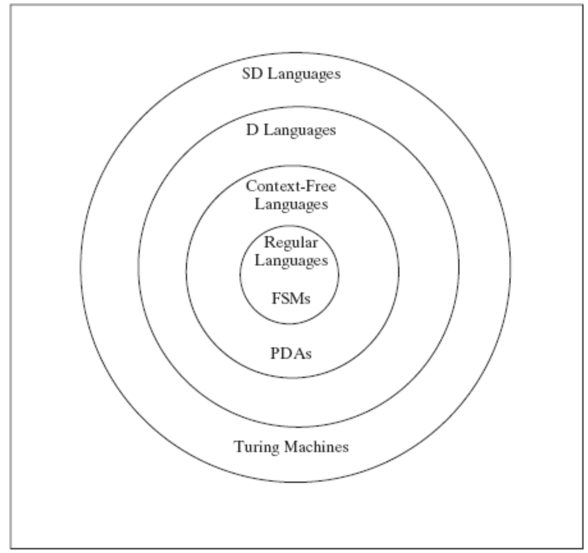
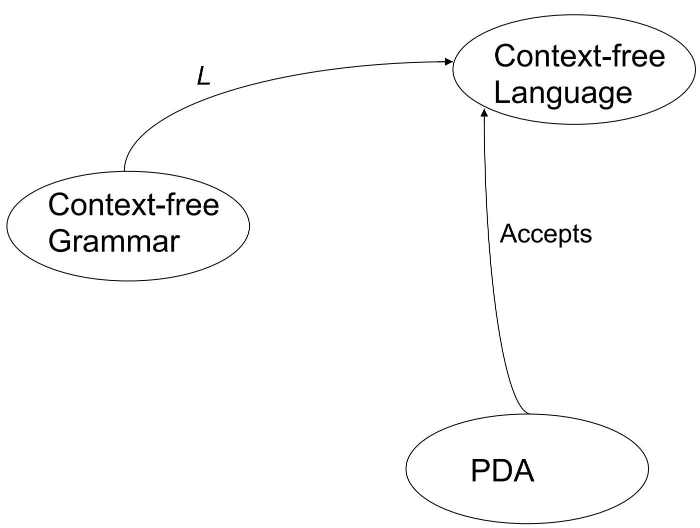
Context-Free Grammars

Chapter 11

Languages and Machines



Context-free Grammars, Languages, and PDAs



Context-Free Grammars

A context-free grammar G is a quadruple,
(V, Σ, R, S), where:

- V is the rule alphabet
  - Σ, a subset of V, the set of terminals
  - $V - \Sigma$, the set of nonterminals
- R, a finite subset of $(V - \Sigma) \times V^*$, the set of rules
- S, an element of $V - \Sigma$, the start symbol

Example:

$(\{S, \mathrm{a}, \mathrm{b}\}, \{\mathrm{a}, \mathrm{b}\}, \{S \rightarrow \mathrm{a}\, S\, \mathrm{b}, S \rightarrow \varepsilon\}, S)$

Derivations

$x \Rightarrow_G y$ iff $x = \alpha\, A\, \beta$

and $A \rightarrow \gamma$ is in R

$y = \alpha\, \gamma\, \beta$

$w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \ldots \Rightarrow_G w_n$ is a derivation in G.

Let $\Rightarrow_G{}^*$ be the reflexive, transitive closure of $\Rightarrow_G$.

Then the language generated by G, denoted $L(G)$, is:

$L(G) = \{w \in \Sigma^* : S \Rightarrow_G{}^* w\}$.

An Example Derivation

Example:

Let $G = (\{S, \mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}, \{S \rightarrow \mathtt{a}\, S\, \mathtt{b}, S \rightarrow \varepsilon\}, S)$

$S \Rightarrow \mathtt{a}\, S\, \mathtt{b} \Rightarrow \mathtt{aa}\, S\, \mathtt{bb} \Rightarrow \mathtt{aaa}\, S\, \mathtt{bbb} \Rightarrow \mathtt{aaabbb}$

$S \Rightarrow^* \mathtt{aaabbb}$

Definition of a Context-Free Grammar

A language L is ***context-free*** iff it is generated by some context-free grammar G.

A^nB^n

$S \rightarrow \varepsilon$
$S \rightarrow \mathtt{a}S\mathtt{b}$

Balanced Parentheses

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

Recursive Grammar Rules

- A rule is ***recursive*** iff it is $X \rightarrow w_1 Y w_2$, where:
  $Y \Rightarrow^* w_3 X w_4$ for some w_1, w_2, w_3, and w in V^*.
- A grammar is recursive iff it contains at least one recursive rule.
- Examples: $S \rightarrow (S)$ $\quad\quad$ $S \rightarrow (T)$
  $T \rightarrow (S)$

Self-Embedding Grammar Rules

- A rule in a grammar G is ***self-embedding*** iff it is :
  $X \rightarrow w_1 Y w_2$, where $Y \Rightarrow^* w_3 X w_4$ and
  both $w_1 w_3$ and $w_4 w_2$ are in Σ^+.
- A grammar is self-embedding iff it contains at least one self-embedding rule.
- Example: $S \rightarrow \mathrm{a}S\mathrm{a}$ is self-embedding
  $S \rightarrow \mathrm{a}S$ is recursive but not self-embedding
  $S \rightarrow \mathrm{a}T$
  $T \rightarrow S\mathrm{a}$ is self-embedding

Where Context-Free Grammars Get Their Power

- If a grammar G is not self-embedding then $L(G)$ is regular.
- If a language L has the property that every grammar that defines it is self-embedding, then L is not regular.

PalEven = $\{ww^R : w \in \{\mathtt{a}, \mathtt{b}\}^*\}$

$G = \{\{S, \mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}, R, S\}$, where:

$$R = \{\; S \to \mathtt{a}S\mathtt{a} \\ S \to \mathtt{b}S\mathtt{b} \\ S \to \varepsilon \;\}.$$

Equal Numbers of a's and b's

Let $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w)\}$.

$G = \{\{S, \mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}, R, S\}$, where:

$$R = \{\; S \to \mathtt{a}S\mathtt{b} \\ S \to \mathtt{b}S\mathtt{a} \\ S \to SS \\ S \to \varepsilon \;\}.$$

Arithmetic Expressions

$G = (V, \Sigma, R, E)$, where

$V = \{+, *, (,), \mathtt{id}, E\}$,

$\Sigma = \{+, *, (,), \mathtt{id}\}$,

$R = \{$

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to \mathtt{id}\ \}$

BNF

A notation for writing practical context-free grammars

- The symbol | should be read as "or".

  Example: $S \to \mathtt{a}S\mathtt{b} \mid \mathtt{b}S\mathtt{a} \mid SS \mid \varepsilon$

- Allow a nonterminal symbol to be any sequence of characters surrounded by angle brackets.

  Examples of nonterminals:

  <program>
  <variable>

BNF for a Java Fragment

```
<block> ::= {<stmt-list>} | {}
<stmt-list> ::= <stmt> | <stmt-list> <stmt>
<stmt> ::= <block> | while (<cond>) <stmt> |
           if (<cond>) <stmt> |
           do <stmt> while (<cond>); |
           <assignment-stmt>; |
           return | return <expression> |
           <method-invocation>;
```

HTML

```
<ul>
    <li>Item 1, which will include a sublist</li>
        <ul>
            <li>First item in sublist</li>
            <li>Second item in sublist</li>
        </ul>
    <li>Item 2</li>
</ul>
```

A grammar:

HTMLtext → *Element* *HTMLtext* | ε

Element → *UL* | *LI* | … (and other kinds of elements that are allowed in the body of an HTML document)

UL → *HTMLtext*

LI → *HTMLtext*

English

S → *NP VP*

NP → the *Nominal* | a *Nominal* | *Nominal* |
ProperNoun | *NP PP*

Nominal → *N* | *Adjs N*

N → cat | dogs | bear | girl | chocolate | rifle

ProperNoun → Chris | Fluffy

Adjs → *Adj Adjs* | *Adj*

Adj → young | older | smart

VP → *V* | *V NP* | *VP PP*

V → like | likes | thinks | shots | smells

PP → *Prep NP*

Prep → with

Designing Context-Free Grammars

- Generate related regions together.

  A^nB^n

- Generate concatenated regions:

  A → *BC*

- Generate outside in:

  A → a*A*b

Concatenating Independent Languages

Let $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m : n, m \geq 0\}$.

The $\mathtt{c}^m$ portion of any string in L is completely independent of the $\mathtt{a}^n\mathtt{b}^n$ portion, so we should generate the two portions separately and concatenate them together.

$G = (\{S, N, C, \mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, R, S\}$ where:

$R = \{ S \rightarrow NC$
$N \rightarrow \mathtt{a}N\mathtt{b}$
$N \rightarrow \varepsilon$
$C \rightarrow \mathtt{c}C$
$C \rightarrow \varepsilon \}$.

$L = \{ a^{n_1}b^{n_1}a^{n_2}b^{n_2}...a^{n_k}b^{n_k} : k \geq 0 \text{ and } \forall i\ (n_i \geq 0)\}$

Examples of strings in L: ε, `abab`, `aabbaaabbbabab`

Note that $L = \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}^*$.

$G = (\{S, M, \mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}, R, S\}$ where:

$R = \{ S \rightarrow MS$
$S \rightarrow \varepsilon$
$M \rightarrow \mathtt{a}M\mathtt{b}$
$M \rightarrow \varepsilon\}$.

Unequal a's and b's

$L = \{\mathtt{a}^n\mathtt{b}^m : n \neq m\}$

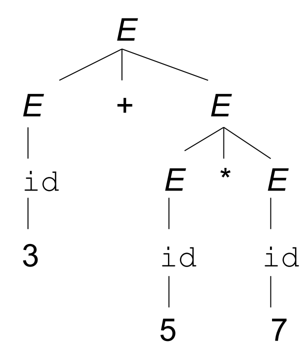
$G = (V, \Sigma, R, S)$, where
$V = \{\mathtt{a}, \mathtt{b}, S, A, B\}$,
$\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
$R =$

$S \rightarrow A$	/* more `a`'s than `b`'s
$S \rightarrow B$	/* more `b`'s than `a`'s
$A \rightarrow \mathtt{a}$	/* at least one extra `a` generated
$A \rightarrow \mathtt{a}A$	
$A \rightarrow \mathtt{a}A\mathtt{b}$	
$B \rightarrow \mathtt{b}$	/* at least one extra `b` generated
$B \rightarrow B\mathtt{b}$	
$B \rightarrow \mathtt{a}B\mathtt{b}$	

Structure

Context free languages:

We care about structure.



Derivations

To capture structure, we must capture the path we took through the grammar. **Derivations** do that.

Example:

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

1 2 3 4 5 6

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

1 2 3 5 4 6

But the order of rule application doesn't matter.

Derivations

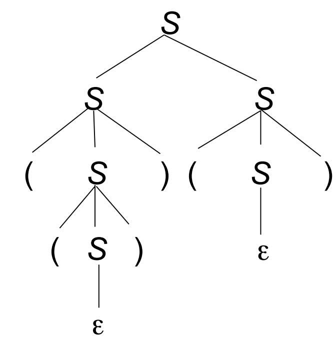
Parse trees capture essential structure:

1 2 3 4 5 6

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

1 2 3 5 4 6



Parse Trees

A **parse tree**, derived by a grammar $G = (V, \Sigma, R, S)$, is a rooted, ordered tree in which:

- Every leaf node is labeled with an element of $\Sigma \cup \{\varepsilon\}$,
- The root node is labeled S,
- Every other node is labeled with some element of: $V - \Sigma$, and
- If m is a nonleaf node labeled X and the children of m are labeled $x_1, x_2, \ldots, x_n$, then R contains the rule $X \rightarrow x_1, x_2, \ldots, x_n$.

Ambiguity

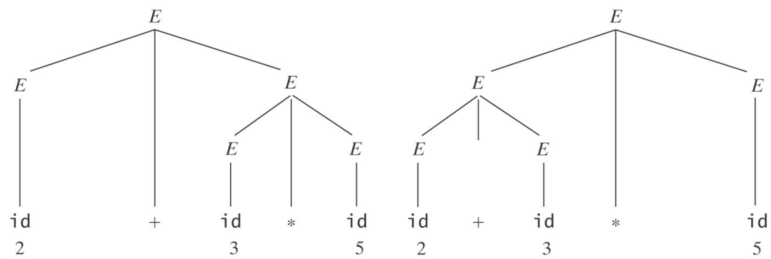
A grammar is ***ambiguous*** iff there is at least one string in $L(G)$ for which G produces more than one parse tree.

For most applications of context-free grammars, this is a problem.

An Arithmetic Expression Grammar

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

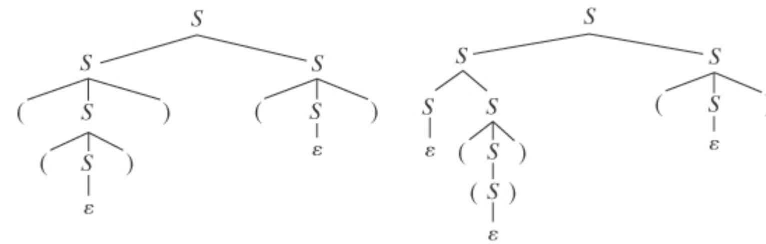
`id + id * id`



Even a Very Simple Grammar Can be Highly Ambiguous

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

(())()



Inherent Ambiguity

Some languages have the property that every grammar for them is ambiguous. We call such languages ***inherently ambiguous***.

Example:

$L = \{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}$.

It can be proved that L is inherently ambiguous.

We can generate $a^n b^n c^m$ and $a^n b^m c^m$ unambiguously but $a^n b^n c^n$ will be generated in two ways.

Inherent Ambiguity

Both of the following problems are undecidable:

- Given a context-free grammar G, is G ambiguous?
- Given a context-free language L, is L inherently ambiguous?

But We Can Often Reduce Ambiguity

We can get rid of:

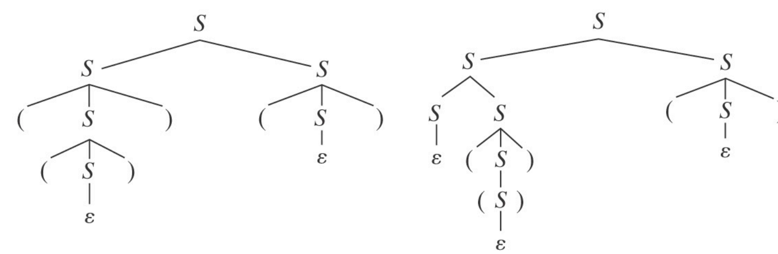
- ε rules like $S \rightarrow \varepsilon$,
- rules with symmetric right-hand sides, e.g.,

  $S \rightarrow SS$
  $E \rightarrow E + E$

- rule sets that lead to ambiguous attachment of optional postfixes.

A Highly Ambiguous Grammar

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



Resolving the Ambiguity with a Different Grammar

The biggest problem is the ε rule.

A different grammar for the language of balanced parentheses:

$S^* \rightarrow \varepsilon$
$S^* \rightarrow S$
$S \rightarrow SS$
$S \rightarrow (S)$
$S \rightarrow ()$

Nullable Variables

A variable X is ***nullable*** iff either:
(1) there is a rule $X \rightarrow \varepsilon$, or
(2) there is a rule $X \rightarrow PQR\ldots$ and P, Q, R, … are all nullable.

So compute N, the set of nullable variables, as follows:

1. Set N to the set of variables that satisfy (1).
2. Repeat until no change
   Add variables satisfying (2)

A General Technique for Getting Rid of ε-Rules

Definition: a rule is ***modifiable*** iff it is of the form:

$P \rightarrow \alpha Q \beta$, for some nullable Q, $P \neq \alpha\beta \neq \varepsilon$

removeEps(G: cfg) =

1. Let G' = G.
2. Find the set N of nullable variables in G'.
3. For each modifiable rule $P \rightarrow \alpha Q \beta$ of G do
   Add the rule $P \rightarrow \alpha\beta$.
4. Delete from G' all rules of the form $X \rightarrow \varepsilon$.
5. Return G'.

$L(G') = L(G) - \{\varepsilon\}$

An Example

$G = (\{S, T, A, B, C, \mathrm{a}, \mathrm{b}, \mathrm{c}\}, \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}, R, S)$, R =

$\{ S \rightarrow \mathrm{a}T\mathrm{a}$
$T \rightarrow ABC$
$A \rightarrow \mathrm{a}A \mid C$
$B \rightarrow B\mathrm{b} \mid C$
$C \rightarrow \mathrm{c} \mid \varepsilon \}$

Nullable varibles = {*A, B, C, T*}

G′: add : remove:

S → aa	T → AC	$C \rightarrow \varepsilon$
T → A	T → BC	
T → B B → b		
T → C A → a		
T → AB		

What If $\varepsilon \in L$?

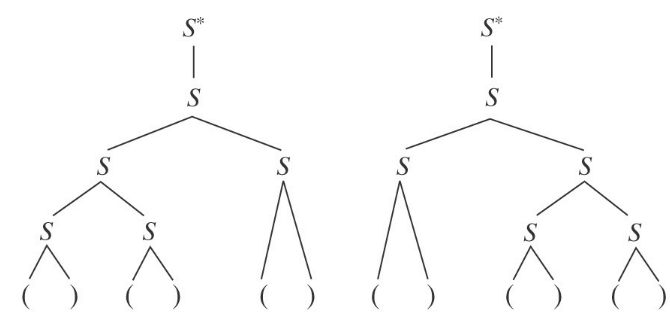
atmostoneEps(G: cfg) =

1. G'' = *removeEps*(G).
2. If S_G is nullable then /* i. e., $\varepsilon \in L(G)$
   2.1 Create in G'' a new start symbol S^*.
   2.2 Add to $R_{G''}$ the two rules:
   $S^* \rightarrow \varepsilon$
   $S^* \rightarrow S_G$.
3. Return G''.

But There is Still Ambiguity

$S^* \rightarrow \varepsilon$
$S^* \rightarrow S$
$S \rightarrow SS$
$S \rightarrow (S)$
$S \rightarrow ()$

What about ()()() ?



Eliminating Symmetric Recursive Rules

$S^* \rightarrow \varepsilon$
$S^* \rightarrow S$
$S \rightarrow SS$
$S \rightarrow (S)$
$S \rightarrow ()$

Replace $S \rightarrow SS$ with one of:

$S \rightarrow SS_1$ /* force branching to the left
$S \rightarrow S_1S$ /* force branching to the right

So we get:

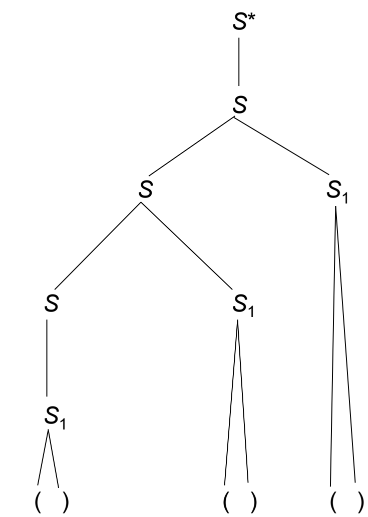
$S^* \rightarrow \varepsilon$
$S^* \rightarrow S$

$S \rightarrow SS_1$
$S \rightarrow S_1$
$S_1 \rightarrow (S)$
$S_1 \rightarrow ()$

Eliminating Symmetric Recursive Rules

So we get:

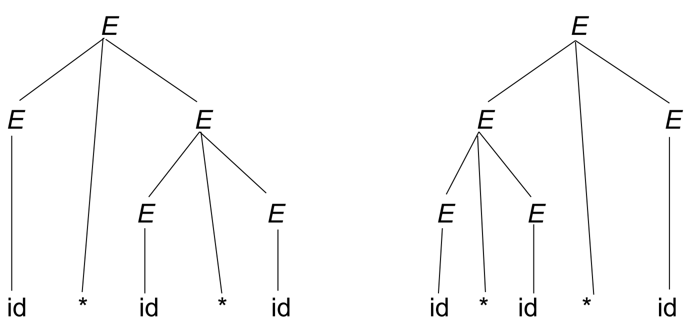
$S^* \rightarrow \varepsilon$
$S^* \rightarrow S$
$S \rightarrow SS_1$
$S \rightarrow S_1$
$S_1 \rightarrow (S)$
$S_1 \rightarrow ()$



Arithmetic Expressions

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

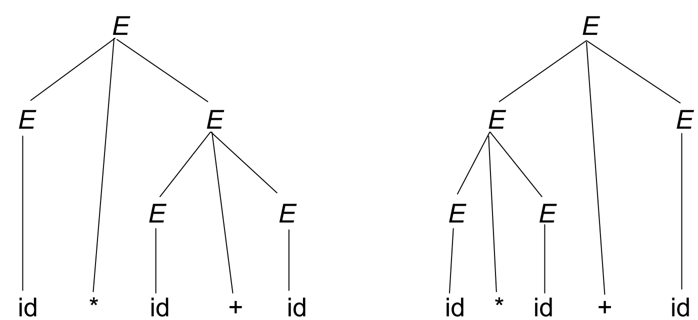
Problem 1: Associativity



Arithmetic Expressions

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

Problem 2: Precedence

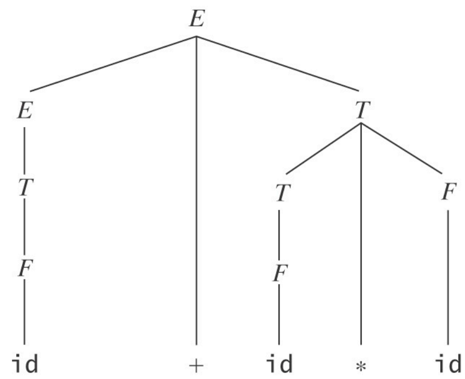


Arithmetic Expressions - A Better Way

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \text{id}$

Example:

`id + id * id`



Dangling “else”

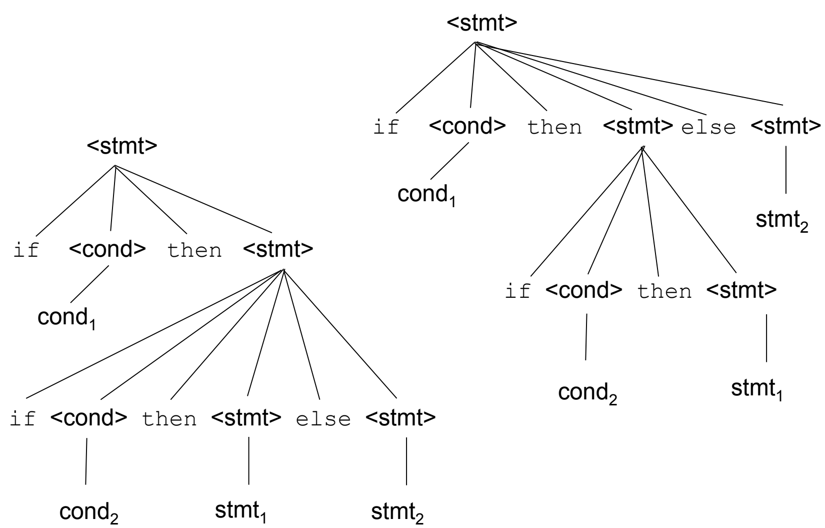
The dangling else problem:

<stmt> ::= `if` <cond> `then` <stmt>
<stmt> ::= `if` <cond> `then` <stmt> `else` <stmt>

Consider:

`if` cond_1 `then` `if` cond_2 `then` stmt_1 `else` stmt_2

Dangling “else” ambiguity



Dangling “else” solution

<stmt> ::= <matched_if>
| <unmatched_if>
<matched_if> ::= `if` <cond> `then` <matched_if> `else` <matched_if>
| <other_stmt>
<unmatched_if> ::= `if` <cond> `then` <stmt>
| `if` <cond> `then` <matched_if> `else` <unmatched_if>

Dangling “else” solution