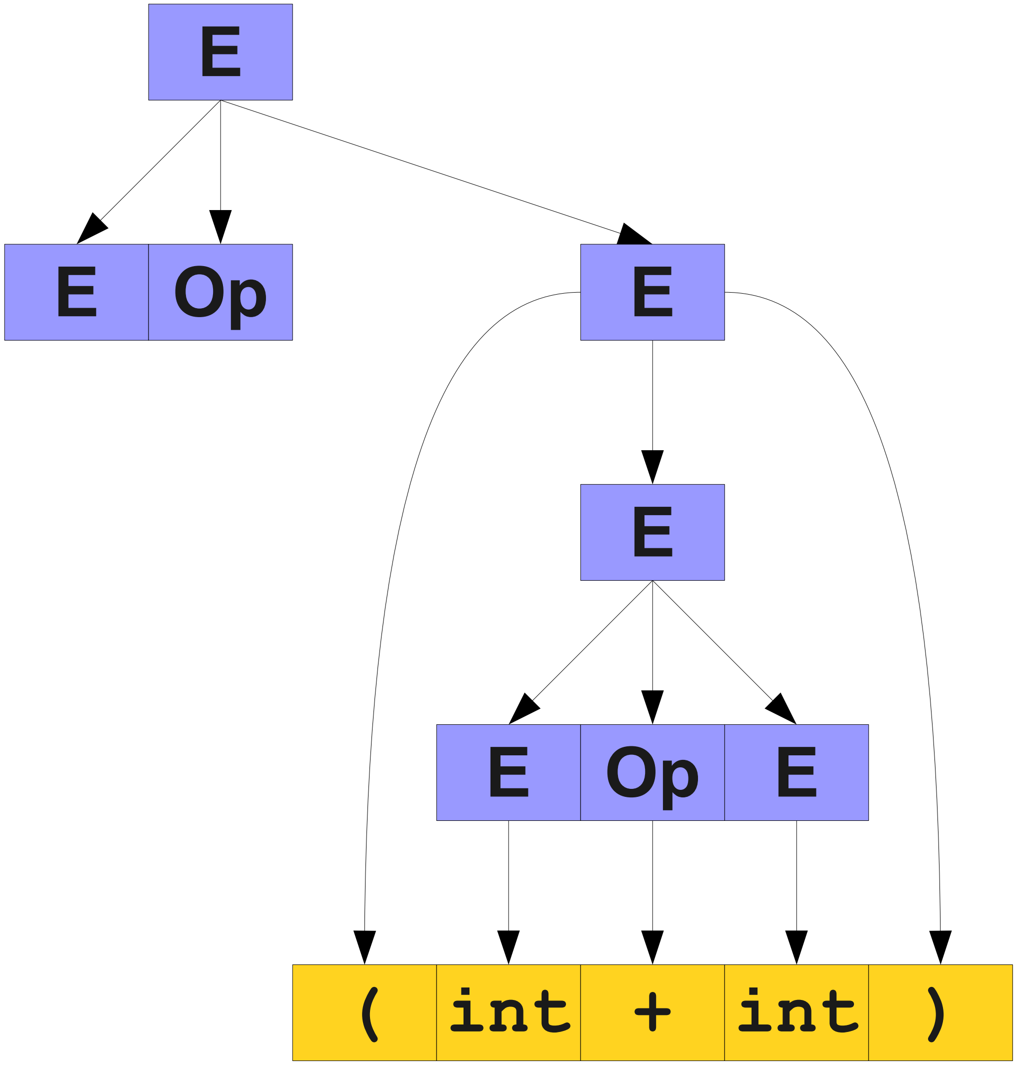
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

$\Rightarrow$ **E Op E**

$\Rightarrow$ **E Op (E)**

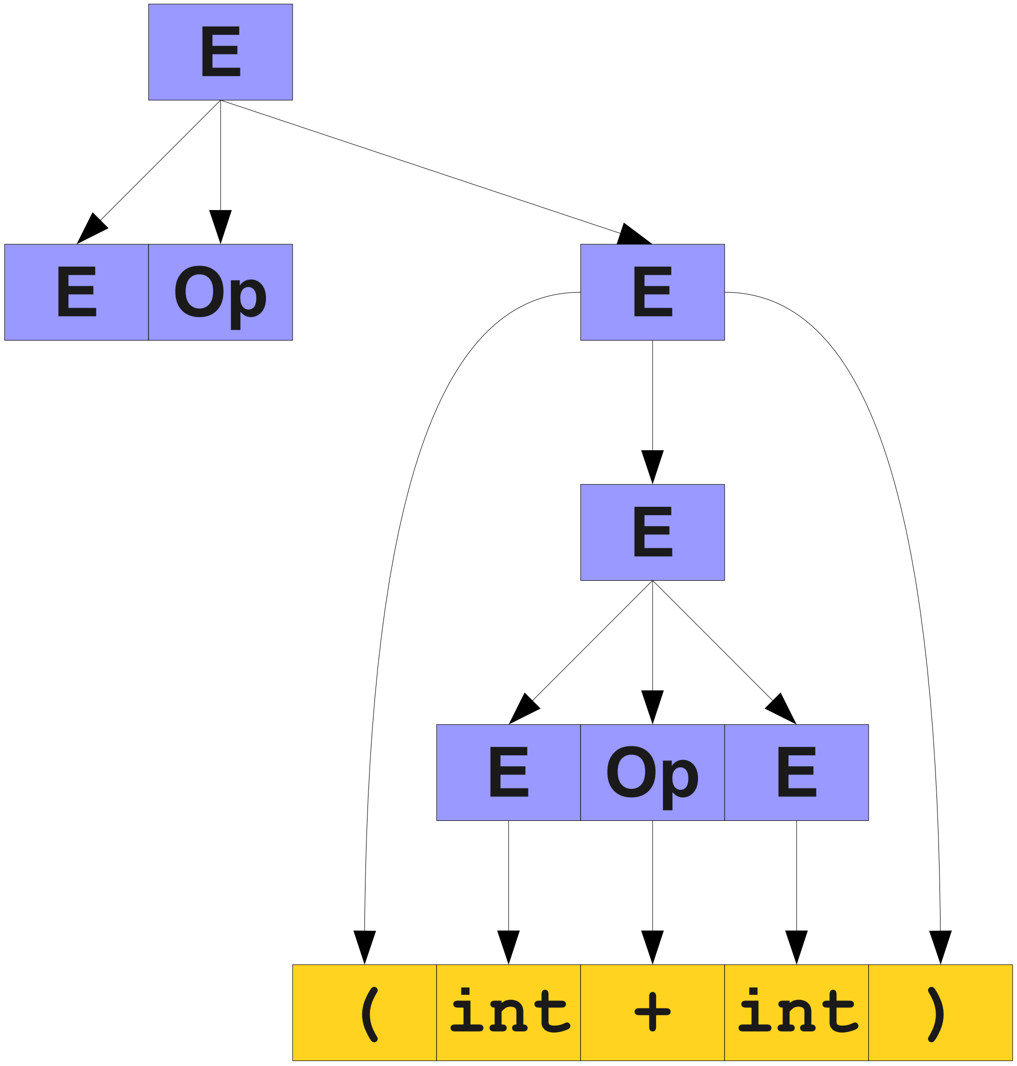
$\Rightarrow$ **E Op (E Op E)**

$\Rightarrow$ **E Op (E Op** `int)`

$\Rightarrow$ **E Op (E** `+ int)`

$\Rightarrow$ **E Op** `(int + int)`

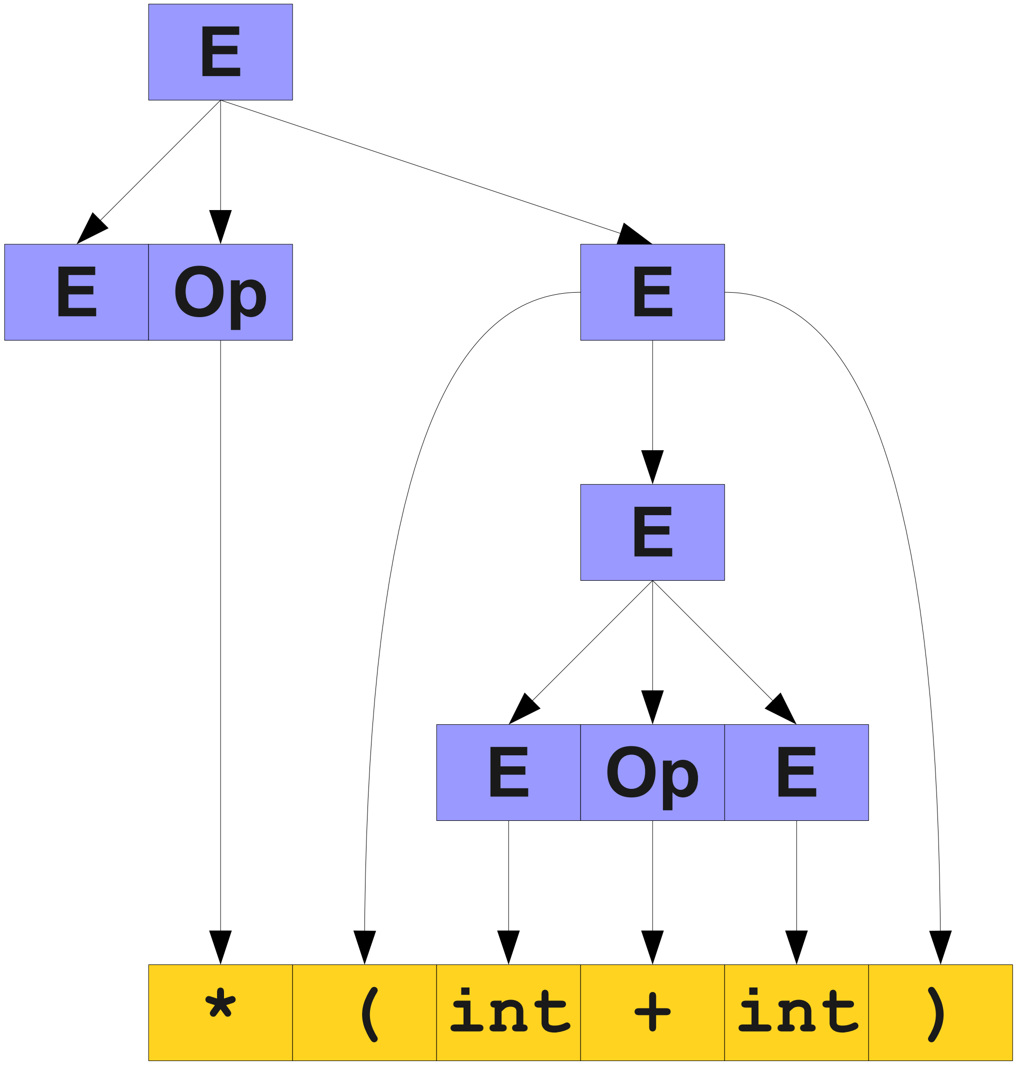
$\Rightarrow$ **E** `* (int + int)`



E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

Parse Trees

E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E Op (E Op int)
⇒ E Op (E + int)
⇒ E Op (int + int)
⇒ E * (int + int)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

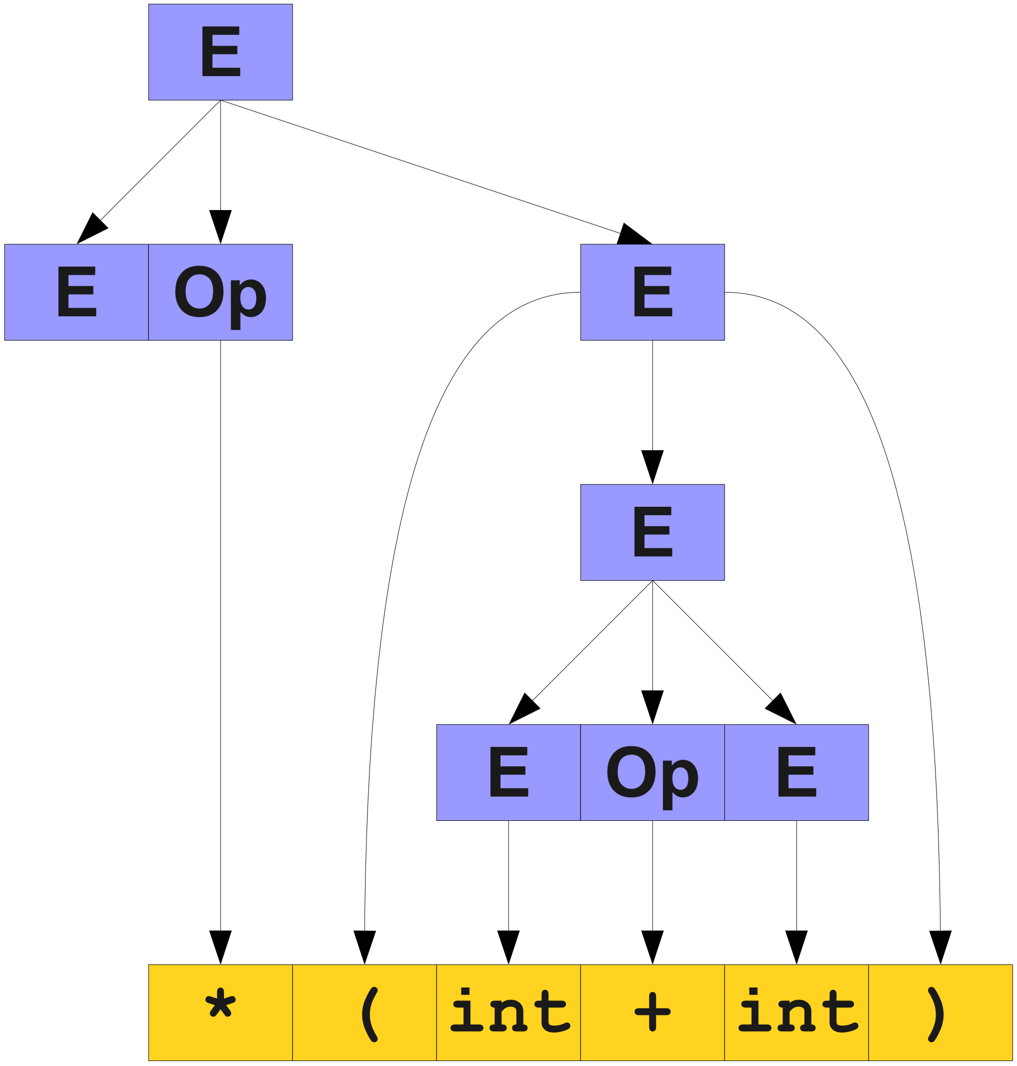
⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)

⇒ int * (int + int)



E → E Op E | int | (E)
Op → + | * | - | /

Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

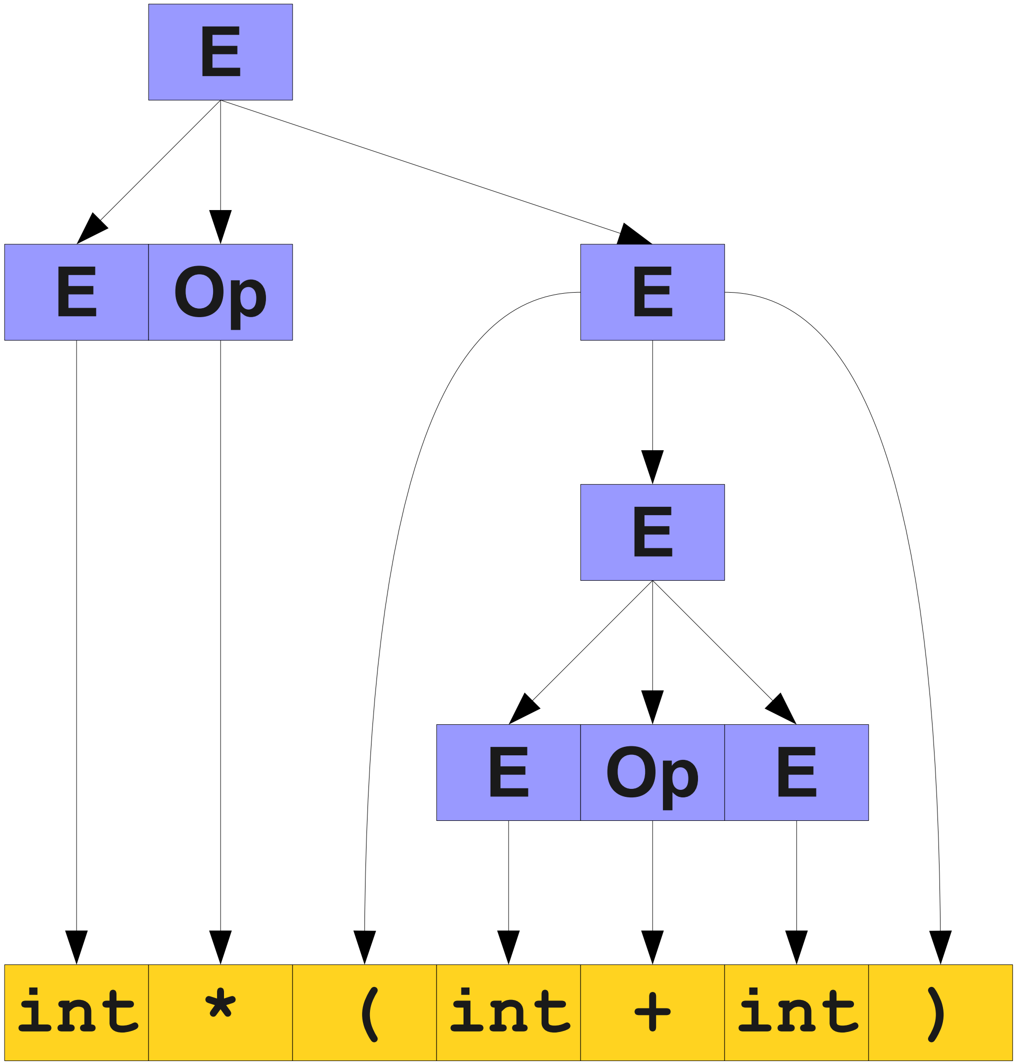
⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

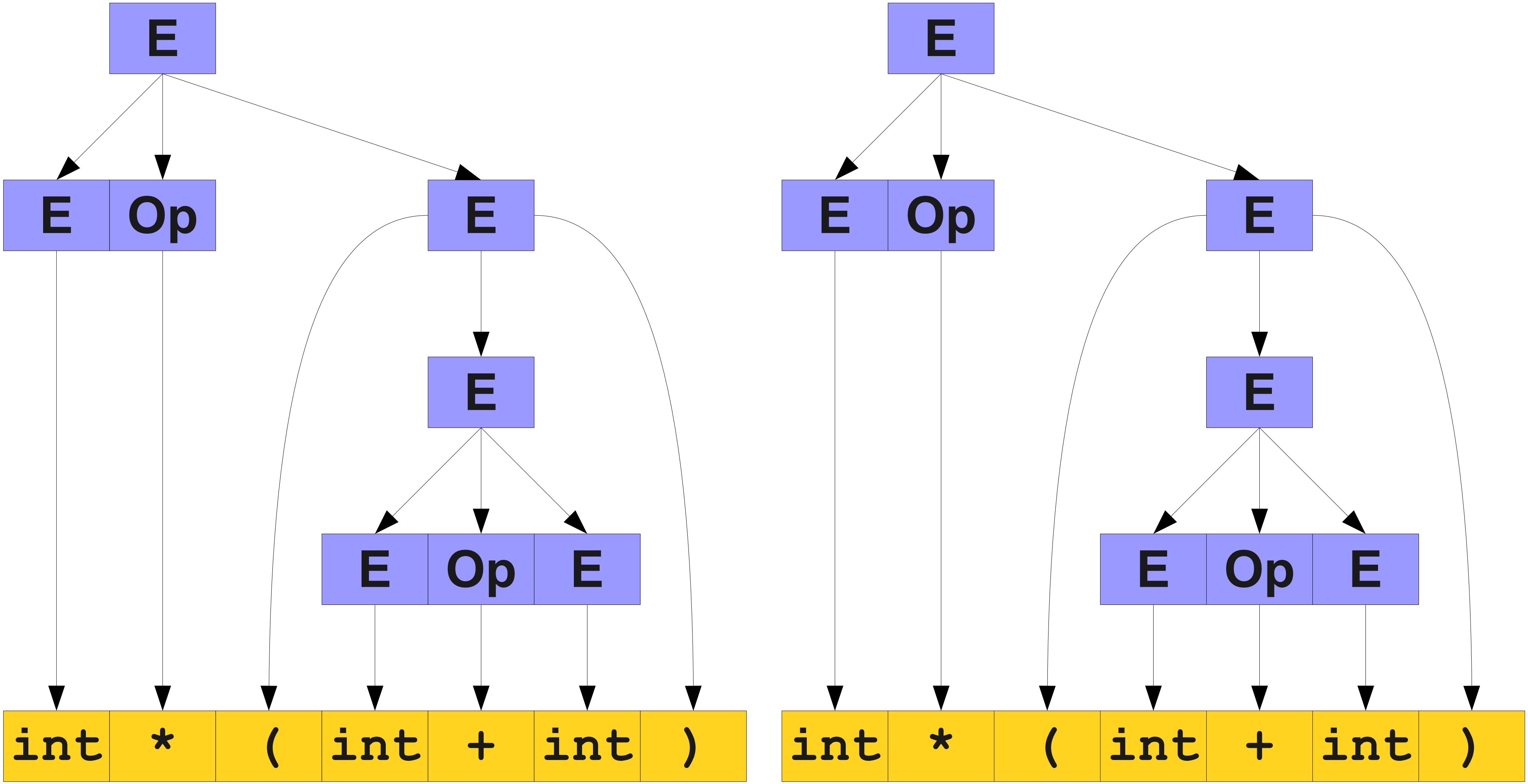
⇒ E * (int + int)

⇒ int * (int + int)



E → E Op E | int | (E)
Op → + | * | - | /

For Comparison



Parse Trees

- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Walking the leaves in order gives the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

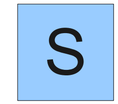
Parse Trees Revisited

S → `[`**P**`]`

P → **RR** | `a`

R → `(`**P**`)` | `b`

Parse Trees Revisited



S → `[`**P**`]`

P → **RR** | `a`

R → `(`**P**`)` | `b`

Parse Trees Revisited

S → [**P**]

P → **RR** | a

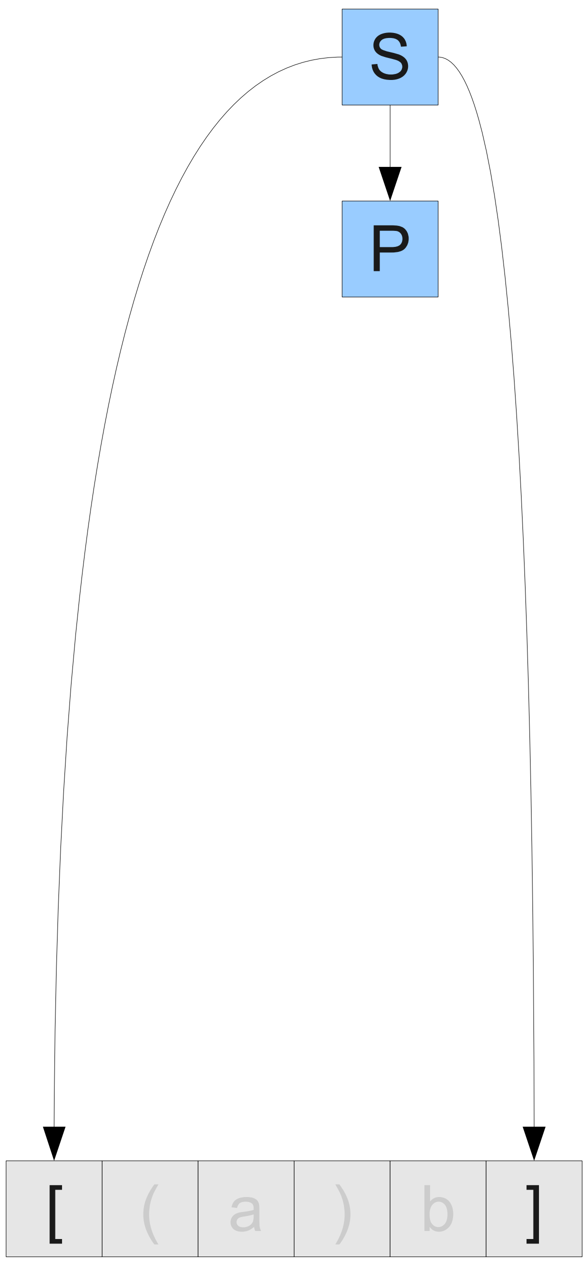
R → (**P**) | b

Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

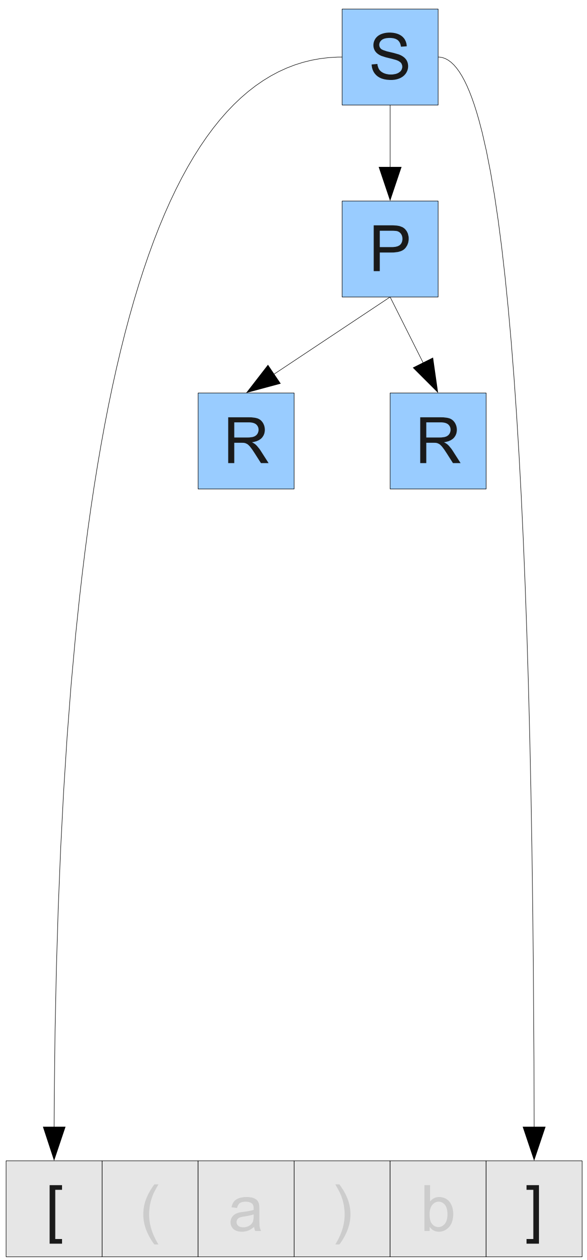


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

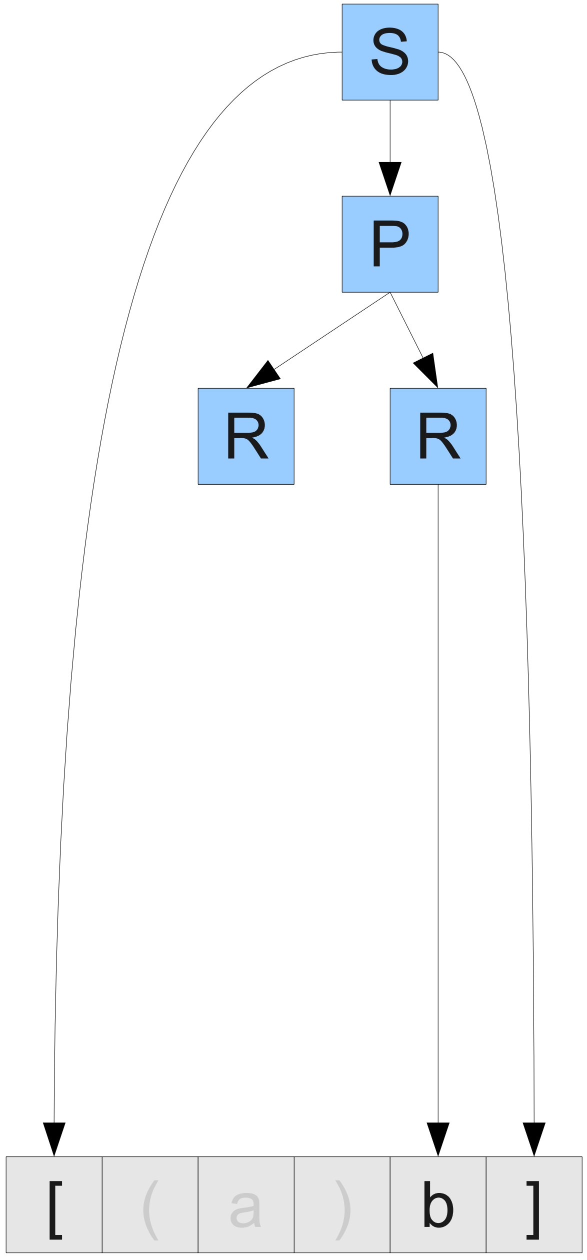


Parse Trees Revisited

S → `[`**P**`]`

P → **RR** | `a`

R → `(`**P**`)` | `b`

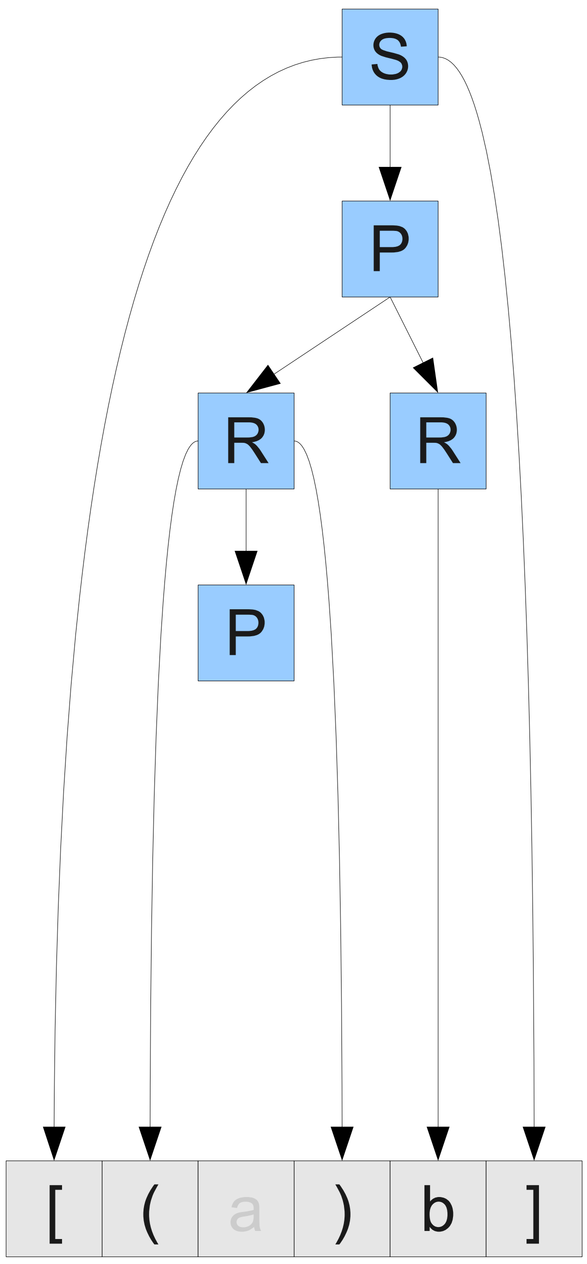


Parse Trees Revisited

S → `[`**P**`]`

P → **RR** | `a`

R → `(`**P**`)` | `b`

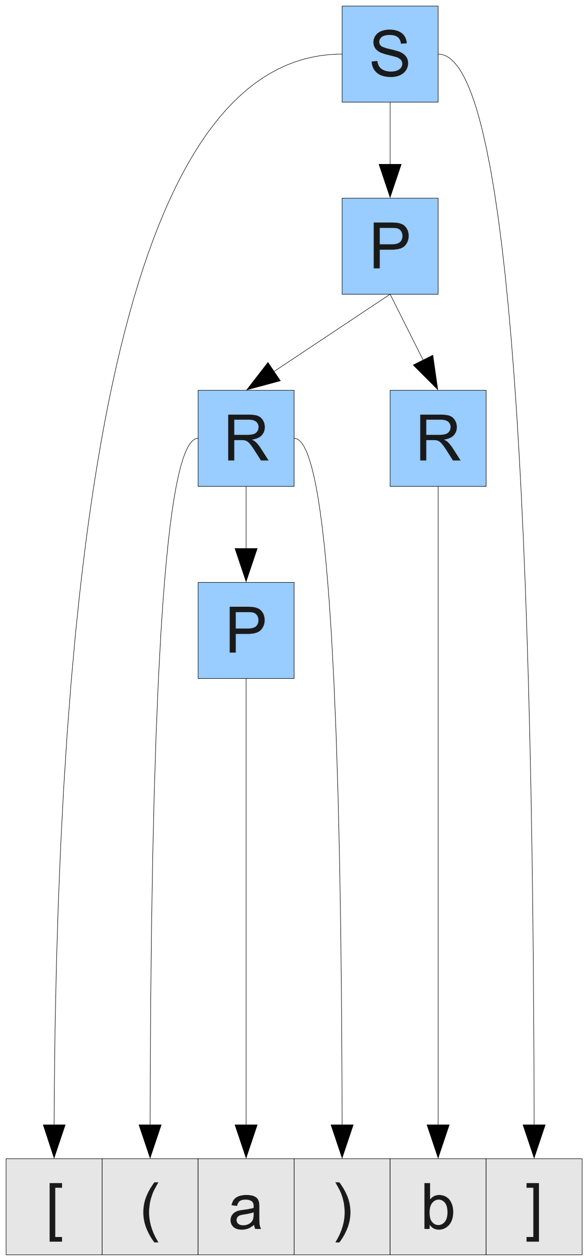


Parse Trees Revisited

$S \rightarrow$ [P]

$P \rightarrow RR$ | a

$R \rightarrow$ (P) | b

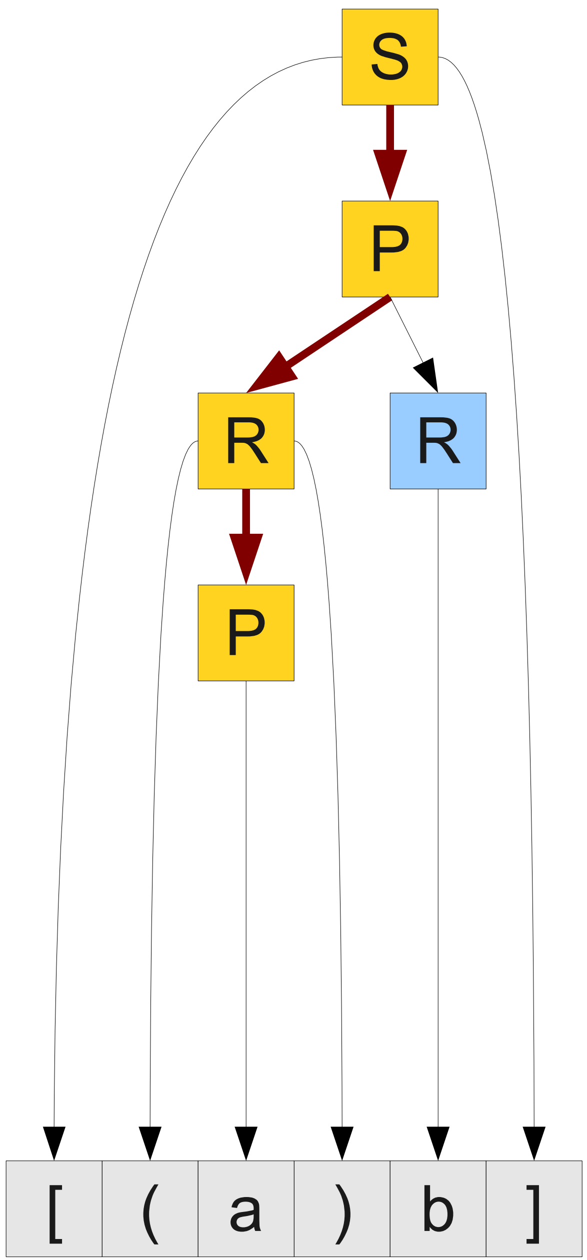


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

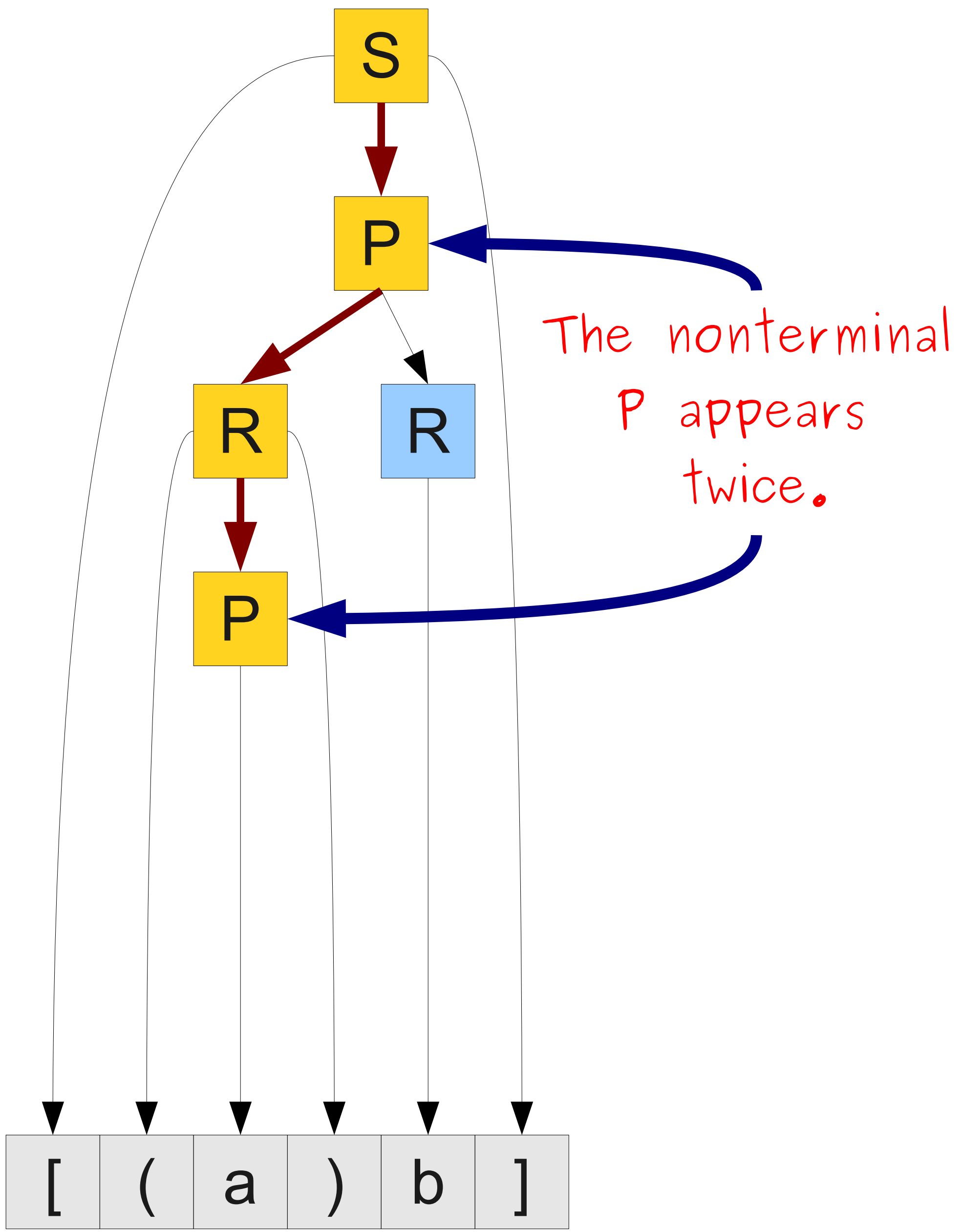


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

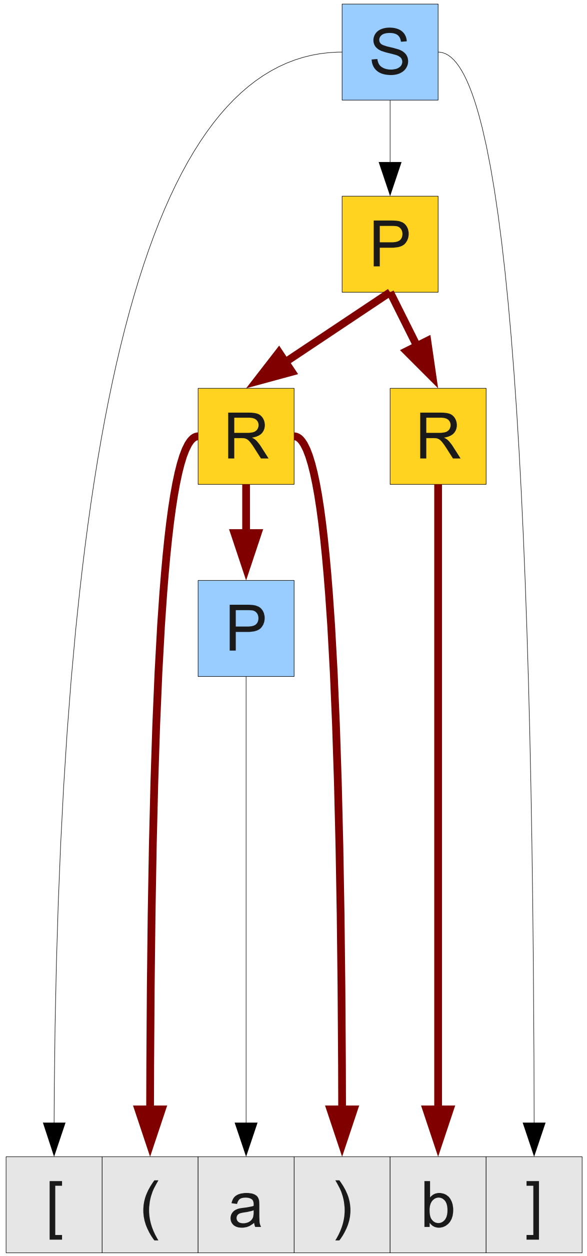


Parse Trees Revisited

$\mathbf{S} \rightarrow$ `[`$\mathbf{P}$`]`

$\mathbf{P} \rightarrow \mathbf{RR}$ | `a`

$\mathbf{R} \rightarrow$ `(`$\mathbf{P}$`)` | `b`

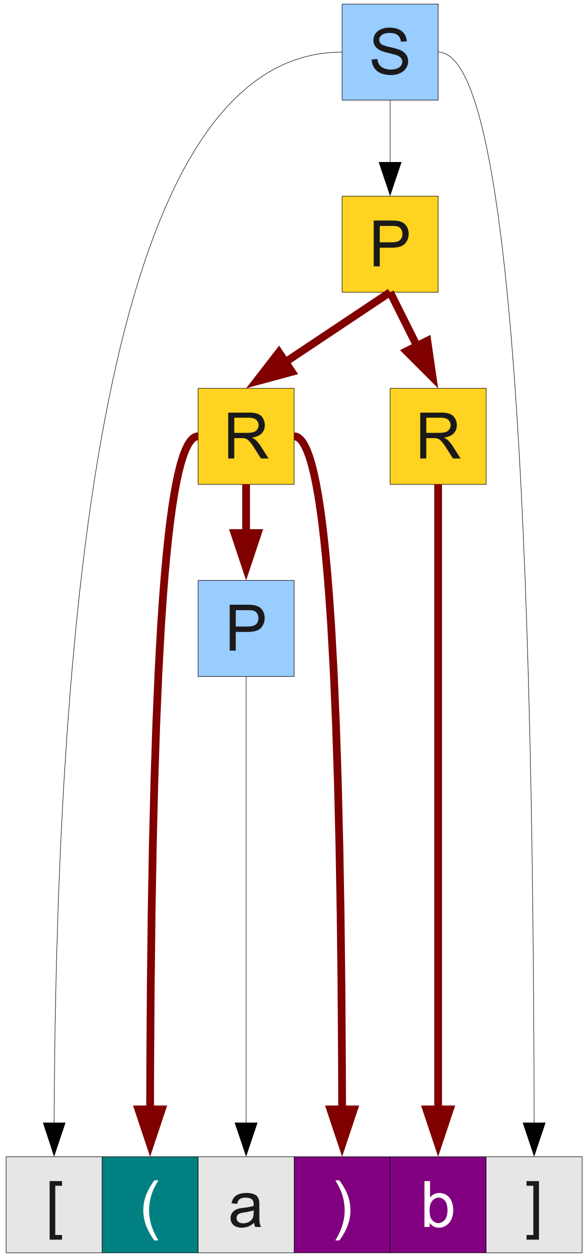


Parse Trees Revisited

$\mathbf{S} \rightarrow$ `[`$\mathbf{P}$`]`

$\mathbf{P} \rightarrow \mathbf{RR} \mid$ `a`

$\mathbf{R} \rightarrow$ `(`$\mathbf{P}$`)` $\mid$ `b`

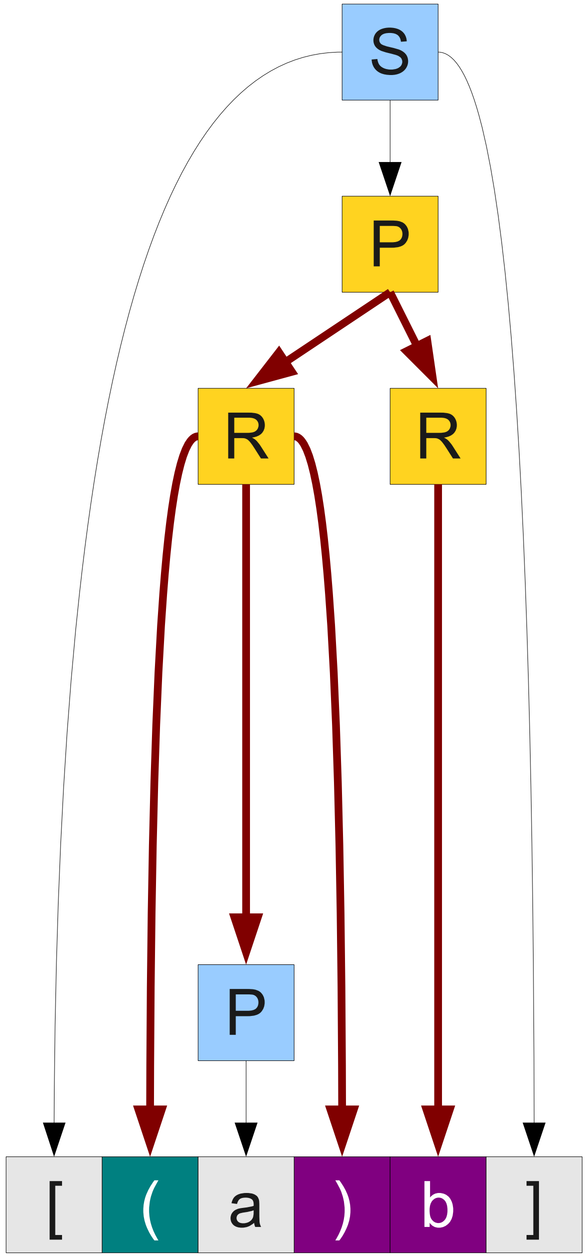


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

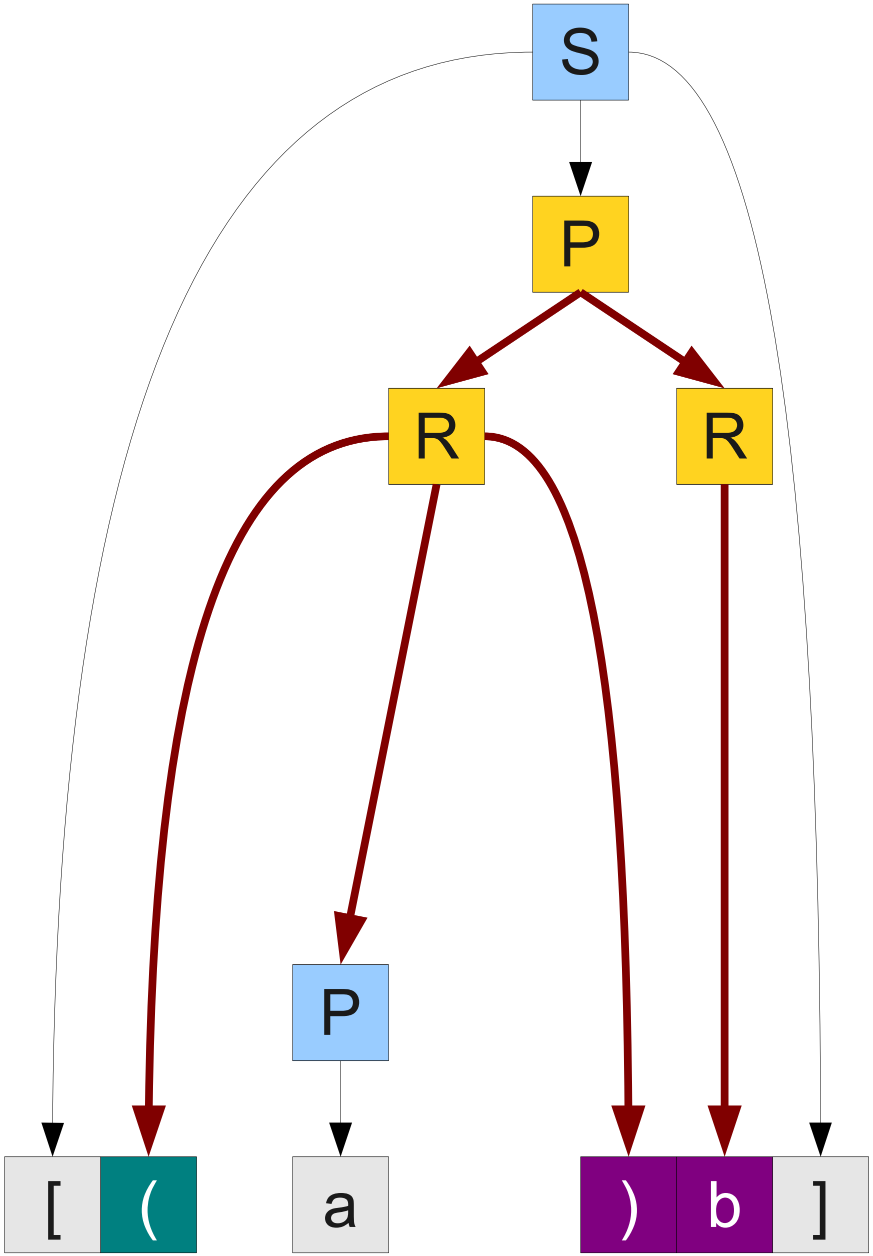


Parse Trees Revisited

$$S \rightarrow [P]$$

$$P \rightarrow RR \mid a$$

$$R \rightarrow (P) \mid b$$

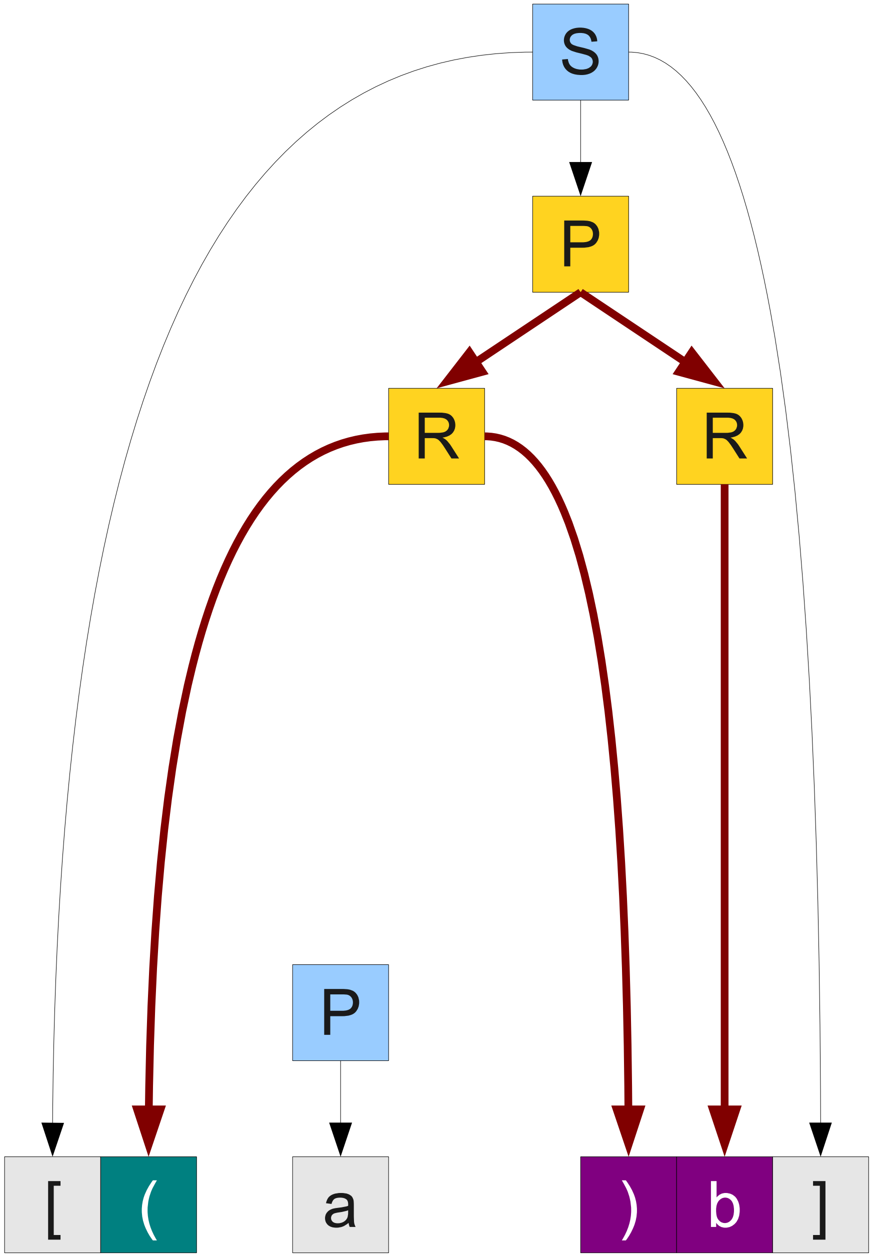


Parse Trees Revisited

$\mathbf{S} \rightarrow$ `[`$\mathbf{P}$`]`

$\mathbf{P} \rightarrow \mathbf{RR}$ | `a`

$\mathbf{R} \rightarrow$ `(`$\mathbf{P}$`)` | `b`

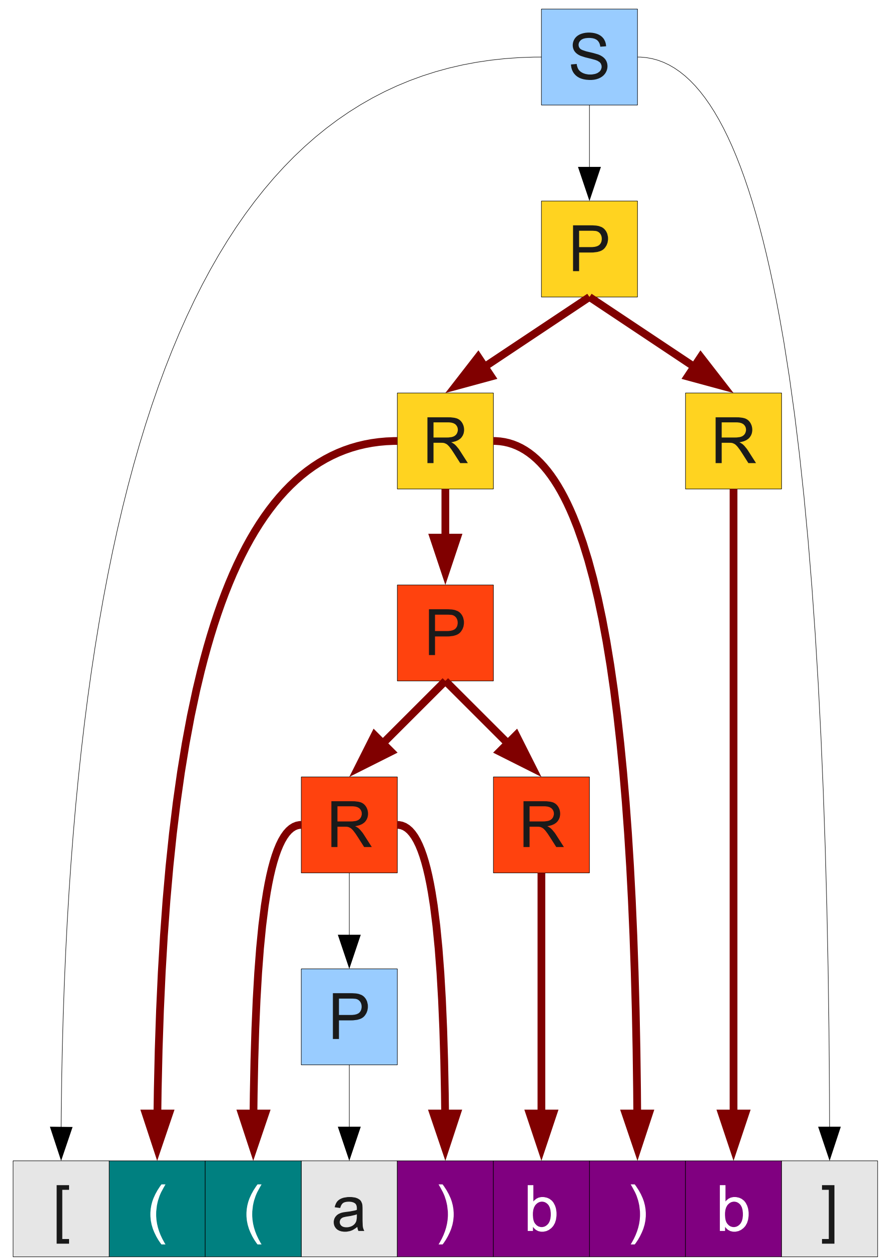


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

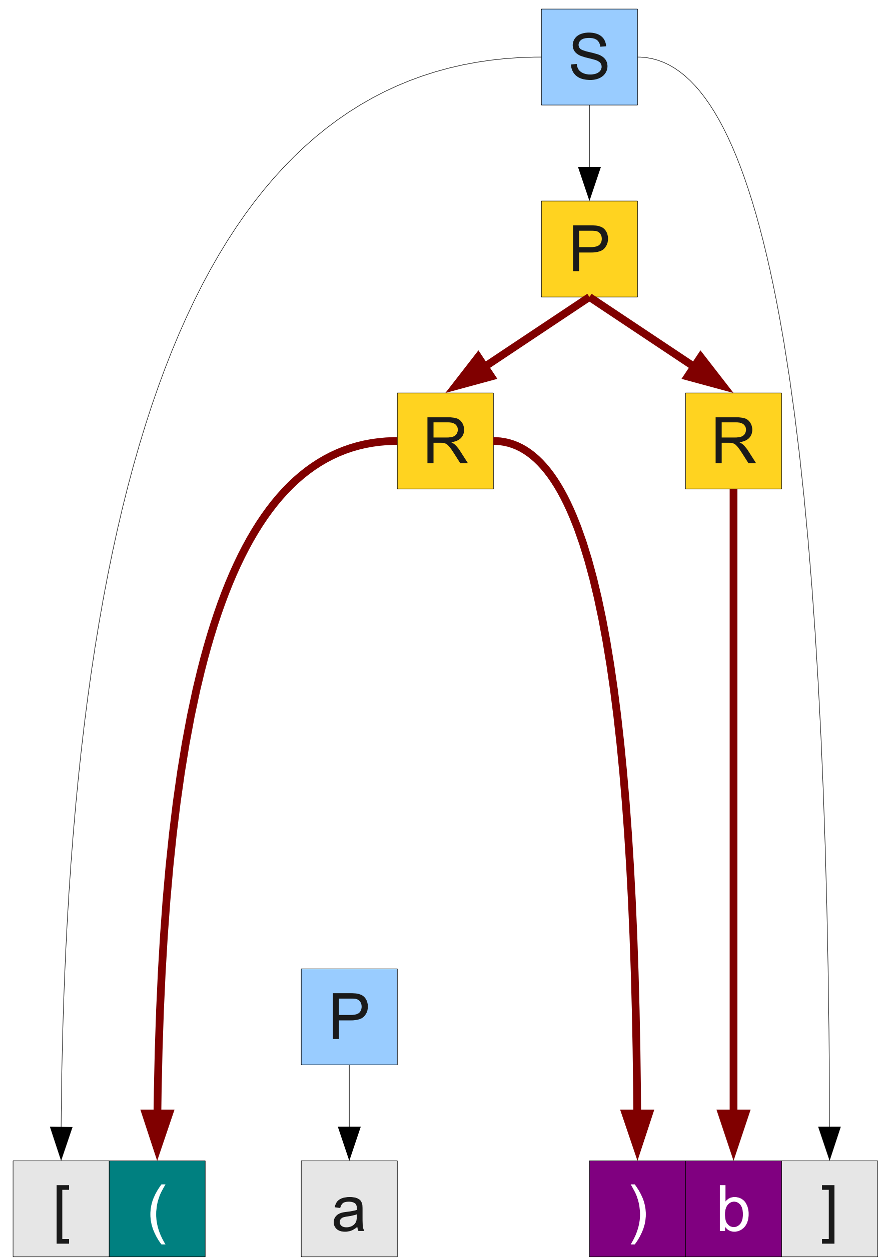


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

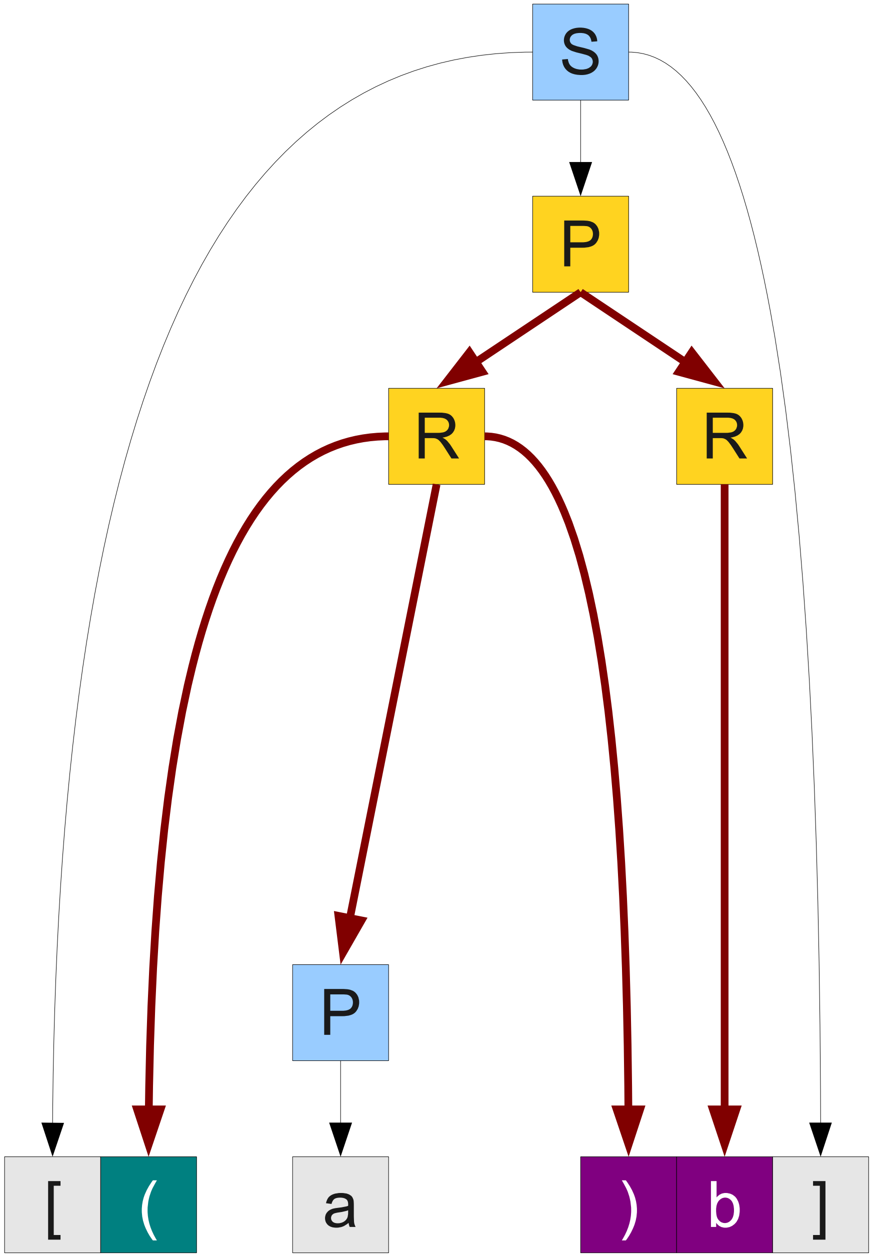


Parse Trees Revisited

$\mathbf{S} \rightarrow$ `[`$\mathbf{P}$`]`

$\mathbf{P} \rightarrow \mathbf{RR} \mid$ `a`

$\mathbf{R} \rightarrow$ `(`$\mathbf{P}$`)` $\mid$ `b`

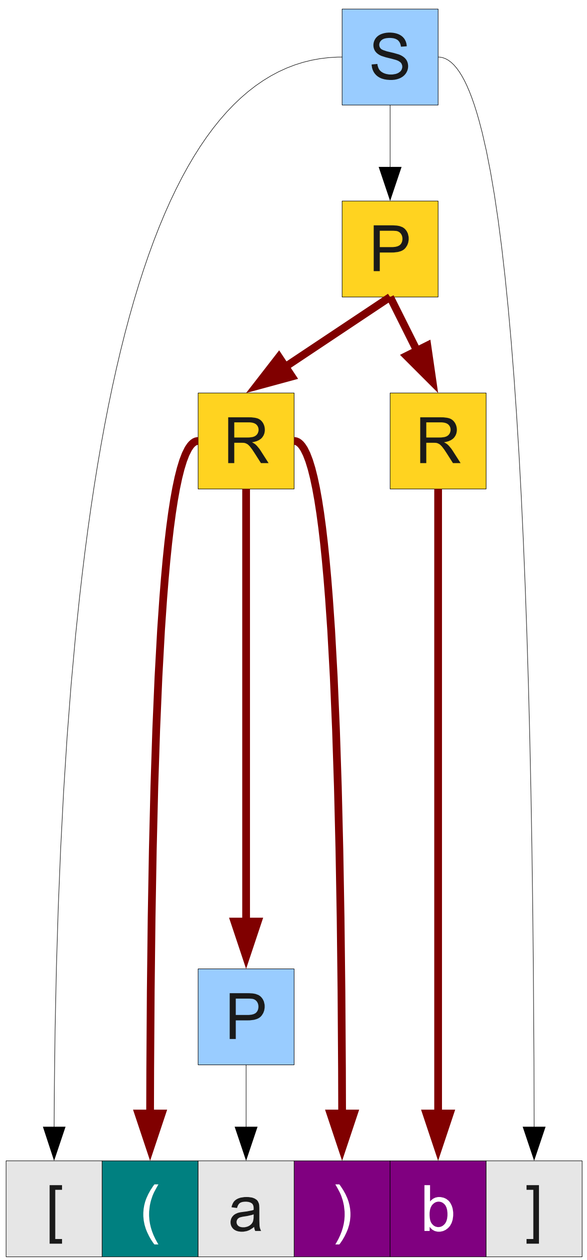


Parse Trees Revisited

S → `[`**P**`]`

P → **RR** | `a`

R → `(`**P**`)` | `b`

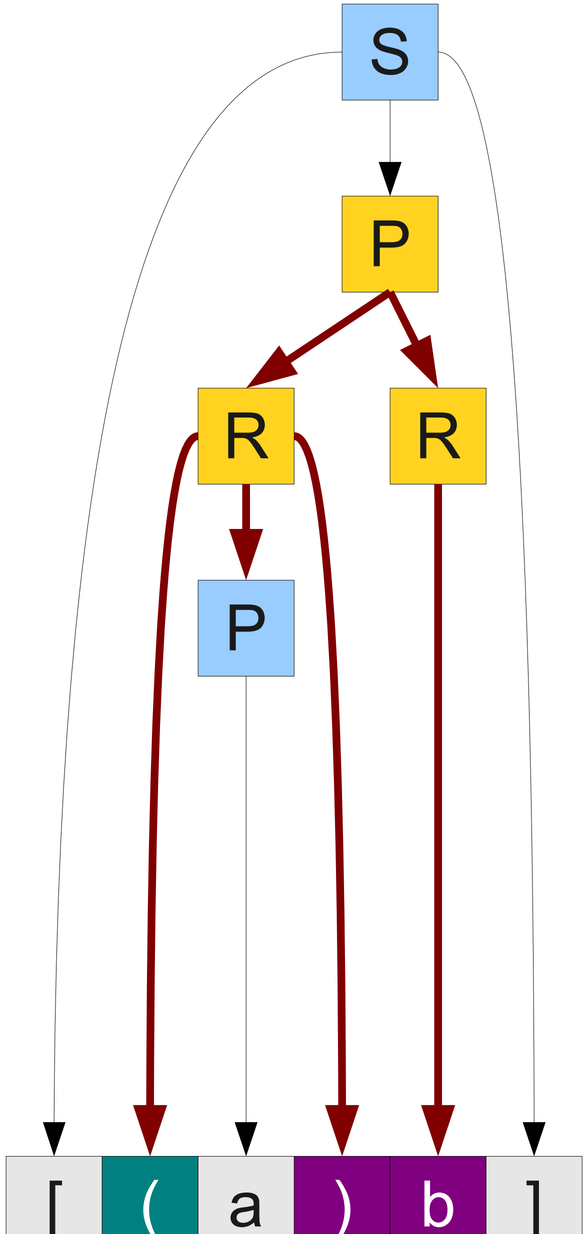


Parse Trees Revisited

S → `[`**P**`]`

P → **RR** | `a`

R → `(`**P**`)` | `b`

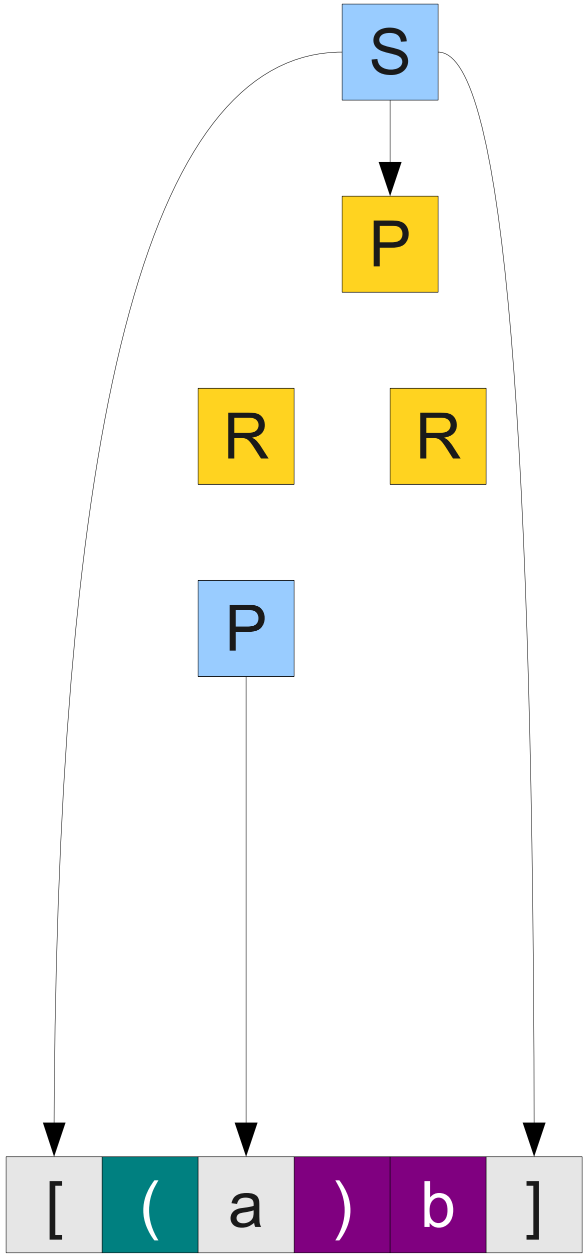


Parse Trees Revisited

$\mathbf{S} \rightarrow$ **[P]**

$\mathbf{P} \rightarrow$ **RR** | **a**

$\mathbf{R} \rightarrow$ **(P)** | **b**

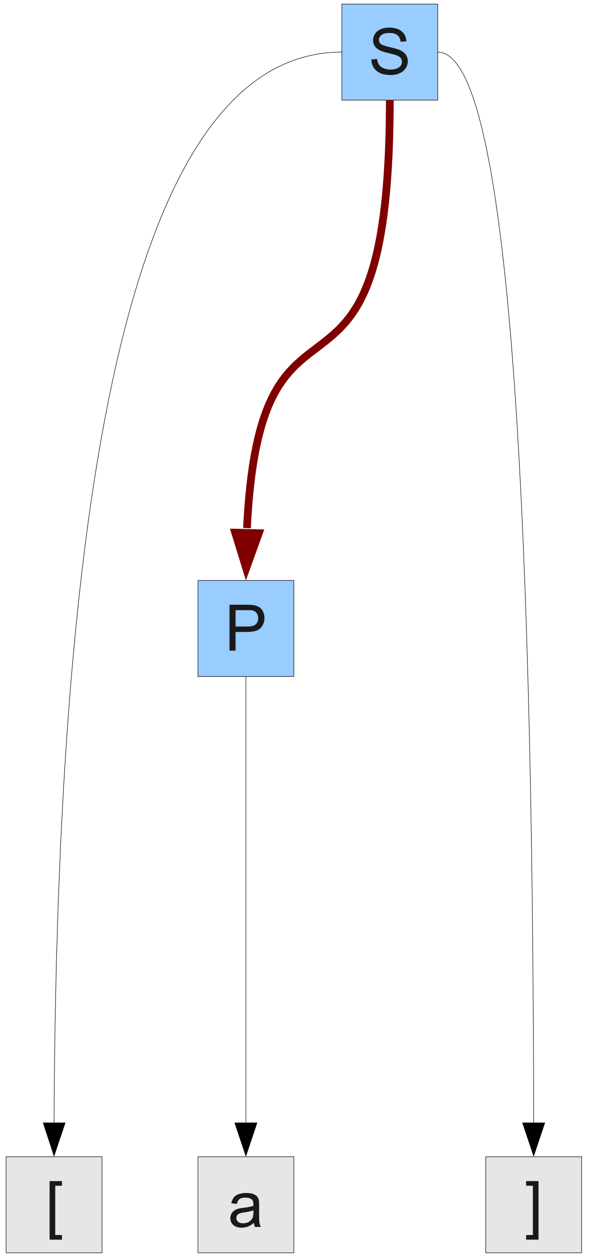


Parse Trees Revisited

S → `[`**P**`]`

P → **RR** | `a`

R → `(`**P**`)` | `b`

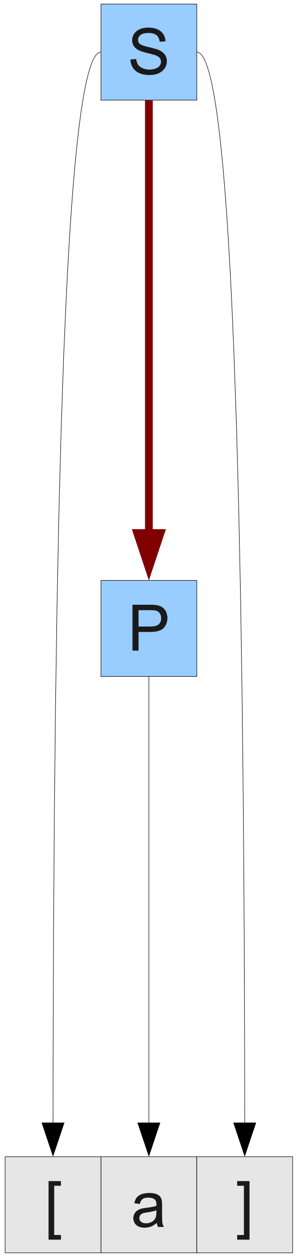


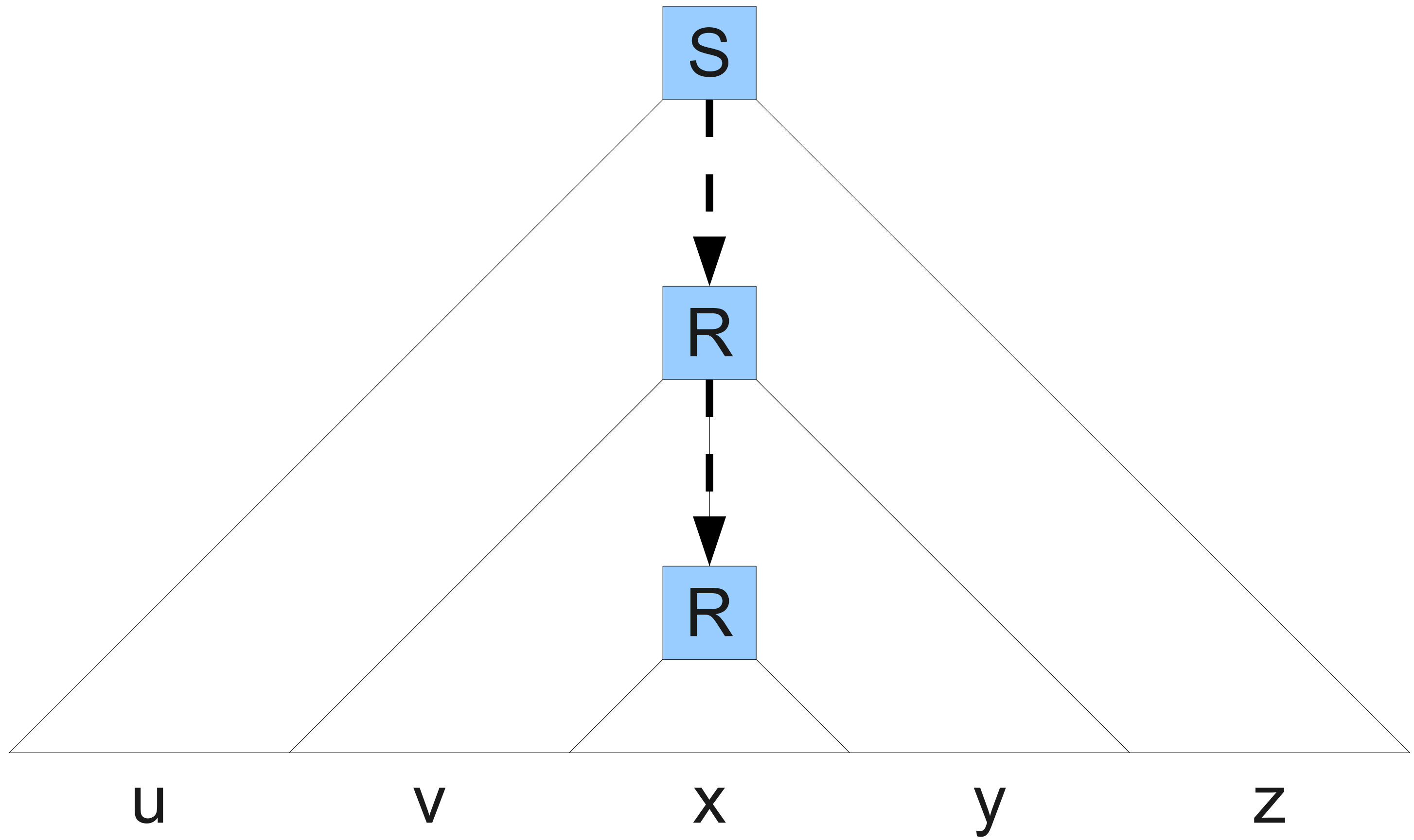
Parse Trees Revisited

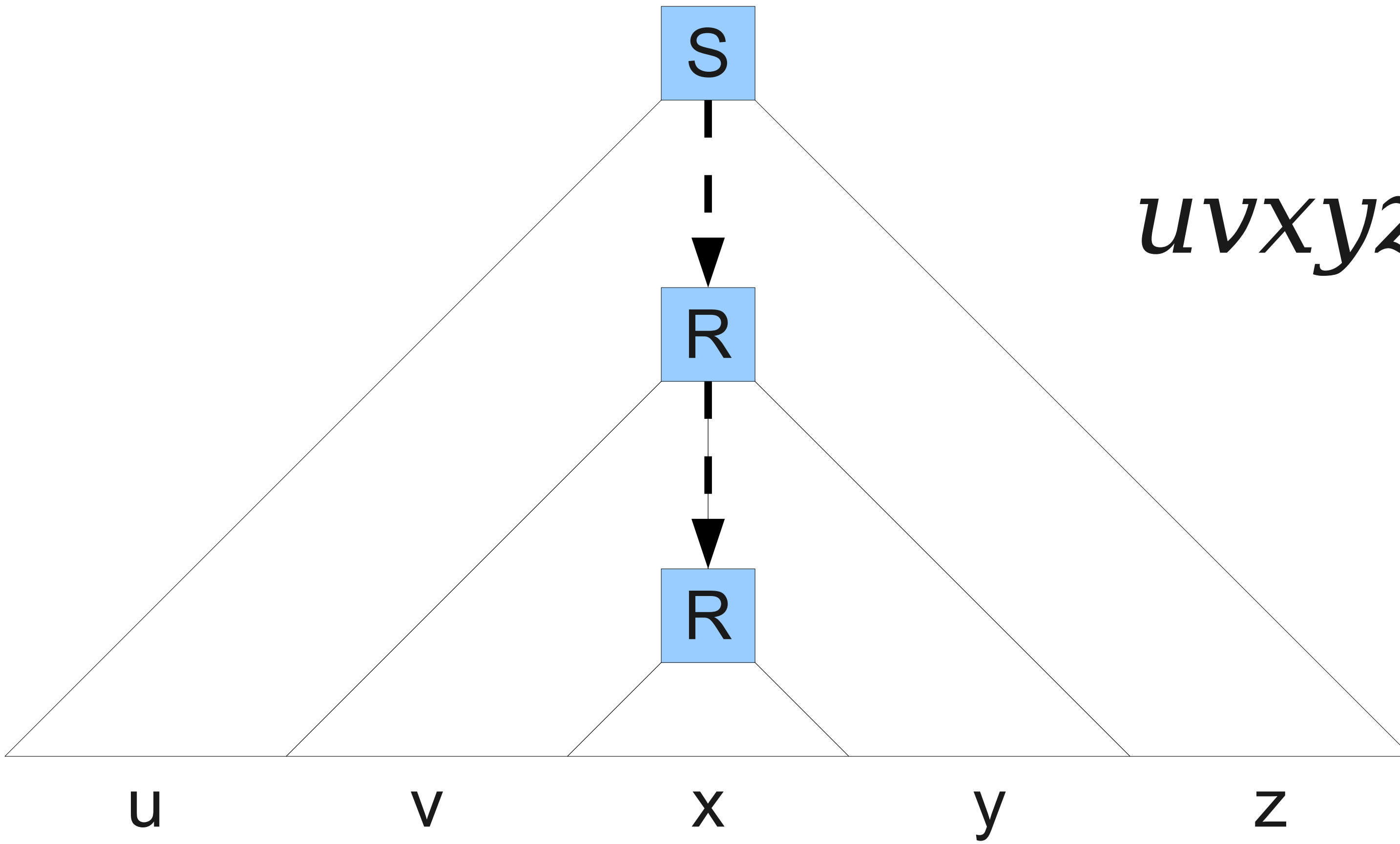
$\mathbf{S} \to \texttt{[}\mathbf{P}\texttt{]}$

$\mathbf{P} \to \mathbf{RR} \mid \texttt{a}$

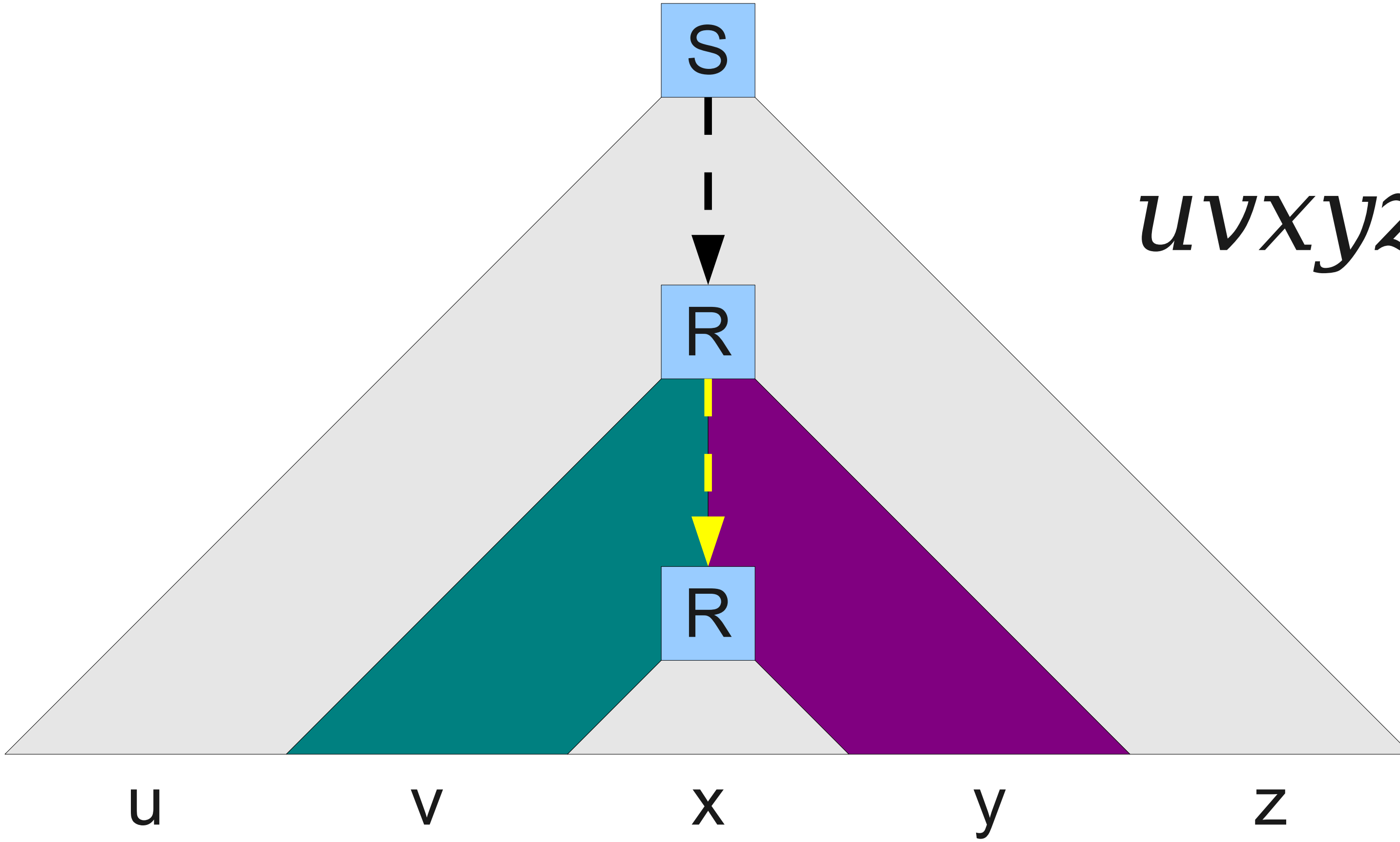
$\mathbf{R} \to \texttt{(}\mathbf{P}\texttt{)} \mid \texttt{b}$



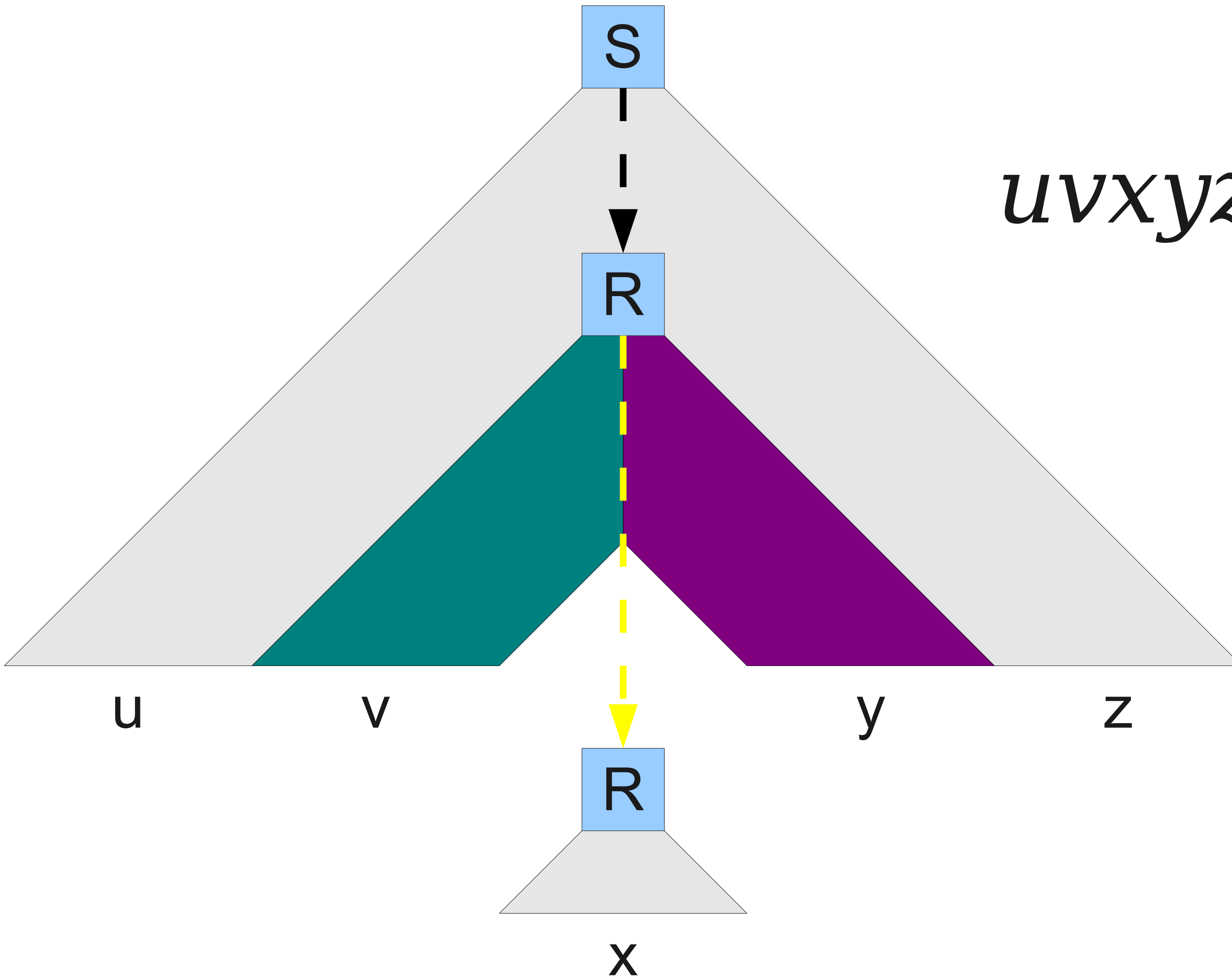

S
R
R
u
v
x
y
z



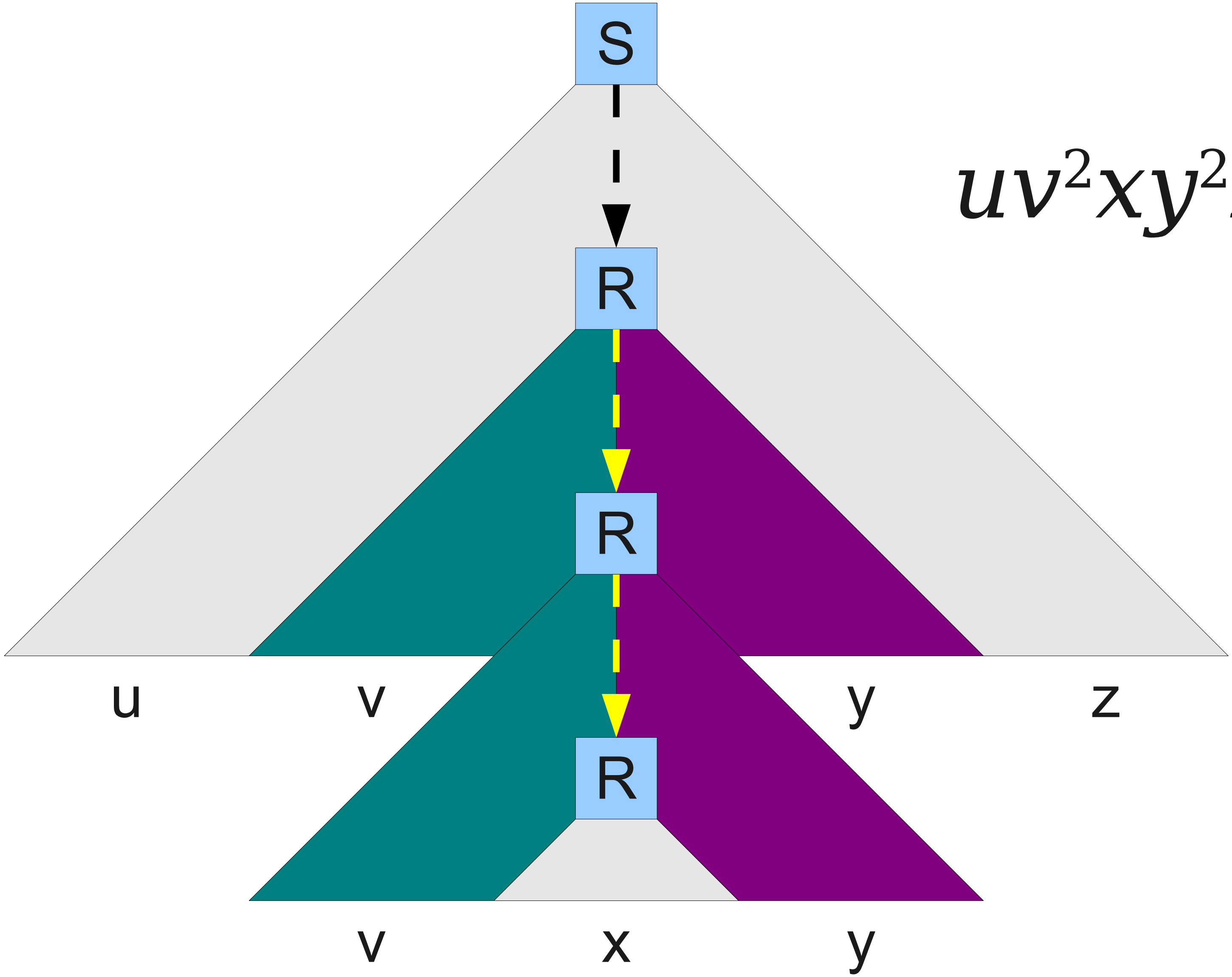
$$uvxyz \in L$$



$uvxyz \in L$

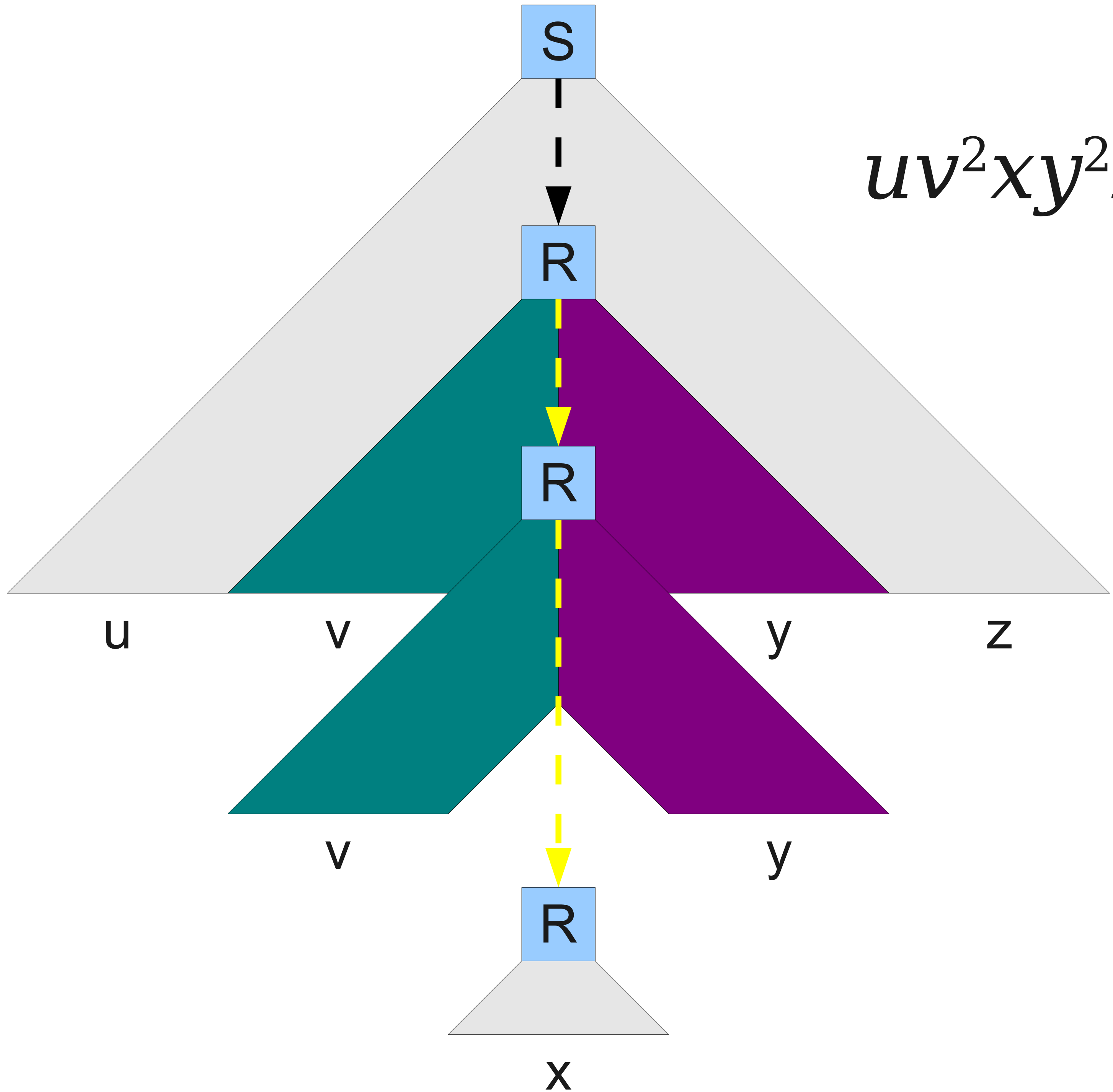


$$uvxyz \in L$$

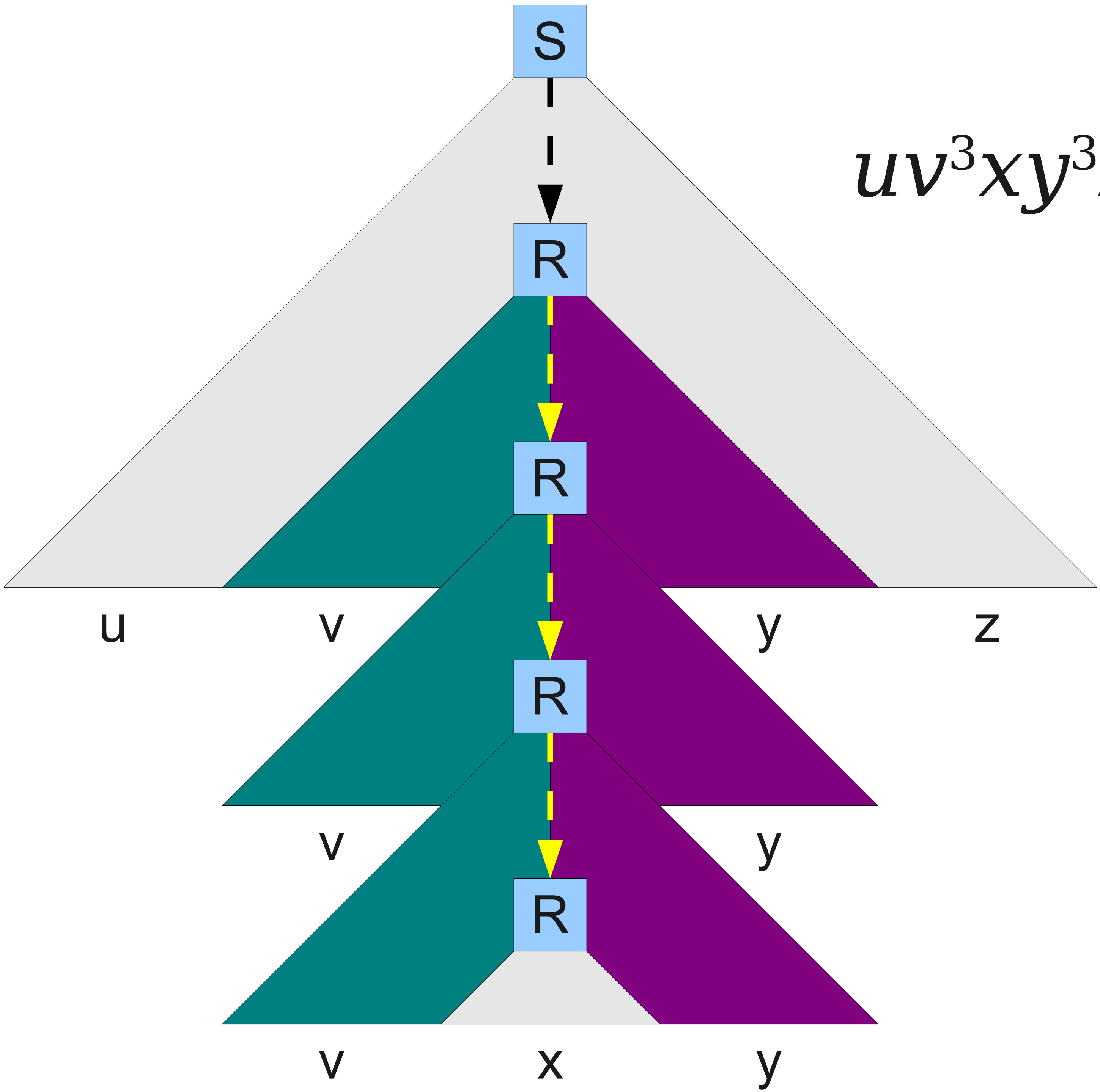


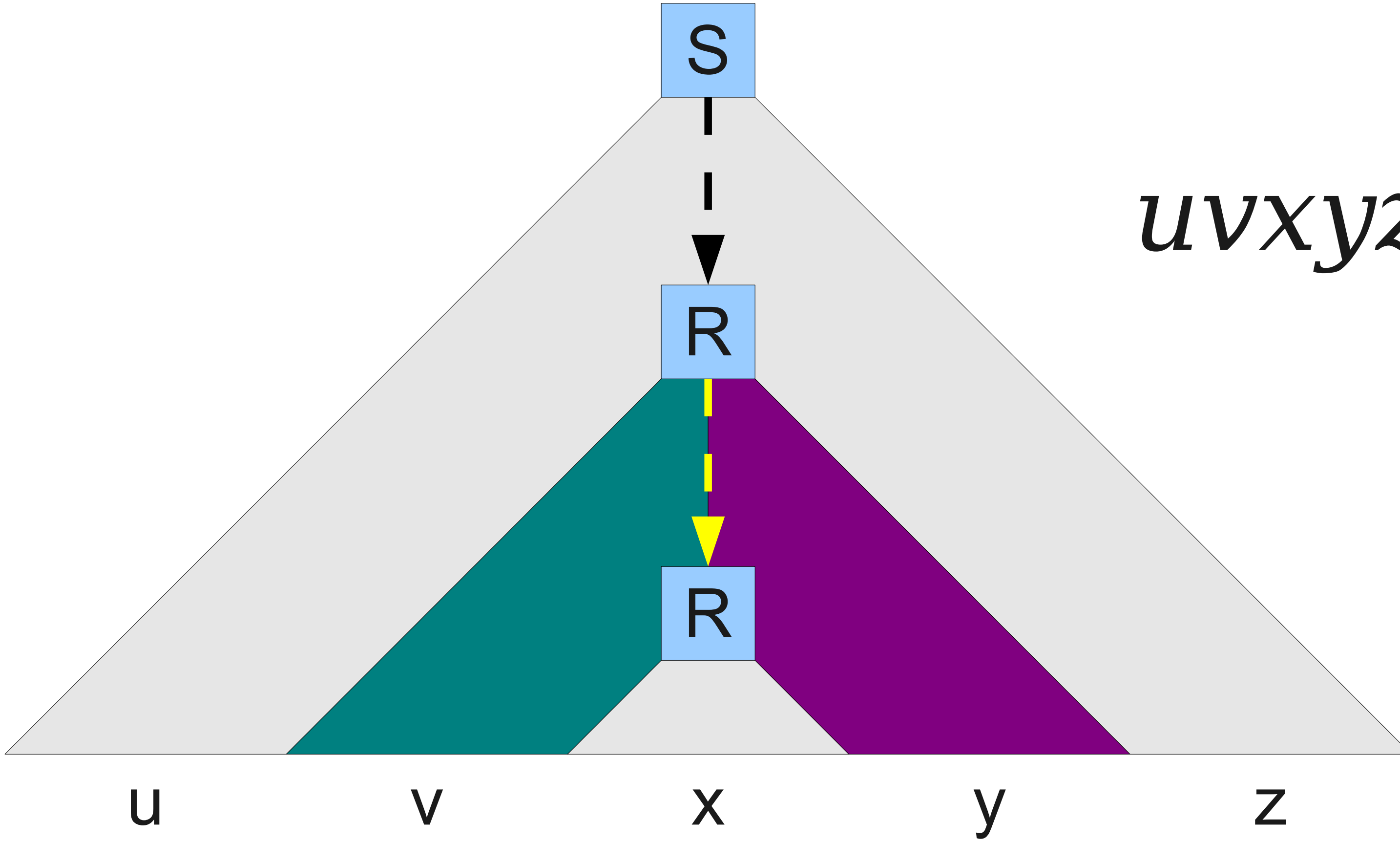
$$uv^2xy^2z \in L$$

$$uv^2xy^2z \in L$$

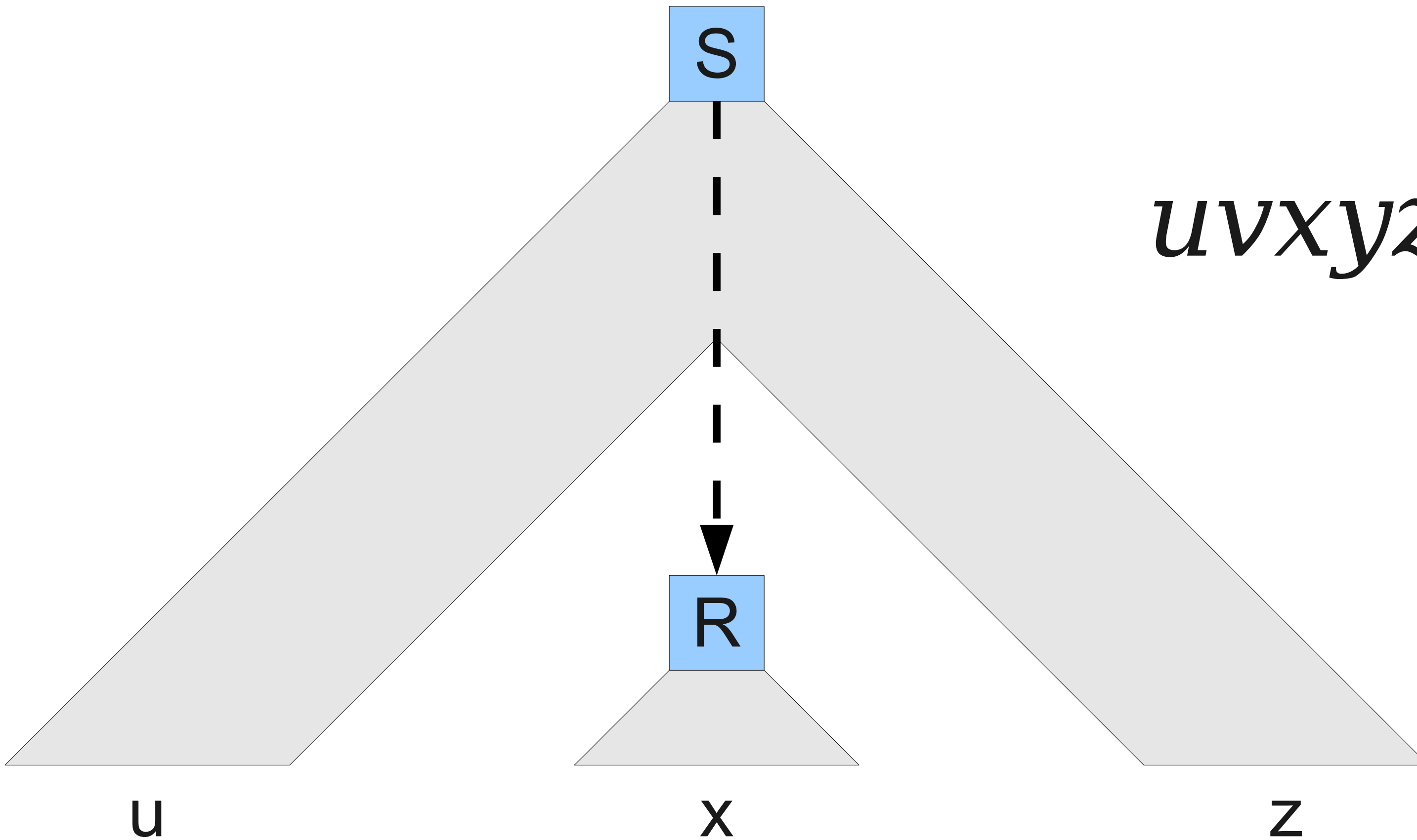


$uv^3xy^3z \in L$

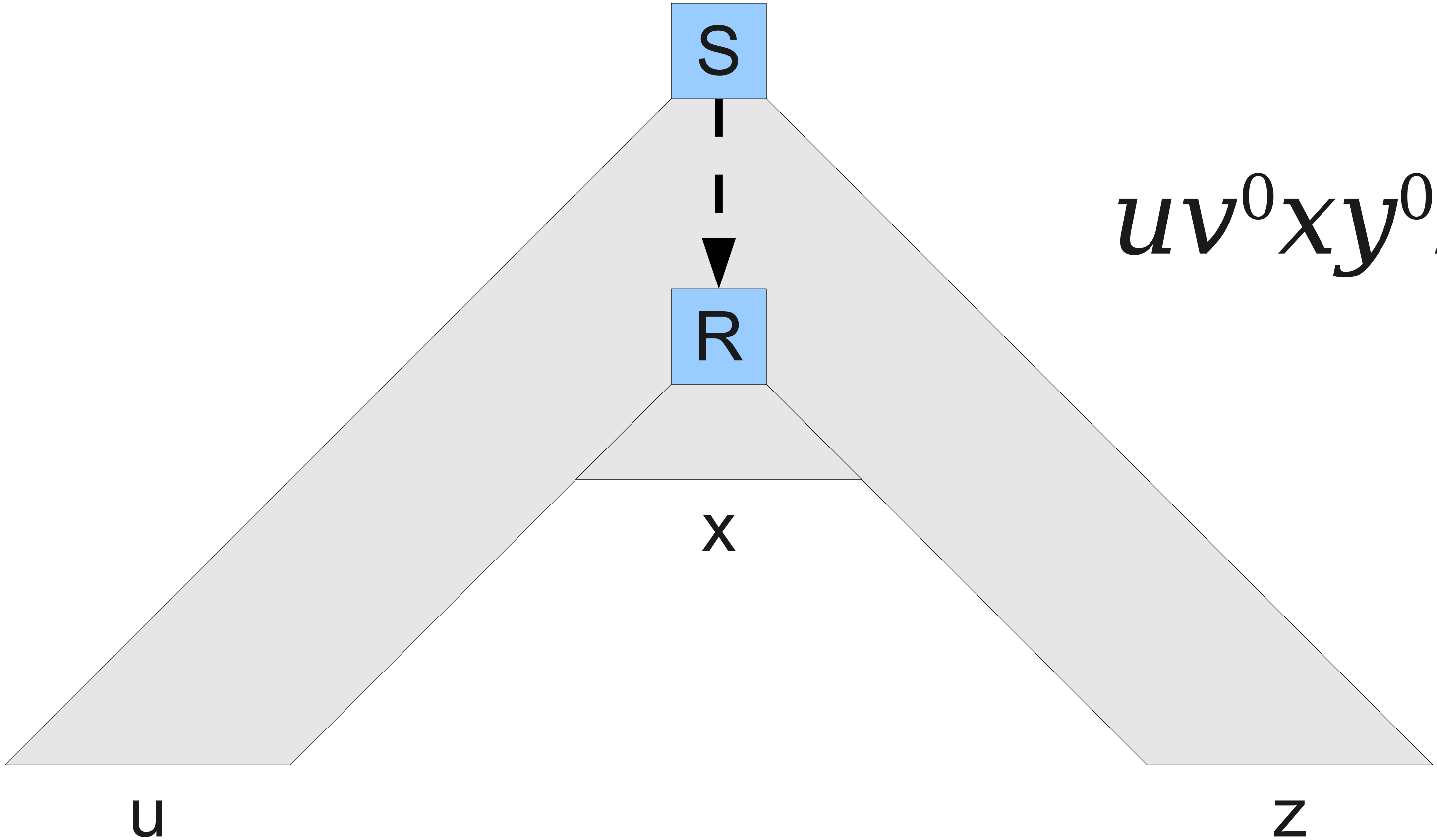




$uvxyz \in L$



$$uvxyz \in L$$



$uv^0xy^0z \in L$

The Pumping Lemma for CFLs

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz,$	w can be broken into five pieces,
$\lvert vxy \rvert \leq n,$	where the middle three pieces aren't too long,
$\lvert vy \rvert > 0$	where the 2nd and 4th pieces aren't both empty, and
$uv^ixy^iz \in L$	where the 2nd and 4th pieces can be replicated 0 or more times

The Pumping Lemma for CFLs

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$, w can be broken into five pieces,

$|vxy| \leq n$, where the middle three pieces aren't too long,

$|vy| > 0$ where the 2nd and 4th pieces aren't both empty, and

$uv^ixy^iz \in L$ where the 2nd and 4th pieces can be replicated 0 or more times

Note that we pump both v and y at the same time, not just one or the other.

The Pumping Lemma for CFLs

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$, w can be broken into five pieces,

$|vxy| \leq n$, where the middle three pieces aren't too long,

$|vy| > 0$ where the 2nd and 4th pieces aren't both empty, and

$uv^ixy^iz \in L$ where the 2nd and 4th pieces can be replicated 0 or more times

The two strings to pump, collectively, cannot be too long.

They also must be close to one another.

The Pumping Lemma for CFLs

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

The pumping length is <u>not</u> simple; see Sipser for details.

$w = uvxyz$,	w can be bro
$\lvert vxy \rvert \leq n$,	where the middle three pieces aren't too long,
$\lvert vy \rvert > 0$	where the 2nd and 4th pieces aren't both empty, and
$uv^ixy^iz \in L$	where the 2nd and 4th pieces can be replicated 0 or more times

The Pumping Lemma Game

$L = \{w \in \{0,1,2\}^* \mid w$ has the same number of 0s, 1s, 2s$\}$

ADVERSARY	YOU
Maliciously choose pumping length n.	
	Cleverly choose a string $w \in L$, $\lvert w \rvert \geq n$
Maliciously split $w = uvxyz$, with $\lvert vy \rvert > 0$ and $\lvert vxy \rvert \leq n$	
	Cleverly choose k such that $uv^kxy^kz \notin L$
Grrr! Aaaargh!	

The Pumping Lemma Game

$L = \{w \in \{0,1,2\}^* \mid w$ has the same number of 0s, 1s, 2s$\}$

ADVERSARY	YOU
Maliciously choose pumping length n.	
	Cleverly choose a string $w \in L$, $\lvert w \rvert \geq n$
Maliciously split $w = uvxyz$, with $\lvert vy \rvert > 0$ and $\lvert vxy \rvert \leq n$	
	Cleverly choose k such that $uv^kxy^kz \notin L$
Grrr! Aaaargh!	

Try Your Best!

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of `0`s, `1`s, `2`s$\}$ is not a CFL.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\texttt{0},\texttt{1},\texttt{2}\}^* \mid w$ has the same # of $\texttt{0}$s, $\texttt{1}$s, $\texttt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma.

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$. Thus $w \in L$ and $|w| = 3n \geq n$.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$.

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^i xy^i z \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n 1^n 2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^i xy^i z \in L$. We consider two cases for vxy:

Proofs using the pumping lemma for CFLs tend to be much harder than those for regular languages because there is no restriction on where in the string the portion that can be pumped can be. The string to pump must be very carefully constructed.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in $\mathtt{0}^n$, $\mathtt{1}^n$, or $\mathtt{2}^n$.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in $\mathtt{0}^n$, $\mathtt{1}^n$, or $\mathtt{2}^n$. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of $\mathtt{0}$ or $\mathtt{1}$ or $\mathtt{2}$ than the other two symbols.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{\mathtt{0},\mathtt{1},\mathtt{2}\}^* \mid w$ has the same # of $\mathtt{0}$s, $\mathtt{1}$s, $\mathtt{2}$s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in $\mathtt{0}^n$, $\mathtt{1}^n$, or $\mathtt{2}^n$. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of $\mathtt{0}$ or $\mathtt{1}$ or $\mathtt{2}$ than the other two symbols.

Case 2: vxy either consists of $\mathtt{0}$s and $\mathtt{1}$s or of $\mathtt{1}$s and $\mathtt{2}$s (it cannot consist of all three symbols, because $|vxy| \leq n$).

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n1^n2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in 0^n, 1^n, or 2^n. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of 0 or 1 or 2 than the other two symbols.

Case 2: vxy either consists of 0s and 1s or of 1s and 2s (it cannot consist of all three symbols, because $|vxy| \leq n$).

Note how we chose w so that vxy can't span all three groups of symbols. This makes it impossible to pump all three groups at once.

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n1^n2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in 0^n, 1^n, or 2^n. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of 0 or 1 or 2 than the other two symbols.

Case 2: vxy either consists of 0s and 1s or of 1s and 2s (it cannot consist of all three symbols, because $|vxy| \leq n$). Then if vxy has no 2s in it, $uv^2xy^2z \notin L$ since it contains more 0s or 1s than 2s.

For any context-free language L,

There exists a positive natural number n such that

For any $w \in L$ with $|w| \geq n$,

There exists strings u, v, x, y, z such that

For any natural number i,

$w = uvxyz$,

$|vxy| \leq n$,

$|vy| > 0$

$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n1^n2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in 0^n, 1^n, or 2^n. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of 0 or 1 or 2 than the other two symbols.

Case 2: vxy either consists of 0s and 1s or of 1s and 2s (it cannot consist of all three symbols, because $|vxy| \leq n$). Then if vxy has no 2s in it, $uv^2xy^2z \notin L$ since it contains more 0s or 1s than 2s. Similarly, if vxy has no 0s in it $uv^2xy^2z \notin L$ because it contains more 1s or 2s than 0s.

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n1^n2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in 0^n, 1^n, or 2^n. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of 0 or 1 or 2 than the other two symbols.

Case 2: vxy either consists of 0s and 1s or of 1s and 2s (it cannot consist of all three symbols, because $|vxy| \leq n$). Then if vxy has no 2s in it, $uv^2xy^2z \notin L$ since it contains more 0s or 1s than 2s. Similarly, if vxy has no 0s in it $uv^2xy^2z \notin L$ because it contains more 1s or 2s than 0s.

In either case, we contradict the pumping lemma.

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n1^n2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in 0^n, 1^n, or 2^n. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of 0 or 1 or 2 than the other two symbols.

Case 2: vxy either consists of 0s and 1s or of 1s and 2s (it cannot consist of all three symbols, because $|vxy| \leq n$). Then if vxy has no 2s in it, $uv^2xy^2z \notin L$ since it contains more 0s or 1s than 2s. Similarly, if vxy has no 0s in it $uv^2xy^2z \notin L$ because it contains more 1s or 2s than 0s.

In either case, we contradict the pumping lemma. Thus our assumption must have been wrong, so L is not a CFL.

For any context-free language L,
There exists a positive natural number n such that
For any $w \in L$ with $|w| \geq n$,
There exists strings u, v, x, y, z such that
For any natural number i,
$w = uvxyz$,
$|vxy| \leq n$,
$|vy| > 0$
$uv^ixy^iz \in L$

Theorem: $L = \{w \in \{0,1,2\}^* \mid w$ has the same # of 0s, 1s, 2s$\}$ is not a CFL.

Proof: By contradiction; assume L is a CFL. Let n be the pumping length guaranteed by the pumping lemma. Let $w = 0^n1^n2^n$. Thus $w \in L$ and $|w| = 3n \geq n$. Therefore we can write $w = uvxyz$ such that $|vxy| \leq n$, $|vy| > 0$, and for any $i \in \mathbb{N}$, $uv^ixy^iz \in L$. We consider two cases for vxy:

Case 1: vxy is completely contained in 0^n, 1^n, or 2^n. In that case, the string $uv^2xy^2z \notin L$, because this string has more copies of 0 or 1 or 2 than the other two symbols.

Case 2: vxy either consists of 0s and 1s or of 1s and 2s (it cannot consist of all three symbols, because $|vxy| \leq n$). Then if vxy has no 2s in it, $uv^2xy^2z \notin L$ since it contains more 0s or 1s than 2s. Similarly, if vxy has no 0s in it $uv^2xy^2z \notin L$ because it contains more 1s or 2s than 0s.

In either case, we contradict the pumping lemma. Thus our assumption must have been wrong, so L is not a CFL. ■

Using the Pumping Lemma

- Keep the following in mind when using the context-free pumping lemma when $w = uvxyz$:
  - Both v and y must be pumped at the same time.
  - v and y need not be contiguous in the string.
  - One of v and y may be empty.
  - vxy may be anywhere in the string.
- I **strongly suggest** reading through Sipser to get a better sense for how these proofs work.

Next Time

- **Turing Machines**
  - A powerful, versatile automaton.
  - Programming Turing machines.