Parsing

Part II

Writing Grammars

When writing a grammar (or RE) for some language, the following must be true:

1. All strings generated are in the language.
2. Your grammar produces all strings in the language.

Example:

$S \rightarrow (S)\ S \mid \varepsilon$

Generates all strings of balanced parentheses

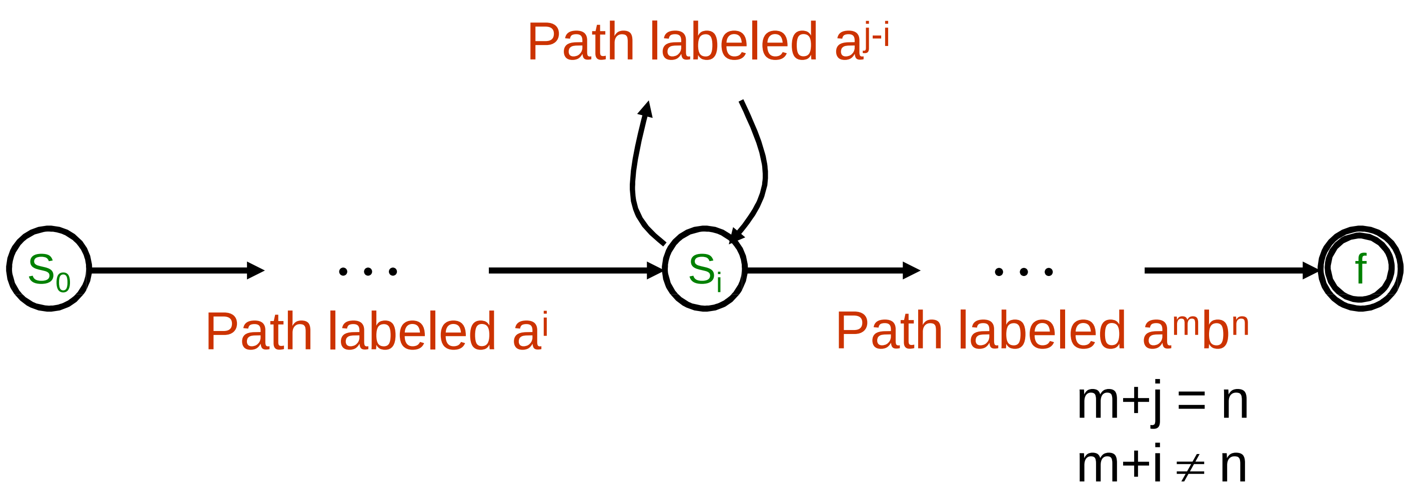
Using induction show that

Every sentence derivable from S is balanced

Every balanced string is derivable from S

- L={ $a^n b^n$ | $n \geq 1$}

- Show that L can be described by a grammar not by a regular expression
- Construct a DFA D with k states to accept L
- For $a^n b^n$ ($n>k$) some state (s_i) of D must be entered twice



Elimination of Ambiguity

Ambiguous Grammar

- A Grammar is ambiguous if there are multiple parse trees for the same sentence
- For the most parsers, the grammar must be unambiguous

Unambiguous grammar

unique selection of the parse tree for a sentence

Disambiguation

- Express Preference for one parse tree over others
  - Add disambiguating rule into the grammar

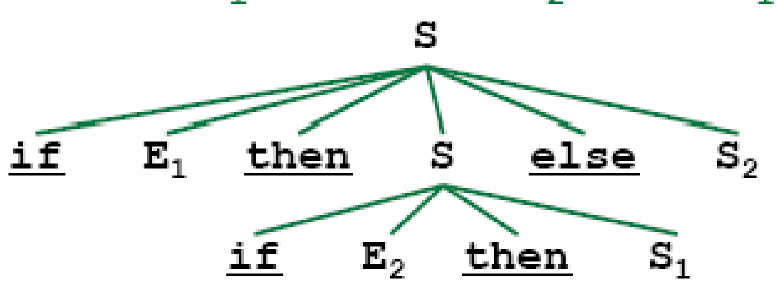
Dangling-else grammar

This grammar is ambiguous!

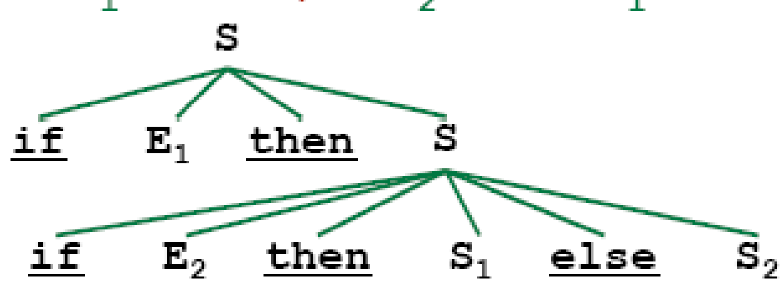
Stmt → **if** Expr **then** Stmt
→ **if** Expr **then** Stmt **else** Stmt
→ ...Other Stmt Forms...

Example String: `if` E_1 `then if` E_2 `then` S_1 `else` S_2

Interpretation #1: `if` E_1 `then (if` E_2 `then` S_1`) else` S_2



Interpretation #2: `if` E_1 `then (if` E_2 `then` S_1 `else` S_2`)`



Stmt → if Expr then Stmt
→ if Expr then Stmt else Stmt
→ ...Other Stmt Forms...

- In most of the programming languages, the ‘else’ is always attached with the innermost ‘if’
- Here, the else Stmt is always part of the innermost if

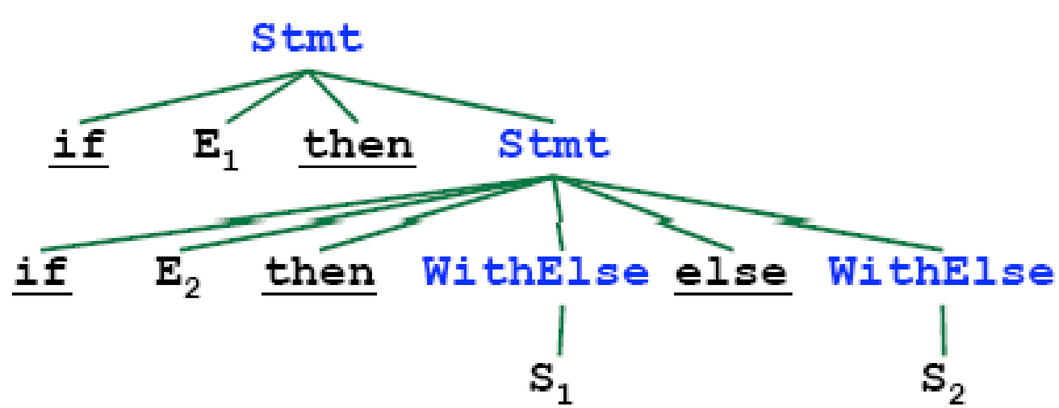
Dangling-else grammar

Goal: "Match `else`-clause to the closest `if` without an `else`-clause already."
Solution:

Stmt	→	`if` Expr `then` Stmt
	→	`if` Expr `then` WithElse `else` Stmt
	→	...Other Stmt Forms...
WithElse	→	`if` Expr `then` WithElse `else` WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between `then` and `else` must have an `else`.
i.e., the Stmt must not end with "`then` Stmt".

Interpretation #2: `if` E_1 `then` (`if` E_2 `then` S_1 `else` S_2)



Dangling-else grammar

Goal: "Match **else**-clause to the closest **if** without an **else**-clause already."

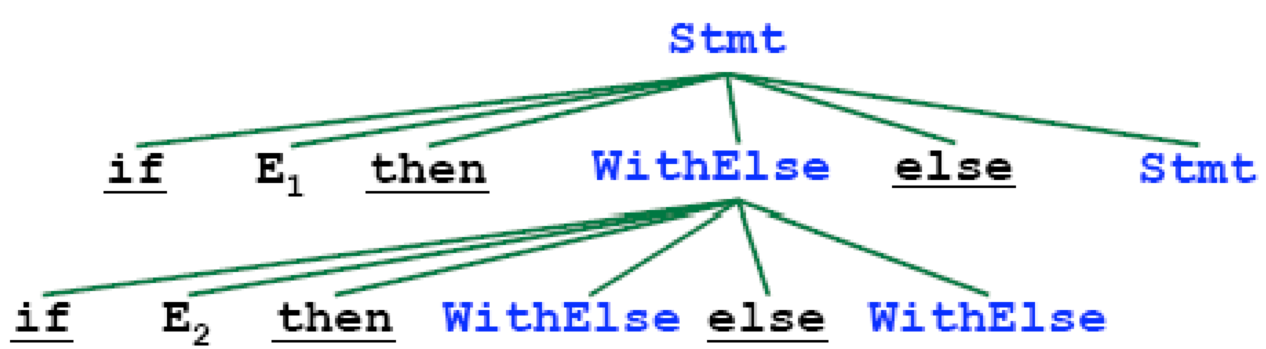
Solution:

Stmt → **if** Expr **then** Stmt
→ **if** Expr **then** WithElse **else** Stmt
→ ...Other Stmt Forms...

WithElse → **if** Expr **then** WithElse **else** WithElse
→ ...Other Stmt Forms...

Any Stmt occurring between **then** and **else** must have an **else**.
i.e., the Stmt must not end with "**then** Stmt".

Interpretation #1: **if** E_1 **then** (**if** E_2 **then** S_1) **else** S_2



Left Recursion

$S \rightarrow Aa$
$A \rightarrow Sb$
$S \rightarrow Sba$
Considering $ba = \alpha$

Whenever

$$A \Rightarrow^{+} A\alpha$$

Simplest Case: Immediate Left Recursion

Left Recursion if a problem for TDP

Given:

$$A \rightarrow A\alpha \mid \beta$$

Transform into:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

where A' is a new nonterminal

More General (but still immediate):

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$$

Transform into:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \varepsilon$$

Immediate Left Recursion Elimination: example

- Grammar

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$ ***Immediate Left Recursion***

  $F \rightarrow (E) \mid \textbf{id}$

- **Left recursion Eliminated**

  $E \rightarrow T E'$

  $E' \rightarrow + T E' \mid \varepsilon$

  $T \rightarrow F T'$

  $T' \rightarrow * F T' \mid \varepsilon$

  $F \rightarrow (E) \mid \textbf{id}$

Left Recursion in More Than One Step

Example:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid e$

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. $S \Rightarrow Af \Rightarrow Sdf$

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

$A \rightarrow Sd$

Get rid of the S. Substitute in the righthand sides of S.

$A \rightarrow Afd \mid bd$

The modified grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Afd \mid bd \mid e$

Now eliminate immediate left recursion involving A.

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid eA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow A\underline{c} \mid S\underline{d} \mid B\underline{e}$

$B \rightarrow A\underline{g} \mid S\underline{h} \mid \underline{k}$

So Far:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow \underline{bd}A' \mid B\underline{e}A'$

$A' \rightarrow \underline{c}A' \mid \underline{fd}A' \mid \varepsilon$

Left Recursion in More Than One Step

The Original Grammar:

$$S \rightarrow Af \mid b$$
$$A \rightarrow Ac \mid Sd \mid Be$$
$$B \rightarrow Ag \mid Sh \mid k$$

So Far:

$$S \rightarrow Af \mid b$$
$$A \rightarrow bdA' \mid BeA'$$
$$A' \rightarrow cA' \mid fdA' \mid \varepsilon$$
$$B \rightarrow Ag \mid Sh \mid k$$

Look at the B rules next; Does any righthand side start with “S”?

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → Ag | Afh | bh | k

Substitute, using the rules for "S"
Af... | b...

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow A\underline{c} \mid S\underline{d} \mid B\underline{e}$

$B \rightarrow A\underline{g} \mid S\underline{h} \mid \underline{k}$

So Far:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow \underline{bd}A' \mid B\underline{e}A'$

$A' \rightarrow \underline{c}A' \mid \underline{fd}A' \mid \varepsilon$

$B \rightarrow A\underline{g} \mid A\underline{fh} \mid \underline{bh} \mid \underline{k}$

Does any righthand side start with "A"?

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | Afh | bh | k

Substitute, using the rules for "A"
bdA'... | BeA'...

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | bdA'fh | BeA'fh | bh | k

Substitute, using the rules for "A"
bdA'... | BeA'...

Left Recursion in More Than One Step

The Original Grammar:

$S \to Af \mid b$

$A \to Ac \mid Sd \mid Be$

$B \to Ag \mid Sh \mid k$

So Far:

$S \to Af \mid b$

$A \to bdA' \mid BeA'$

$A' \to cA' \mid fdA' \mid \varepsilon$

$B \to bdA'g \mid BeA'g \mid bdA'fh \mid BeA'fh \mid bh \mid k$

Finally, eliminate any immediate Left recursion involving "B"

Next Form

$S \to Af \mid b$

$A \to bdA' \mid BeA'$

$A' \to cA' \mid fdA' \mid \varepsilon$

$B \to bdA'gB' \mid bdA'fhB' \mid bhB' \mid kB'$

$B' \to eA'gB' \mid eA'fhB' \mid \varepsilon$

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b

A → Ac | Sd | Be | C

B → Ag | Sh | k

C → BkmA | AS | j

If there is another nonterminal, then do it next.

So Far:

S → Af | b

A → bdA' | BeA' | CA'

A' → cA' | fdA' | ε

B → bdA'gB' | bdA'fhB' | bhB' | kB' | CA'gB' | CA'fhB'

B' → eA'gB' | eA'fhB' | ε

Algorithm for Eliminating Left Recursion $O(n^2)$

Assume the nonterminals are ordered $A_1, A_2, A_3, \ldots$

(In the example: S, A, B)

<u>for</u> <u>each</u> nonterminal A_i (for i = 1 to N) <u>do</u>

  <u>for</u> <u>each</u> nonterminal A_j (for j = 1 to i-1) <u>do</u>

    Let $A_j \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_N$ be all the rules for A_j

    <u>if</u> there is a rule of the form

$$A_i \rightarrow A_j\alpha$$

    <u>then</u> replace it by

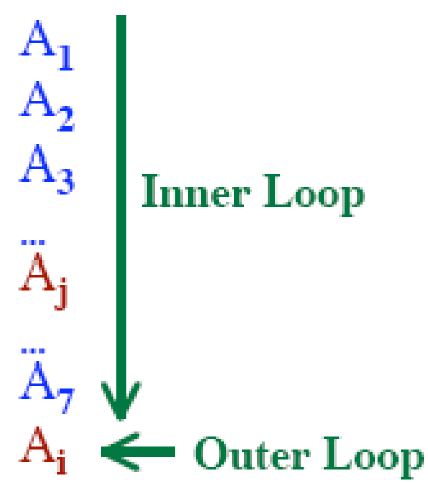
$$A_i \rightarrow \beta_1\alpha \mid \beta_2\alpha \mid \beta_3\alpha \mid \ldots \mid \beta_N\alpha$$

    <u>endIf</u>

  <u>endFor</u>

  Eliminate immediate left recursion among the A_i rules

<u>endFor</u>



Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-town parser

Problem:

Stmt → `if` Expr `then` Stmt `else` Stmt

→ `if` Expr `then` Stmt

→ OtherStmt

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

Stmt → `if` Expr `then` Stmt ElsePart

→ OtherStmt

ElsePart → `else` Stmt | ε

General Approach:

Before: A $\rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: A $\rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

C $\rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left Factoring

Problem:

$$\underbrace{\text{Stmt}}_{A} \to \underbrace{\underline{\texttt{if}}\ \text{Expr}\ \underline{\texttt{then}}\ \text{Stmt}}_{\alpha}\ \underbrace{\underline{\texttt{else}}\ \text{Stmt}}_{\beta_1}$$

$$\to \underbrace{\underline{\texttt{if}}\ \text{Expr}\ \underline{\texttt{then}}\ \text{Stmt}}_{\alpha}\ \underbrace{\varepsilon}_{\beta_2}$$

$$\to \underbrace{\text{OtherStmt}}_{\delta_1}$$

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

$$\underbrace{\text{Stmt}}_{A} \to \underbrace{\underline{\texttt{if}}\ \text{Expr}\ \underline{\texttt{then}}\ \text{Stmt}}_{\alpha}\ \underbrace{\text{ElsePart}}_{C}$$

$$\to \underbrace{\text{OtherStmt}}_{\delta_1}$$

$$\underbrace{\text{ElsePart}}_{C} \to \underbrace{\underline{\texttt{else}}\ \text{Stmt}}_{\beta_1} \mid \underbrace{\varepsilon}_{\beta_2}$$

General Approach:

Before: $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: $A \to \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

$C \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left Factoring Algorithm

- For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If α ≠ ε, then replace all of A-productions A→ $\alpha\beta_1 | \alpha\beta_2 \mid \alpha\beta_n \mid \gamma$ by
  - A → α A' | γ
  - A' → $\beta_1 \mid \beta_2 \mid \ldots . \mid \beta_n$
- Example:

  S → aSSbS / aSaSb / abb / b
- Modified grammar:

  S → aS' / b

  S' → SSbS / SaSb / bb

Again, this is a grammar with common prefixes.

Further modified:

S → aS' / b

S' → SA / bb

A → SbS / aSb