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 → `/`

E
⇒ **E Op E**
⇒ **E Op** `int`
⇒ `int` **Op** `int`
⇒ `int / int`

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**

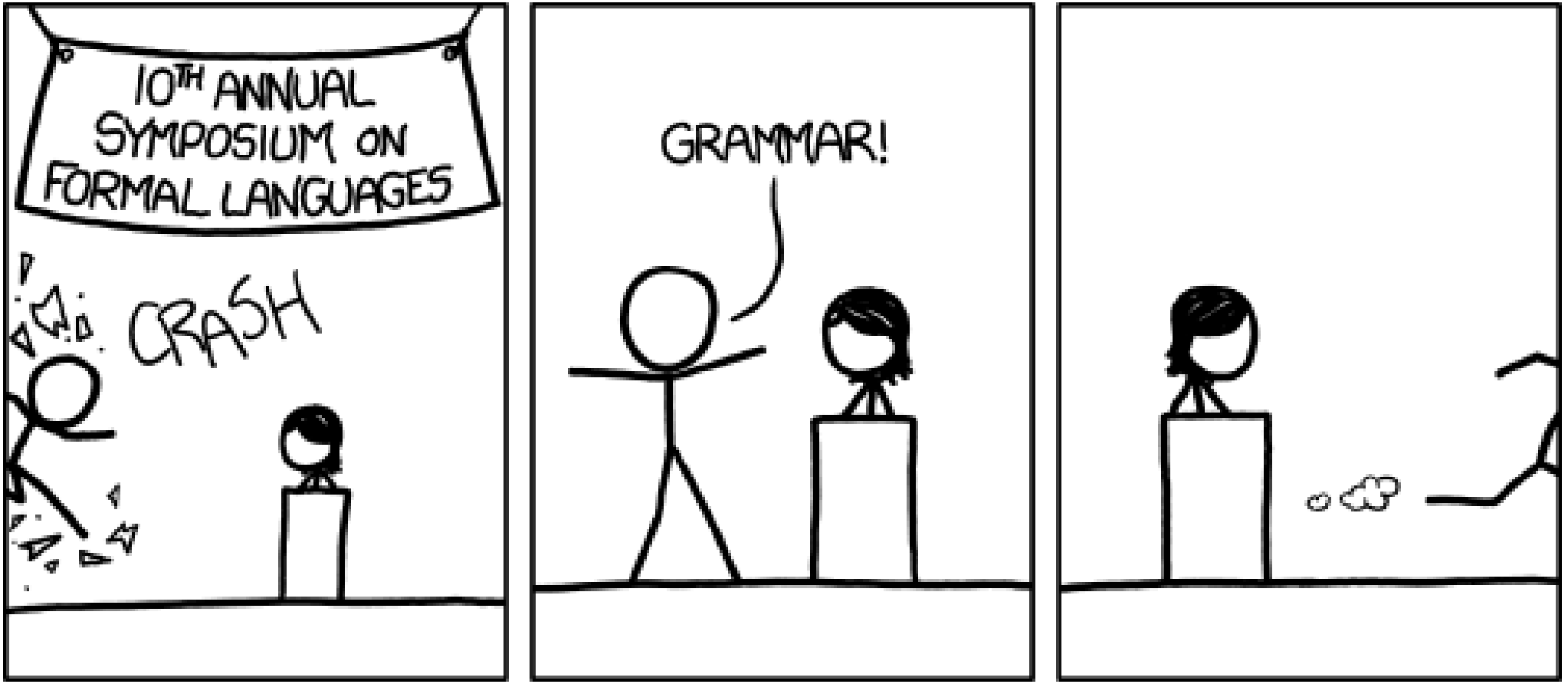
E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`



http://xkcd.com/1090/

A Notational Shorthand

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

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.

S → `a*b`

From Regexes to CFGs

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

$$S \rightarrow Ab$$
$$A \rightarrow Aa \mid \varepsilon$$

From Regexes to CFGs

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

$$S \rightarrow aX$$
$$X \rightarrow b \mid C$$
$$C \rightarrow Cc \mid \varepsilon$$

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

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

- 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

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

Derivations

E → **E Op E** | `int` | (**E**)
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)`

- This sequence of steps is called a **derivation**.
- A string $\alpha \mathbf{A} \omega$ **yields** string $\alpha\gamma\omega$ iff $\mathbf{A} \rightarrow \gamma$ 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:

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

- 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 $\Sigma = \{$`0`, `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 `0`ω`0` and `1`ω`1` are palindromes.

S → `ε` | `0` | `1` | `0`**S**`0` | `1`**S**`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

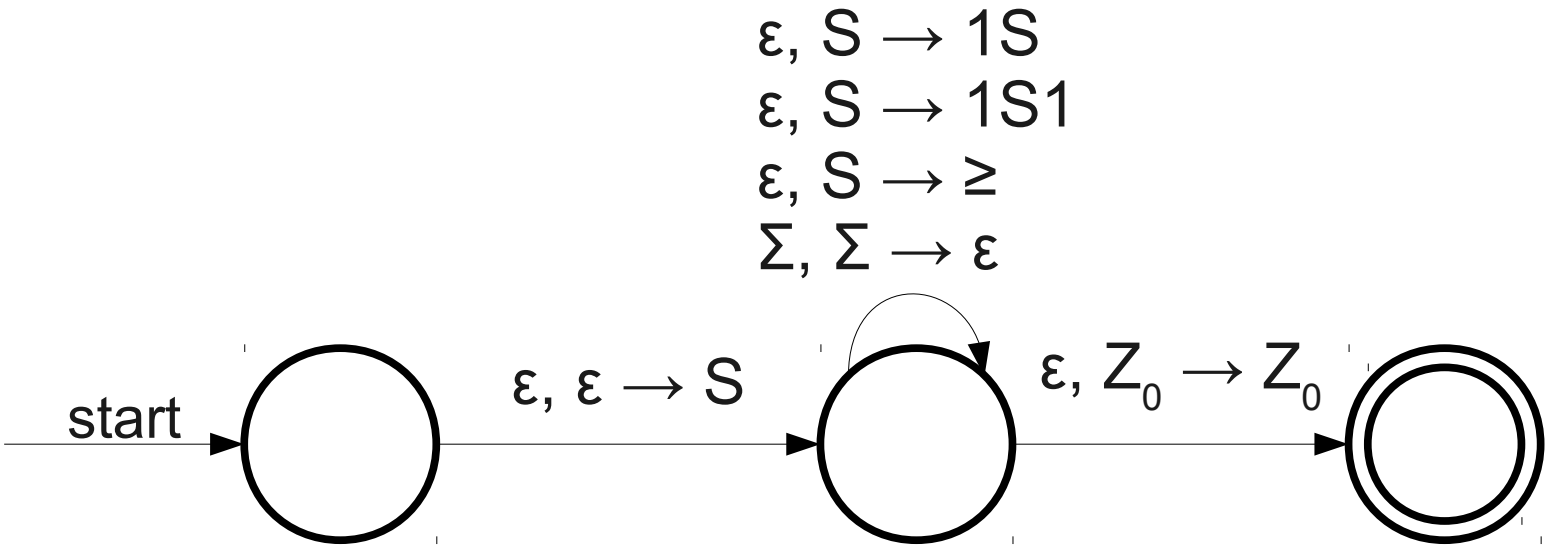
- Example: Let $\Sigma = \{ \texttt{1}, \texttt{≥} \}$ and let $GE = \{ \texttt{1}^m\texttt{≥}\texttt{1}^n \mid m, n \in \mathbb{N} \wedge m \geq n \}$
  - $\texttt{111≥11} \in GE$
  - $\texttt{11≥11} \in GE$
  - $\texttt{1111≥11} \in GE$
  - $\texttt{≥} \in GE$
- One CFG for GE is the following:

$$\mathbf{S} \to \texttt{1}\mathbf{S}\texttt{1} \mid \texttt{1}\mathbf{S} \mid \texttt{≥}$$

- How would we build a PDA for GE?

From CFGs to PDAs

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



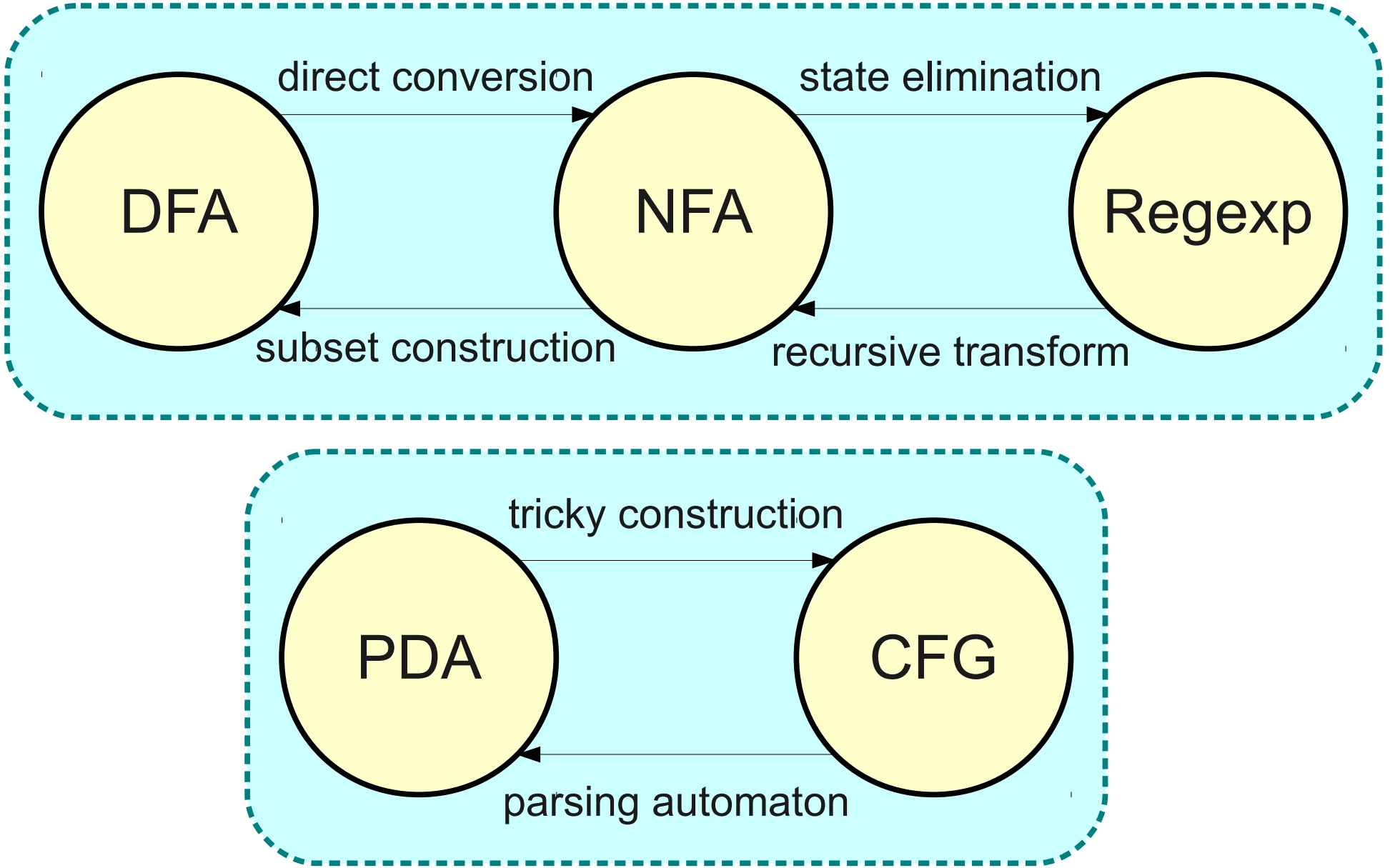
From CFGs to PDAs

- 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, $Z_0 \rightarrow Z_0$ 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 Op E

⇒ int Op E

⇒ int * E

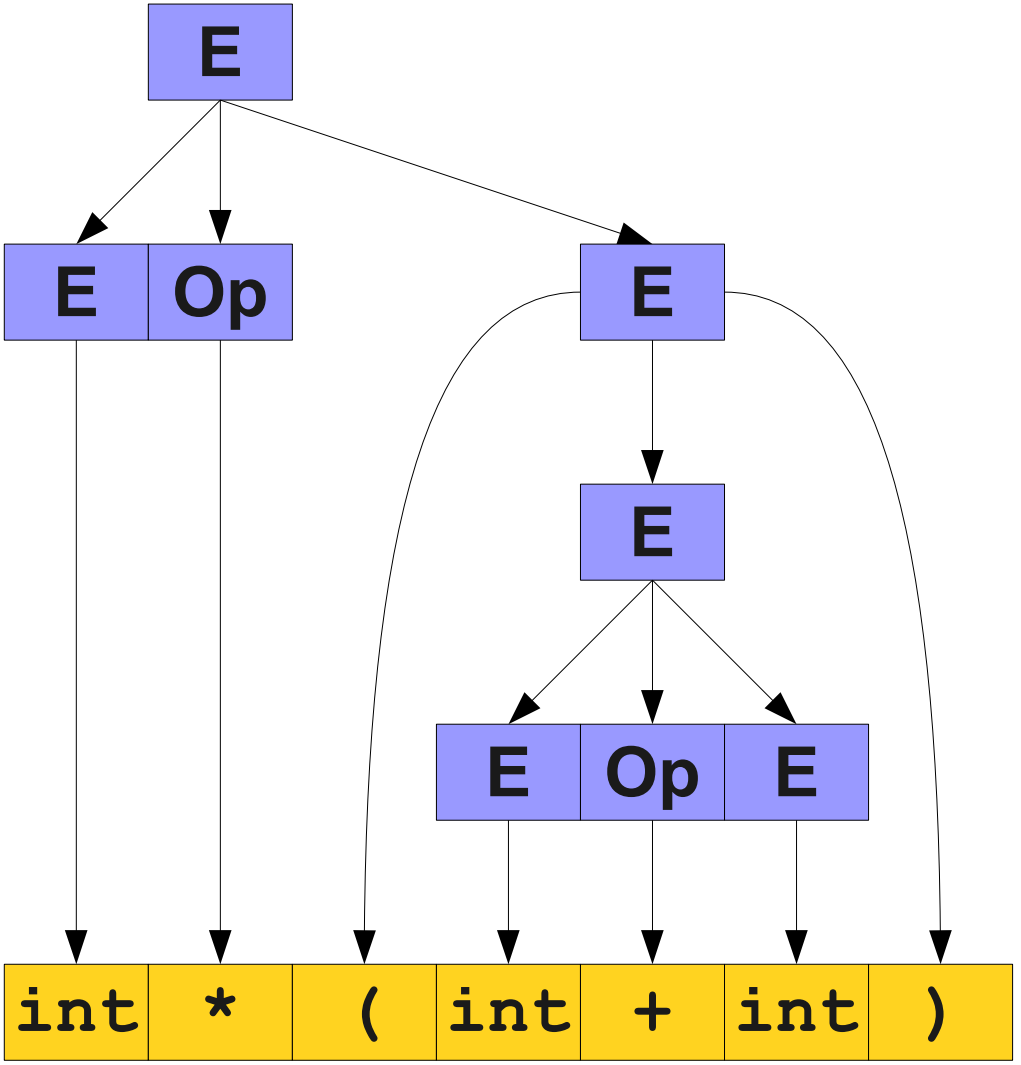
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



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

Parse Trees

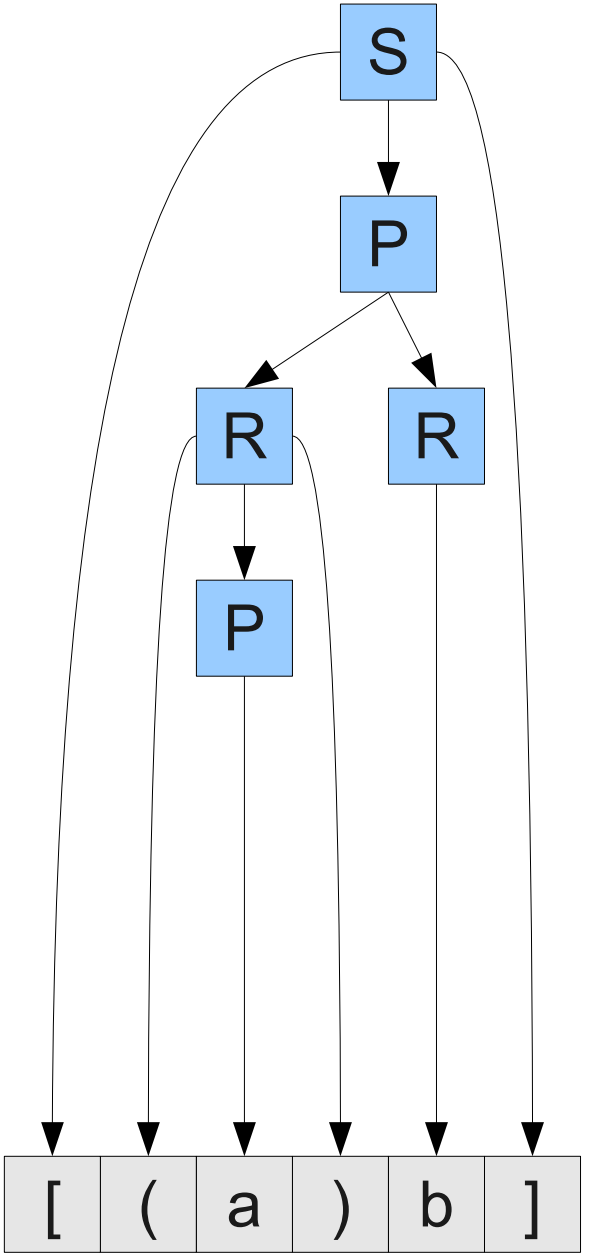
- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Walking the leaves in order gives the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

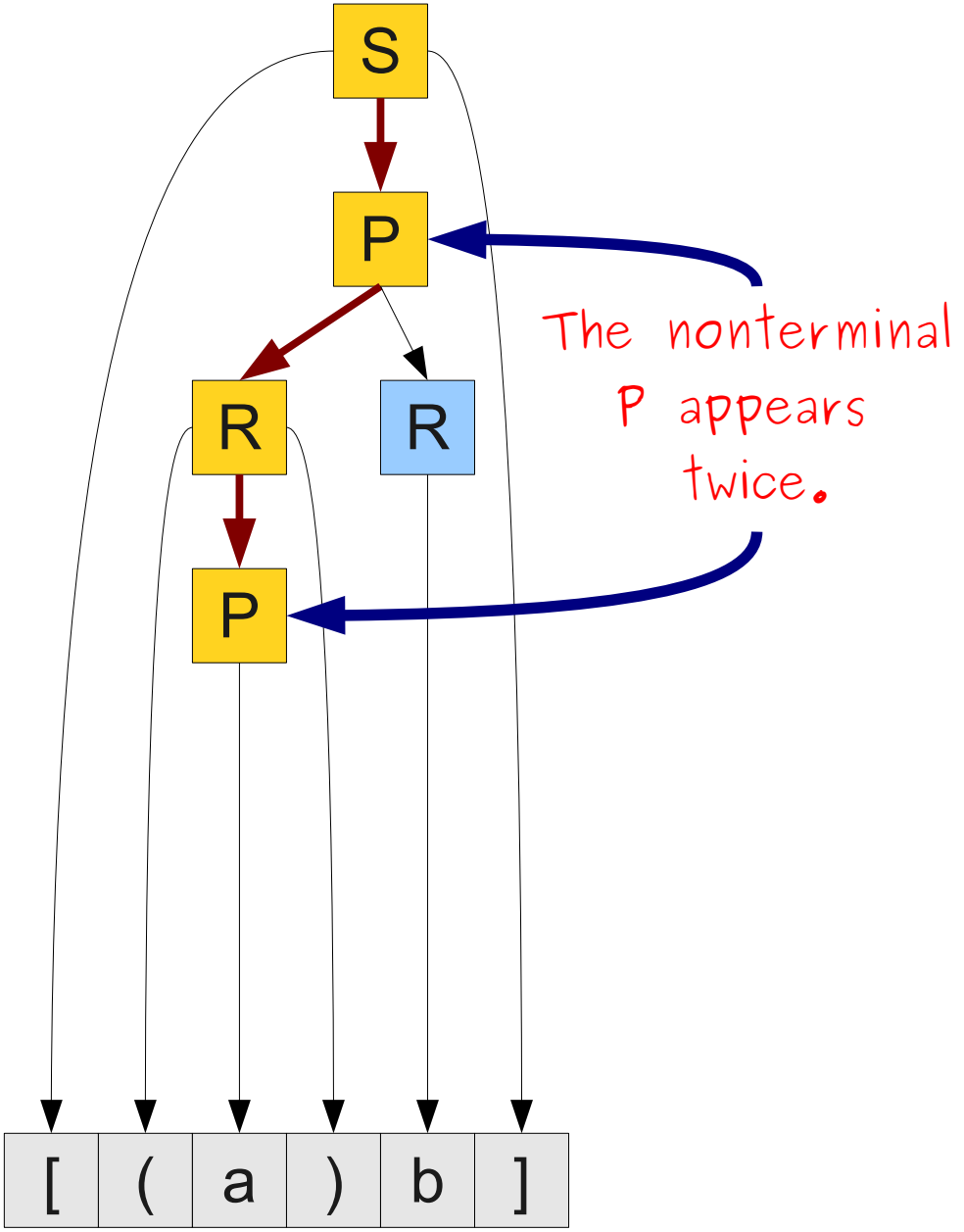


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

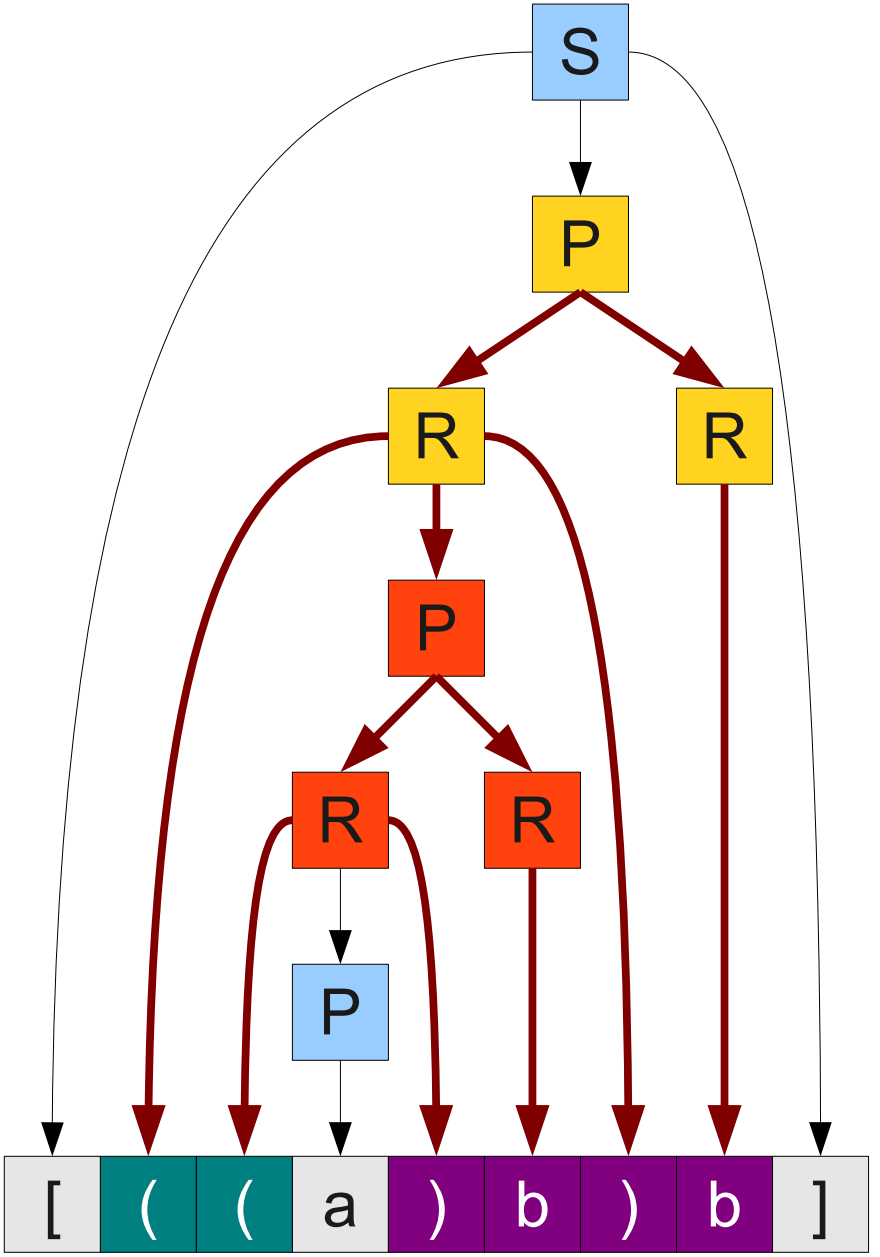


Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**

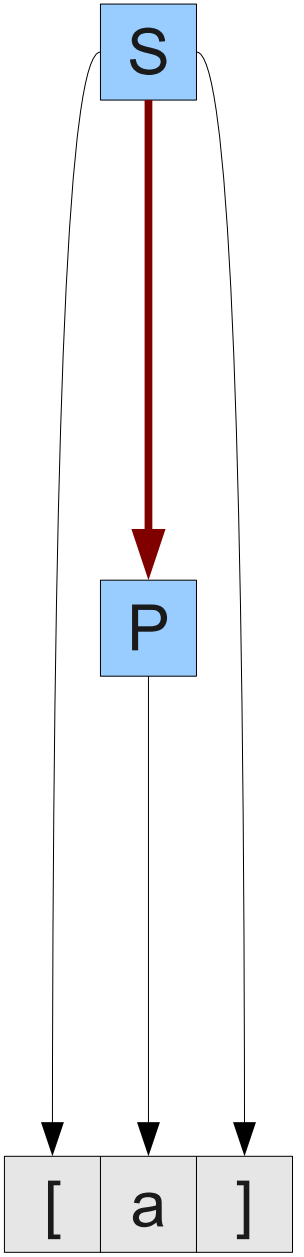


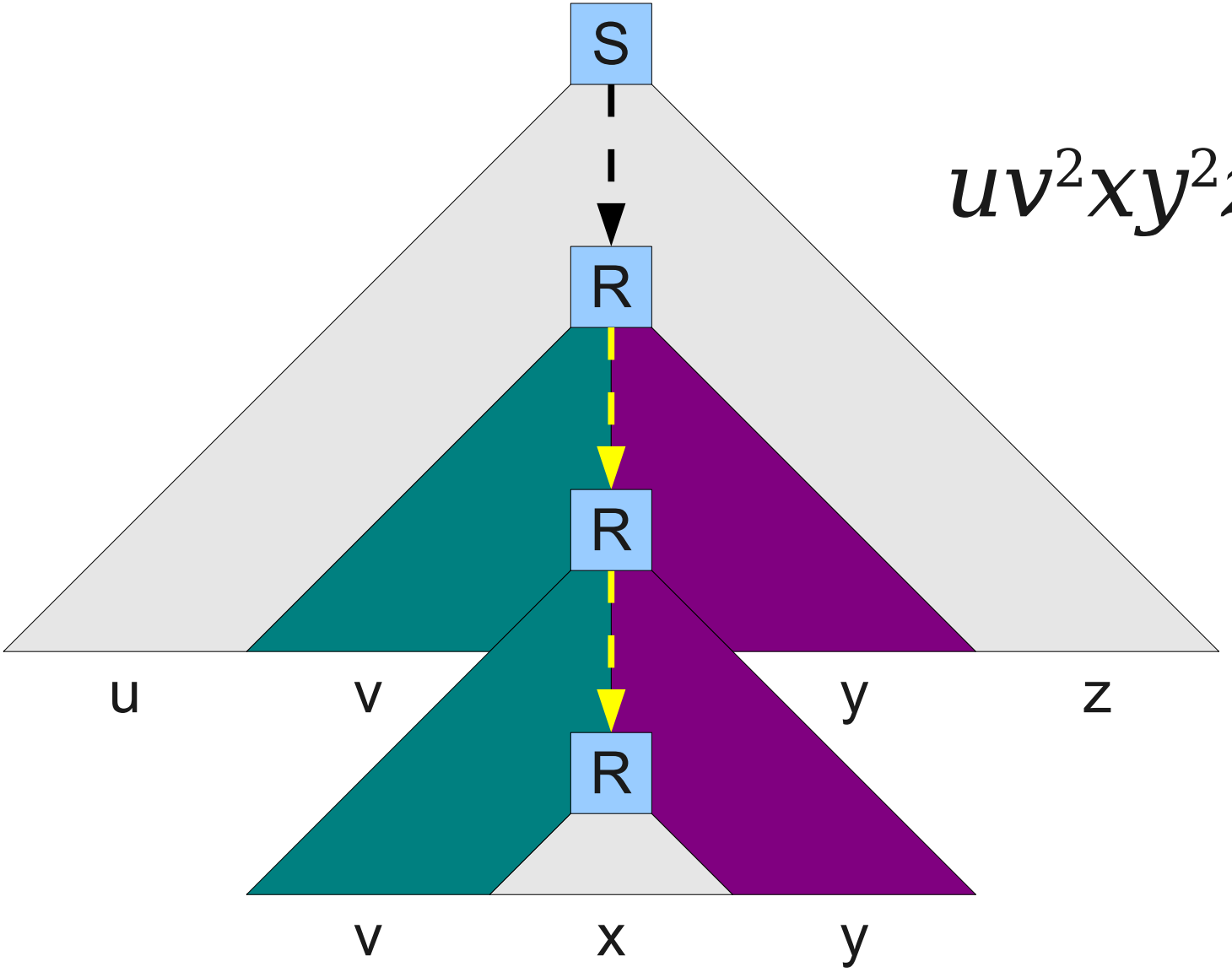
Parse Trees Revisited

S → **[P]**

P → **RR** | **a**

R → **(P)** | **b**





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

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,

$|vxy| \leq n$, 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,

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

$w = uvxyz$, 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

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,

$w = uvxyz$, w can be bro

$|vxy| \leq n$, 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 pumping length is not simple; see Sipser for details.

The Pumping Lemma Game

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

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: $L = \{w \in \{\texttt{0},\texttt{1},\texttt{2}\}^* \mid w$ has the same # of `0`s, `1`s, `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 $w = \texttt{0}^n\texttt{1}^n\texttt{2}^n$. 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 $\texttt{0}^n$, $\texttt{1}^n$, or $\texttt{2}^n$. 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 `0`s and `1`s or of `1`s and `2`s (it cannot consist of all three symbols, because $|vxy| \leq n$). Then if vxy has no `2`s in it, $uv^2xy^2z \notin L$ since it contains more `0`s or `1`s than `2`s. Similarly, if vxy has no `0`s in it $uv^2xy^2z \notin L$ because it contains more `1`s or `2`s than `0`s.

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.