Compilers

by
Marwa Yusuf

Lecture 6
Wed. 1-3-2023

Chapter 4 (4.2 to 4.3)

Syntax Analysis

Context-Free Grammars

- Systemically describes the syntax of a programming language.
- Ex: *stmt* → **if (** *expr* **)** *stmt* **else** *stmt*
- Consists of:
  - ***Terminals*** (basic symbols, token names);
  - ***Non-terminals*** (syntactic variables denoting sets of strings, defining hierarchical structure);
  - ***Start symbol*** (the language generated by grammar);
  - ***Productions*** (head or left side, → or ::= and body or right side).
- Ex: terminals are:…... non-terminals:…….

$$expression \rightarrow expression + term \mid expression - term \mid term$$

$$term \rightarrow term * factor \mid term \mathbin{/} factor \mid factor$$

$$factor \rightarrow (expression) \mid \mathbf{id}$$

Context-Free Grammars (cont.)

- Conventions:

  1) Terminals: early lower case letters (*a*, *b*, *c*), operators (+, *), punctuation(;, parenthesis), digits (0, 1, 2), boldface strings (**id**, **num**).
  2) Non-terminals: early uppercase letters (*A*, *B*, *C*), *S* (start), italic strings (*expr*, *stmt*), uppercase letter when discussing constructs (*E*, *F*).
  3) Uppercase late letters: grammar symbols; terminal or non-terminal (*X*, *Y*, *Z*).
  4) Lowercase late letters: strings of terminals (including ε) (*u*, *v*, *w*).
  5) Lowercase Greek: strings of grammar symbols (α, β, γ). $A \rightarrow \alpha$.
  6) $A \rightarrow \alpha_1$

     $A \rightarrow \alpha_2$

     $A \rightarrow \alpha_3$

     can be transformed to

     $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$

     alternatives for a head.
  7) By default, the head of 1st production is the start symbol.

Derivations

- Using rewriting rules.
- Beginning with the start symbol, in each step replace a non-terminal with the body of one of its productions (corresponds to top-down construction of parse tree).
- Ex:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \textbf{id}$$

- If E denotes an expression, then – E also denotes an expression.
- Replacement of E by – E is denoted by

$$E \Rightarrow -E \text{ (}E\text{ derives }-E\text{)}$$

Derivations

- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$ (derivation of -(**id**) from *E*, -(**id**) is an instance of expression).
- Derivation definition:
  - given $\alpha A\beta$ and $A \rightarrow \gamma$ then $\alpha A\beta \Rightarrow \alpha\gamma\beta$
- $\Rightarrow$ derives in one step.
- $\Rightarrow^*$ derives in zero or more steps.
- $\Rightarrow^+$ derives in one or more steps.
- If $S \Rightarrow^* \alpha$, then α is called *sentential form of G.*
- A *sentence* of *G* is a sentential form with no non-terminals.
- The language generated by a grammar is its set of sentences.
- *w* is in *L(G)* iff *w* is a sentence of *G* (or $S \Rightarrow^* w$).
- A language generated from a grammar is a context free language.
- Equivalent grammars: generate the same language.

Derivations

- Ex: **-(id+id)** is a sentence of grammar

  $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$

  because :

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$
- All these internal strings are sentential forms of G.
- $E \Rightarrow^* -(\boldsymbol{id}+\boldsymbol{id})$
- Alternative derivation:

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$

Derivations

1) Leftmost derivations: $\alpha \Rightarrow_{\mathrm{lm}} \beta$, the leftmost non-terminal is replaced first.

2) Rightmost derivations: $\alpha \Rightarrow_{\mathrm{rm}} \beta$

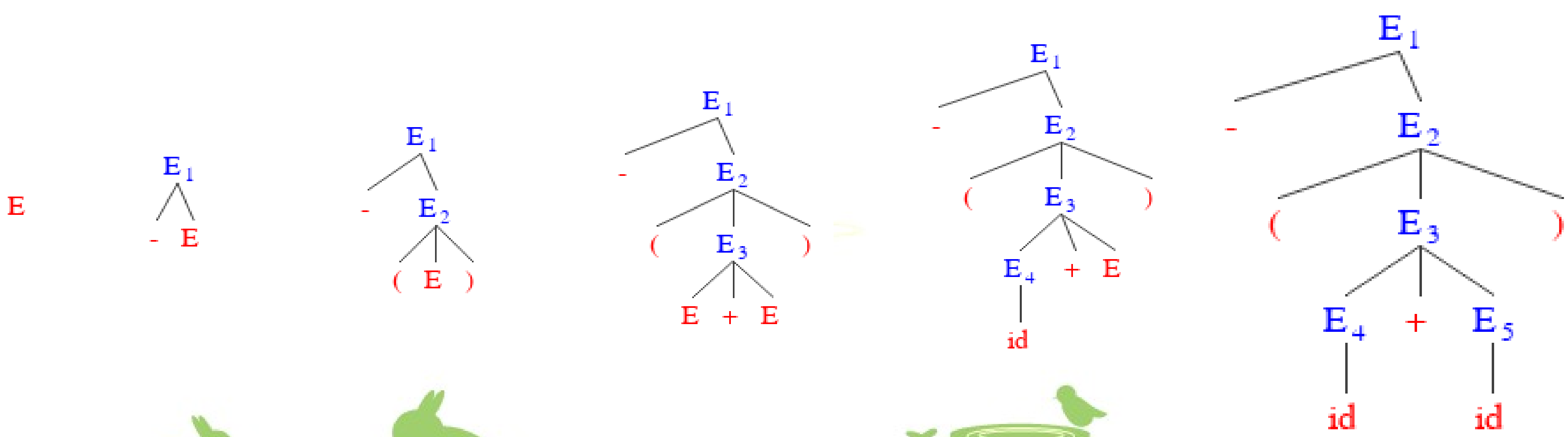
$$E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E+E) \Rightarrow_{lm} -(\mathbf{id}+E) \Rightarrow_{lm} -(\mathbf{id}+\mathbf{id})$$

$$\mathrm{E} \Rightarrow_{rm} -\mathrm{E} \Rightarrow_{rm} -(\mathrm{E}) \Rightarrow_{rm} -(\mathrm{E}+\mathrm{E}) \Rightarrow_{rm} -(\mathrm{E}+\mathbf{id}) \Rightarrow_{rm} -(\mathbf{id}+\mathbf{id})$$

- $S \Rightarrow_{lm}^{*} \alpha$ then α is a leftmost sentential form of G.

Parse Trees and Derivations

- A graphical representation of derivation filtering out ordering of derivations (many to one).
- Leaves read from left or right constitute a sentential form, called yield or frontier of the tree.



Ambiguity

- Grammar that produces more than one parse tree.
- Ex: E $\rightarrow$ $E + E \mid E * E \mid (E) \mid$ **id**

id + id * id

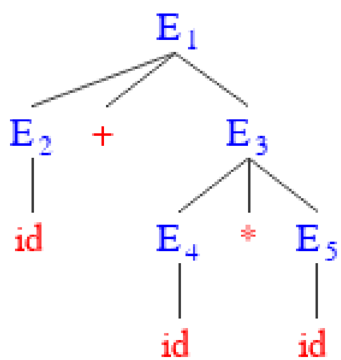
- E $\Rightarrow$ E+E

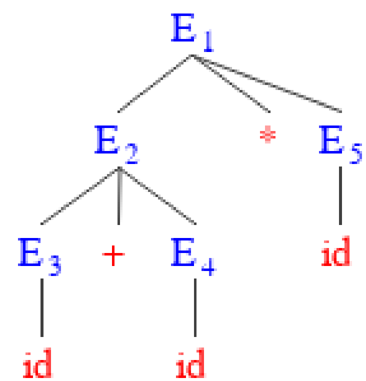
  $\Rightarrow$ id + E

  $\Rightarrow$ id + E * E

  $\Rightarrow$ id + id * E

  $\Rightarrow$ id + id * id





- E $\Rightarrow$ E*E

  $\Rightarrow$ E + E * E

  $\Rightarrow$ id + E * E

  $\Rightarrow$ id + id * E

  $\Rightarrow$ id + id * id

- For most parsers, it is desirable for the grammar to be unambiguous. However, can use carefully chosen ambiguous grammar with disambiguating rules to throw away unwanted parse trees.

Skipped

- Sections 4.2.6 & 4.2.7 are skipped.

Writing a Grammar

- Grammar can describe “most” of the syntax.
- Ex: Condition that **id** declared before use cannot be defined by grammar.
- Later steps after parser deal with such cases.

Lexical vs. Syntactic Analysis

- Section 4.3.1 for reading (MUST).

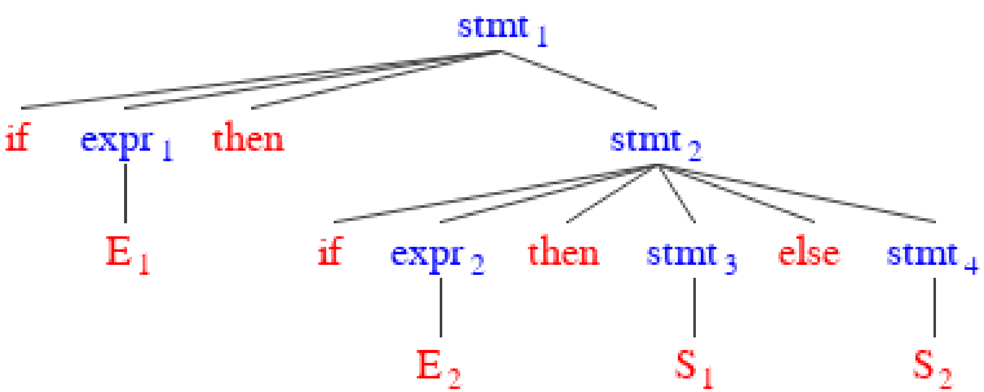
Eliminating Ambiguity

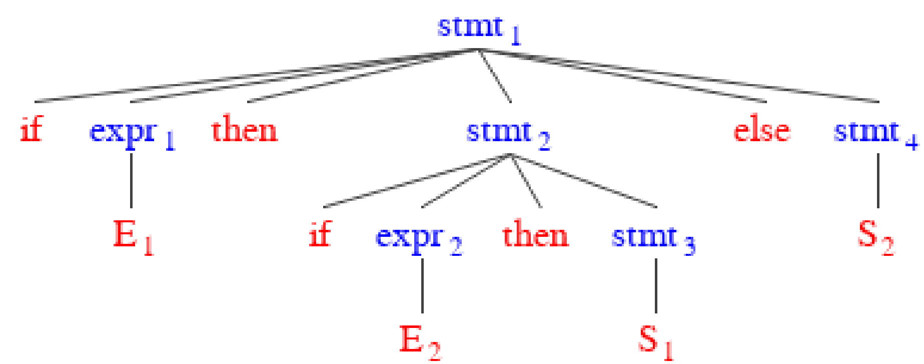
- *stmt* → **if** *expr* **then** *stmt*

  | **if** *expr* **then** *stmt* **else** *stmt*

  | **other**

- if E_1 then if E_2 then S_1 else S_2





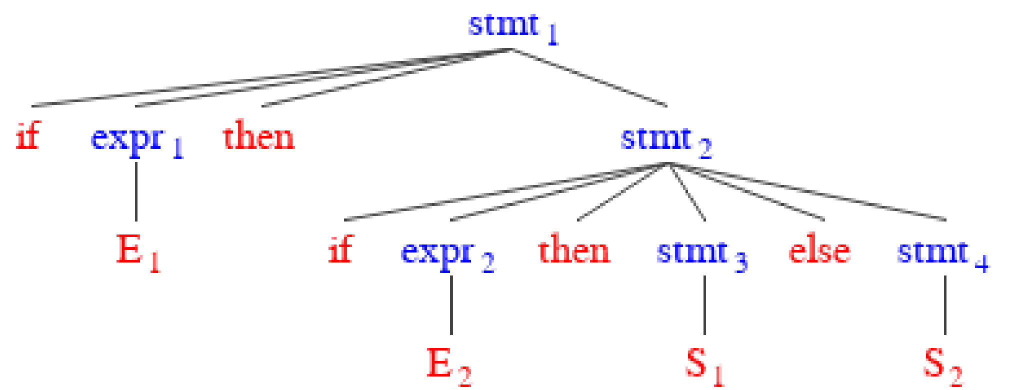
- 1st one is preferred, can be enforced by grammar, but usually not.

Eliminating Ambiguity

- *stmt* → *matched_stmt* | *open_stmt*
- *matched_stmt* → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**
- *open_stmt* → **if** *expr* **then** *stmt*

  | **if** *expr* **then** *matched_stmt* **else** *open_stmt*

- **The idea is that between then and else, there is always a matched statement, hence, any else is associated with the closest then.**
- if E_1 then if E_2 then S_1 else S_2



Elimination of Left Recursion

- Left recursive grammar if: $A \Rightarrow^+ A\alpha$
- Top-down parsing cannot handle left recursion.
- Immediate left recursion if $A \rightarrow A\alpha \mid \beta$
- Can be transformed to:

$$A \rightarrow \beta A'$$

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

- This rule is sufficient for many grammars.

Look at sec. 2.4.5 fig. 2.20

Elimination of Left Recursion

- Ex:

$E \rightarrow E + \mathrm{T}$

$E \rightarrow T$

$T \rightarrow T * \mathrm{F}$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$

$E \rightarrow T E'$

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

$T \rightarrow F T'$

$T' \rightarrow * \mathrm{F}\ T' \mid \varepsilon$

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

- **Given:** $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

  where no β_i begins with A

- **Then:** $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

  $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Elimination of Left Recursion

- Ex: $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \varepsilon$

  $S \Rightarrow Aa \Rightarrow Sda$

Elimination of Left Recursion Algorithm

- **INPUT:** Grammar G with no cycles($A \Rightarrow^+ A$) or ε-productions ($A \rightarrow \varepsilon$)
- **OUTPUT:** Equivalent Grammar with no left-recursion
- **METHOD:** Apply the following code to G. Note: resulting grammar may have ε-productions.

Elimination of Left Recursion Code

arrange the non-terminals in some order $A_1, A_2, \ldots, A_n$.
for (each i from 1 to n) {
    for (each j from 1 to $i - 1$) {
        replace each production of the form $A_i \rightarrow A_j \gamma$ by the
          productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$, where
          $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current A_j-productions
    }
    eliminate the immediate left recursion among the A_i-productions
}

Elimination of Left Recursion

- Ex: S → Aa | b

  A → Ac | Sd | ε   (ε is harmless in this case)

- For i = 1, no immediate left recursion, nothing happens.
- For i = 2, substitute for $S \rightarrow Aa \mid b$ to get

  $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

- Then, eliminate the immediate left-recursion

  $S \rightarrow Aa \mid b$

  $A \rightarrow bdA' \mid A'$

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

Left Factoring

- When it is not clear which to choose directly:
  - *stmt* → **if** *stmt* **then** *stmt* **else** *stmt*

    | **if** *stmt* **then** *stmt*
- Rewrite the production to defer the decision

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Left Factoring Algorithm

- **INPUT:** Grammar G
- **OUTPUT:** Equivalent left-factored grammar
- **METHOD:** For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_\mathrm{n} \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

- Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Left Factoring Example

- For dangling else problem:

$$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$$

$$E \rightarrow b$$

becomes:

$$S \rightarrow i\ E\ t\ S\ S' \mid a$$

$$S' \rightarrow e\ S \mid \varepsilon$$

$$E \rightarrow b$$

- Both are ambiguous

Non-Context-Free Language Constructs

- Skipped (Section 4.3.5)

Thanks!