SYNTAX ANALYSIS – Part I

Based on Chapter 4 of Aho, Lam, Sethi, Ullman:

Compilers: Principles, Techniques, & Tools

2nd Ed, Addison Wesley, 2007

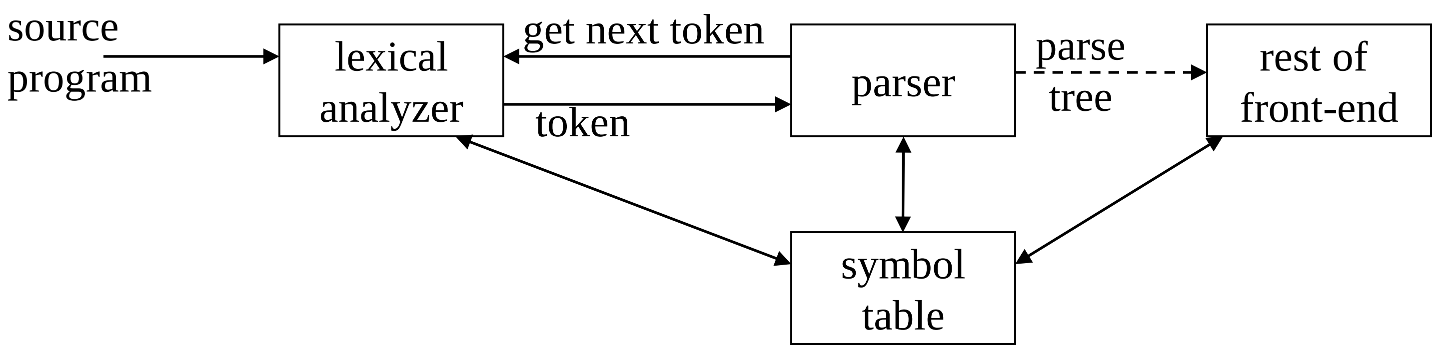
Table of Contents

Introduction

- Every programming language has rules that prescribe the syntactic structure of well-formed programs
- The syntax of programming language constructs can be described by context-free grammars
- Formal grammars
  - a precise and easy-to-understand syntactic specification
  - automatic construction of an efficient parser
  - syntax-directed translation
  - evolution of programming language
- Parsing
  - generating a parse tree
  - various LL and LR parsing techniques
  - parser generator: yacc(bison), JavaCC, etc.

The Role of Parser

- Interaction of lexical analyzer with parser



- Secondary tasks
  - reporting errors
  - recovery of errors

Context-Free Grammars

- Recursive Structure of High-level Language

  <stmt> → if <exp> then <stmt> else <stmt>

  - this form of statement cannot be specified using regular expressions
- Definition
  - a CFG G contains <T, N, S, P>
    - T : a set of terminal symbols (tokens)
    - N : a set of non-terminal symbols
    - S : a distinguished non-terminal symbol that denotes the language defined by the grammar
    - P : a set of production rules

      $A \rightarrow \alpha$ (A in N, α in $(T \cup N)^*$)

Context-Free Grammars

- Example: Arithmetic Expressions

expr -> *expr* + *term*

expr -> *expr* – *term*

expr -> *term*

term -> *term* ***** *factor*

term -> *term* **/** *factor*

term -> *factor*

factor -> **(** *expr* **)**

foctor -> **id**

expr, *term*, *factor*: non-terminal symbols
( ) - id + - * / : terminal symbols
expr : start symbol

- BNF(Backus Naur Form) and Algol60

Context-Free Grammars

- Notational Conventions
  - Terminal symbols
    - lower-case letters early in the alphabet: a, b, c
    - operator symbols: +, -, etc.
    - punctuation symbols: ( ) , etc.
    - digits: 0,1,…, 9
    - boldface strings such as **id** or **if**
  - Non-terminal symbols
    - upper-case letters early in the alphabet: A,B,C
    - the letter S (usually start symbol)
    - lower-case italic names such as *expr* or *stmt*
  - Upper-case letter late in the alphabet, X,Y, Z: grammar symbols
  - Lower-case letter late in the alphabet, u, v, …, z: strings of terminals
  - Lower-case Greek letters, α, β, γ : strings of grammar symbols

Derivations

- A derivation is viewed as the process by which a grammar defines a language
  - top-down construction of a parse tree

- General form (one-step derivation)

  $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production

  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ "α_1 derives α_n"

- Example

  E -> E + E | E * E | (E) | - E | **id**

  $E \Rightarrow - E$ "E derives -E"

  $E \Rightarrow - E \Rightarrow - (E) \Rightarrow - (\mathbf{id})$ "a derivation of - (**id**) from E

Derivations

- More notations

  $\overset{*}{\Rightarrow}$: "derives in zero or more steps"

  $\overset{+}{\Rightarrow}$: "derives in one or more steps"

  $\alpha \overset{*}{\Rightarrow} \alpha$

  $\alpha \overset{*}{\Rightarrow} \beta$ & $\beta \Rightarrow \gamma$, then $\alpha \overset{+}{\Rightarrow} \gamma$

- L(G), language generated by G

  $\{w \mid S \overset{+}{\Rightarrow} w,$ w is a string of terminals$\}$

- Sentential form

  $S \overset{*}{\Rightarrow} \alpha$: α is a sentential form of G

  $E \Rightarrow - E \Rightarrow - (E) \Rightarrow - (E+E \Rightarrow - (id + E) \Rightarrow -(id + id)$

- Rightmost and leftmost derivations
  - rightmost(leftmost) non-terminal is replaced at each step

    $E \Rightarrow - E \Rightarrow - (E) \Rightarrow - (E+E) \Rightarrow - (E + id) \Rightarrow - (id + id)$

- Leftmost(rightmost) sentential form $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$

Examples

- Example 1

  E -> (E) | a

  $L(G) = \{a, (a), ((a)), \ldots\} = \{ (^n a)^n \mid n \geq 0 \}$

  e.g. $E \Rightarrow (E) \Rightarrow ((E)) \Rightarrow ((a))$

  cf. E -> (E) → infinite recursion

- Example 2

  E -> E + a | a

  $L(G) = \{ a, a + a, a + a + a, \ldots \}$

  e.g. $E \Rightarrow E + a \Rightarrow E + a + a \Rightarrow E + a + a + a \Rightarrow a + a + a + a$

Examples

- Example 3: statements

statement -> *if-stmt* | **other**

if-stmt -> **if (** *exp* **)** *statement*

| **if (** *exp* **)** *statement* **else** *statement*

exp -> **0** | **1**

E.g. **if (1) other else if (0) other else other**

statement ⇒ if-stmt ⇒ **if (** exp **)** statement **else** statement

⇒**if (1)** statement **else** statement ⇒ **if (1) other else** statement

⇒**if (1) other else** if-stmt

⇒**if (1) other else if (** exp **)** statement **else** statement

⇒**if (1) other else if (0)** statement **else** statement

⇒**if (1) other else if (0) other else** statement

⇒**if (1) other else if (0) other else other**

Examples

- Example 4: Balanced parentheses

$A \rightarrow (A) A \mid \varepsilon$

e.g. $A \Rightarrow (A) A \Rightarrow ( ) A \Rightarrow ( ) (A) A \Rightarrow ( ) ((A) A) A$

$\Rightarrow ( ) (( ) A) A \Rightarrow ( ) (( )) A \Rightarrow ( ) (( ))$

or

$A \rightarrow AB \mid B$

$B \rightarrow (A) \mid ( )$

$A \Rightarrow AB \Rightarrow BB \Rightarrow ( ) B \Rightarrow ( ) (A) \Rightarrow ( ) (B) \Rightarrow ( ) (( ))$

Examples

- Example 5: statement sequence

  stmt-sequence -> *stmt* ; *stmt-sequence* | *stmt*

  stmt -> **s**

  L(G) = { s, s ; s, s ; s ; s, … }   ; as a separator

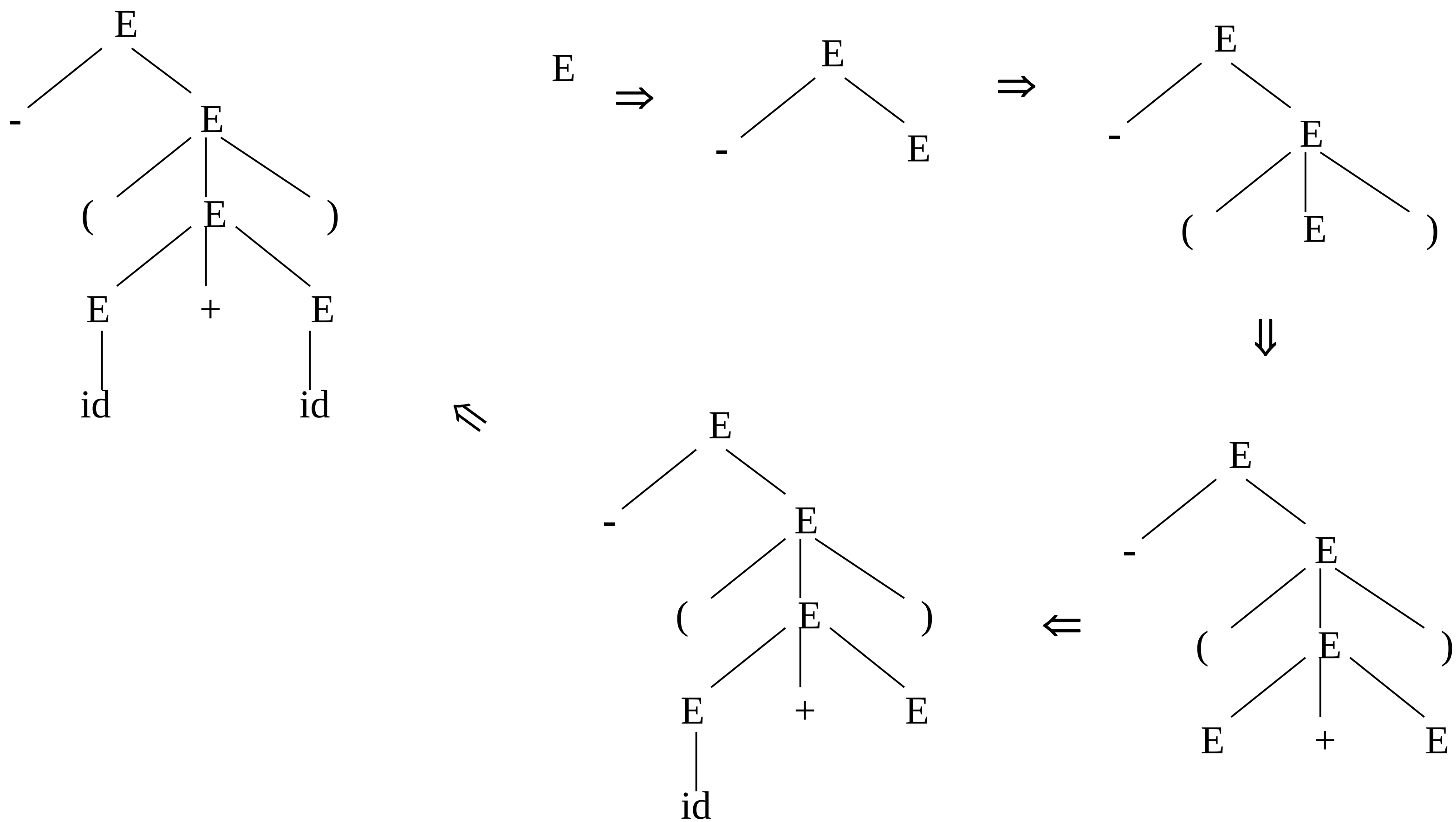
  stmt-sequence -> *stmt* ; *stmt-sequence* | ε

  stmt -> s

  L(G) = {ε , s ; , s ; s ; , s ; s ; s ; , …. }   ; as a terminator

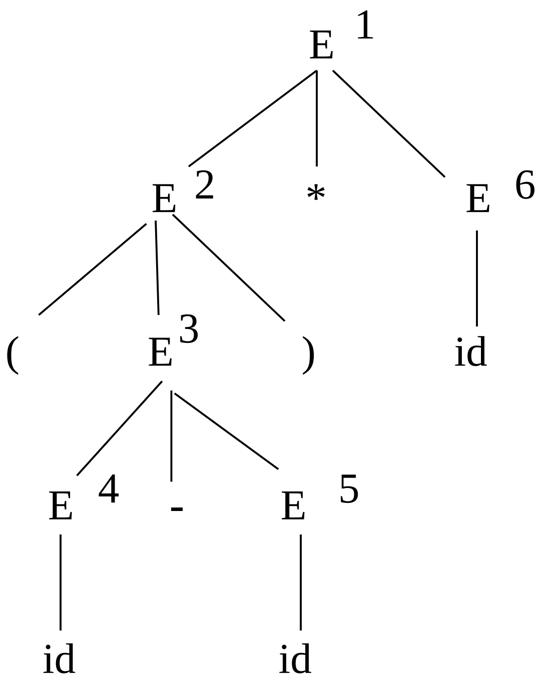
Parse Trees and Derivations

- A parse tree can be viewed as a graphical representation for a derivation without considering replacement order

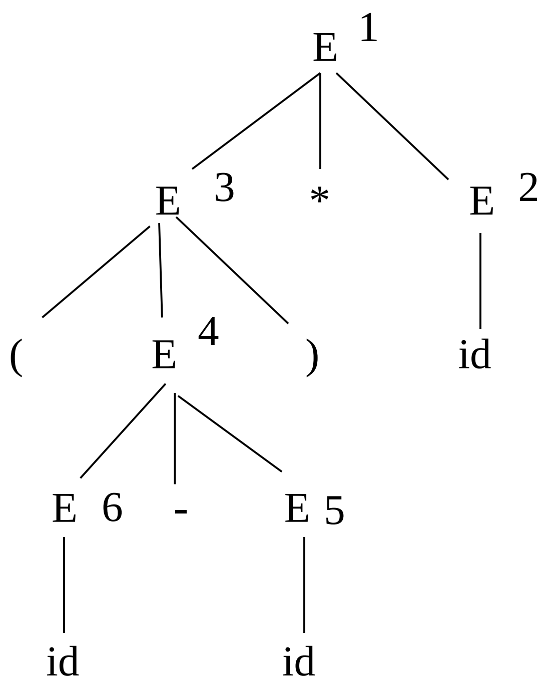


Parse Trees and Derivations

- Derivation and parse tree : (a – b) * c



leftmost derivation-
preorder numbering

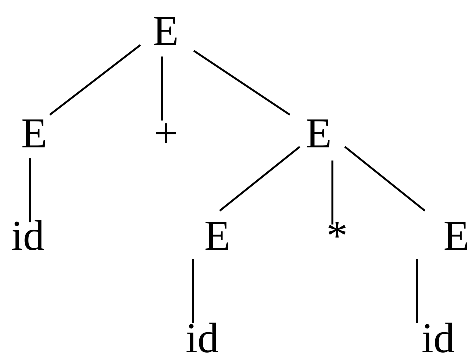


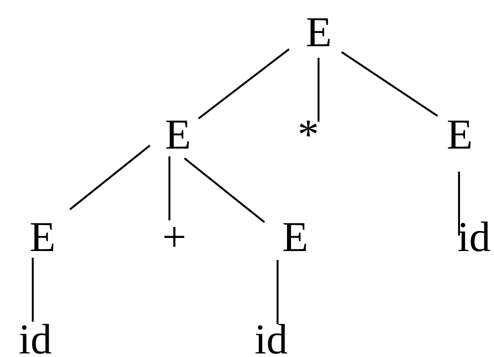
rightmost derivation –
reverse in postordering

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*
- Example : id + id * id

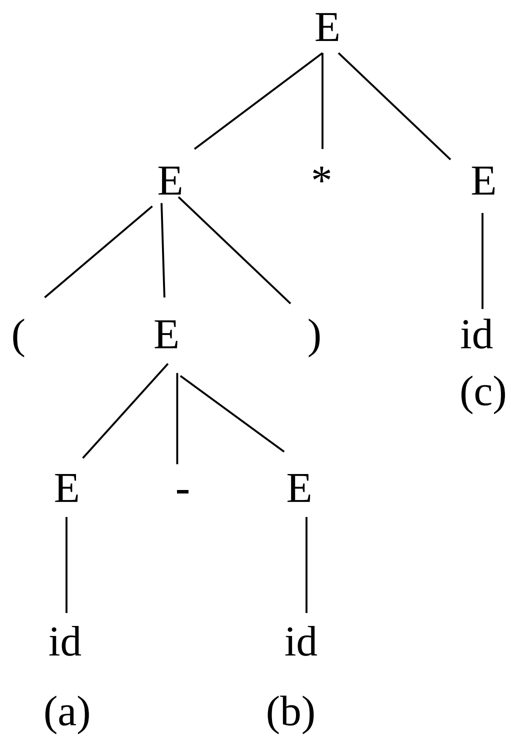
$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow id + E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
\qquad\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$

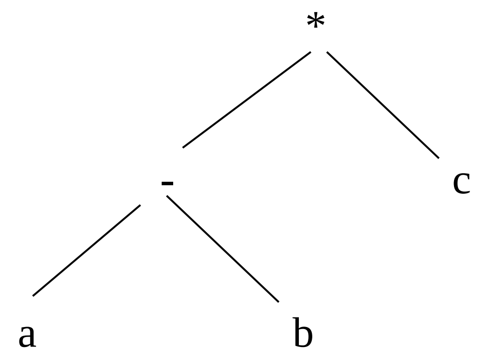




Syntax Trees

- Parse tree contains much more information than is necessary for a compiler to produce executable code
- Syntax tree(or abstract syntax tree) is an abstraction of the actual source code token sequences

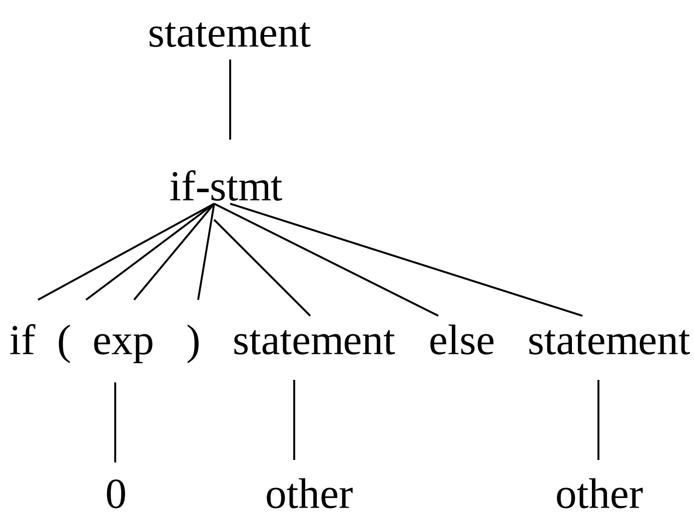




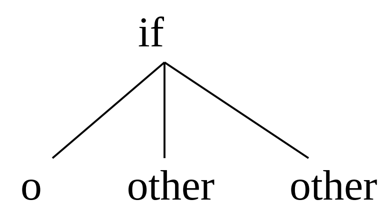
Syntax Tree

Syntax Trees

- Example: if-statement



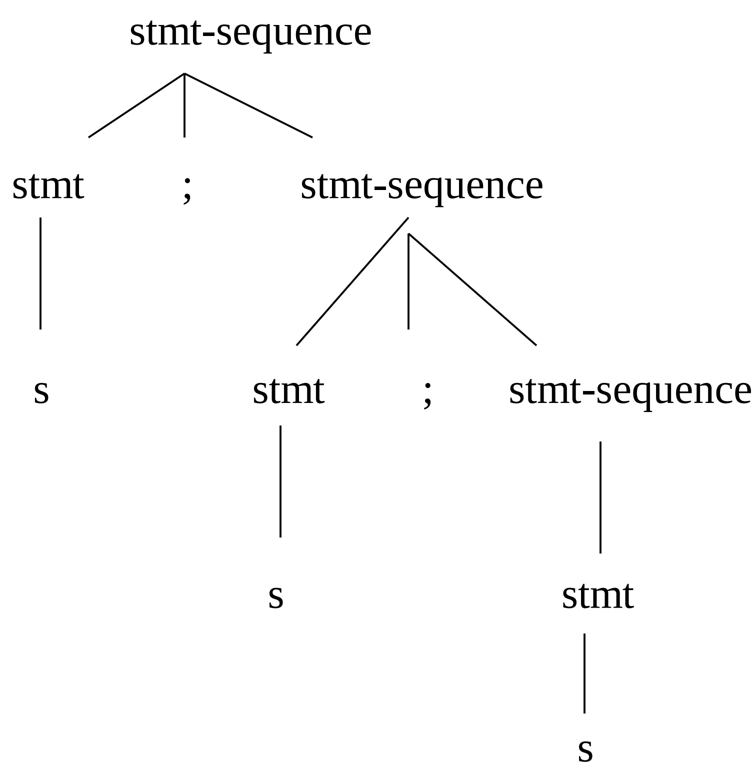
Parse Tree

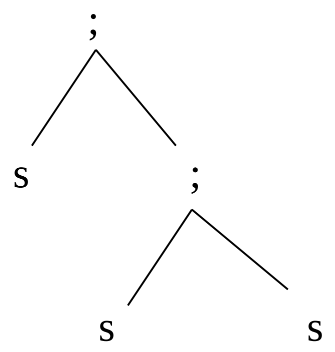


Syntax Tree

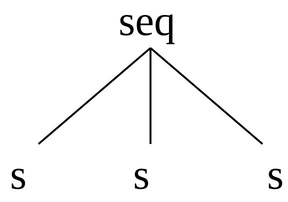
Syntax Trees

- Example: Statement Sequence





or



Writing a Grammar

- Grammatical rules of programming languages
  - lexical rule : regular expressions - lexical analyzer
  - syntactic rule: CFG - parser
  - semantic rule(context-sensitive syntax): semantic analyzer
    - e.g. requirement that identifiers should be declared before they are used
- Each parsing method can handle grammars only of a certain form
  - rewriting initial grammars
  - using operator precedence and associativity
  - left-recursion elimination
  - left factoring

Regular Expressions vs CFG

- Regular Grammar : <T, N, S, P>
  - right regular grammar (right linear grammar)
    - A -> aB
    - A -> a
    - A -> ε
  - left regular grammar (left linear grammar)
    - A -> Ba
    - A -> a
    - A -> ε

- A regular grammar is a CFG
  - but, simple and easy to implement efficiently

Regular Expressions vs CFG

- Every regular expression can be converted into a CFG
  - example : (a|b)*abb

    A_0 -> aA_0 | bA_0 | aA_1

    A_1 -> bA_2

    A_2 -> bA_3

    A_3 -> ε
- Converting a NFA to a regular grammar
  - state i $\Rightarrow$ non-terminal Ai
  - transition j = T(i, a) $\Rightarrow$ production A_i -> aA_j
  - transition j = T(i, ε) $\Rightarrow$ production A_i -> A_j
  - i $\in$ F $\Rightarrow$ production A_i -> ε
  - i = start state $\Rightarrow$ A_i : start symbol

Eliminating Ambiguity

- Using Operator Precedence and Associativity
  - Example:

    E -> E + E

    E -> E - E

    E -> E * E

    E -> (E)

    E -> -E

    E -> id
  - Grouping operators of equal precedences
  - Lower precedence => higher in parse tree
  - Left Associativity => left recursion

    E -> E **addop** T | T

    T -> T **multop** F | F

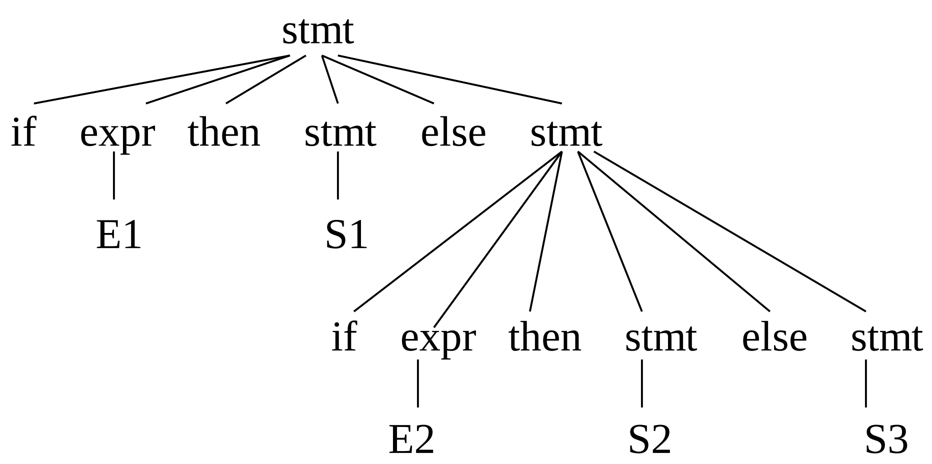
    F -> **id** | (E) | - E

Eliminating Ambiguity

- Dangling-else grammar

  stmt -> **if** expr **then** stmt
  | **if** expr **then** stmt **else** stmt
  | S

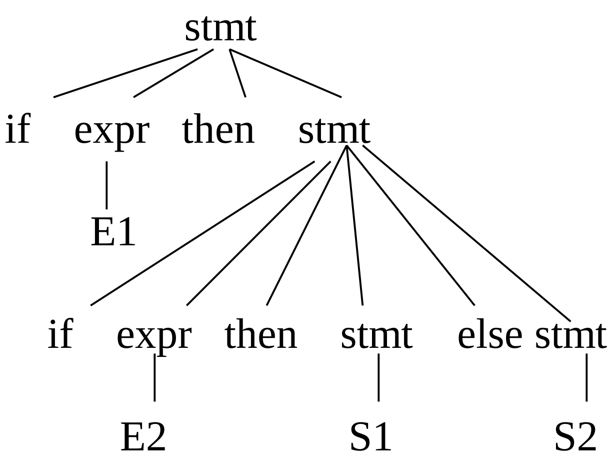
  if E1 **then** S1 **else if** E2 **then** S2 **else** S3

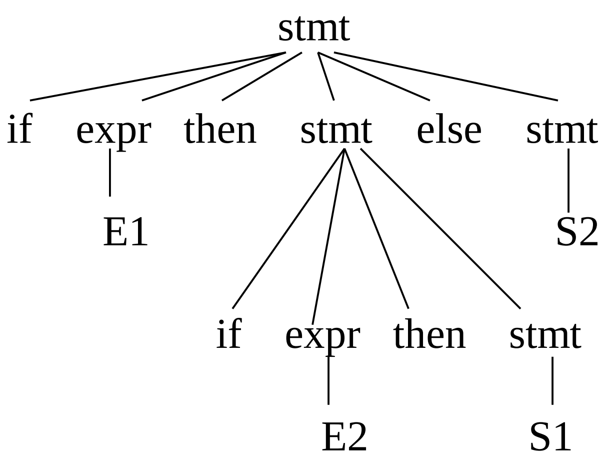


Eliminating Ambiguity

- Ambiguous if-stmt

if E1 **then** **if** E2 **then** S1 **else** S2





Eliminating Ambiguity

- Rewriting grammar

stmt -> matched_stmt

| unmatched_stmt

matched_stmt-> **if** expr **then** matched_stmt **else** matched stmt

| **other**

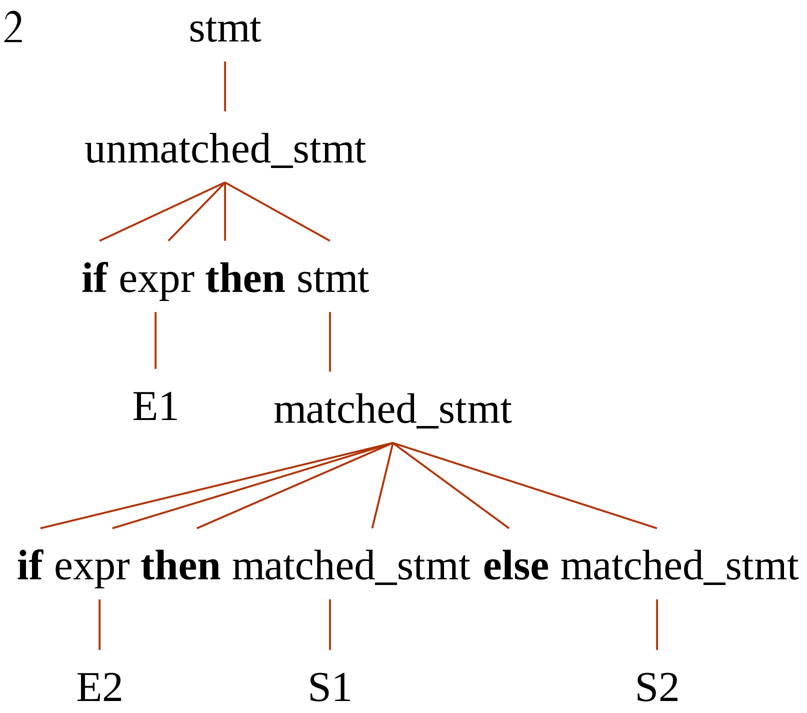
unmatched_stmt -> **if** expr **then** stmt

| **if** expr **then** matched_stmt **else** unmatched_stmt

Eliminating Ambiguity

- Rewriting grammar

if E1 **then** **if** E2 **then** S1 **else** S2



Elimination of Left Recursion

- A grammar is left-recursive if there is A $\Rightarrow$ A α

  expr -> expr + term | term

- Top-down parsing cannot handle left-recursive grammars
- Eliminating left recursion

  A -> A a | b $\Rightarrow$ A -> bA'

  A' -> aA' | ε

  - Example

    E -> E + T | T $\Rightarrow$ E -> TE'

    T -> T * F | F

    F -> (E) | id

    E' -> +TE' | ε

    T -> FT'

    T' -> *FT' | ε

    F -> (E) | id

Elimination of Left Recursion

- General case (immediate left-recursion)

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

=>

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

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

- It does not eliminate left recursion involving derivations of two or more steps
- Non-immediate left-recursion

S ->Aa | b		S->Aa |b
A -> Ac | Sd | ε	=>	A -> SdA' | A' // left-recursive
		A' -> cA' | ε

S => Aa => Sda

Elimination of Left Recursion

- Eliminating left-recursion (non-immediate recursion)
  - Arrange the nonterminals in some order, $A_1, A_2, \ldots, A_n$
  - for each Ai, (i = 1 to n)

    for j = 1 to i - 1

    replace $A_i \rightarrow A_j\gamma$

    by $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,

    where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are A_j-productions

    eliminate the immediate left recursion among the Ai-productions
- Example

  S -> Aa | b

  A -> Ac | Sd | ε

  1. ordering: S A
  2. for S, no left-recursion
  3. for A, replace A->Sd with S-productions

     A -> Ac | Aad | bd | ε
  4. elimination of left-recursion

     A->bdA' | A'

     A'->cA' | adA' | ε

Left Factoring

- Two productions for a non-terminal A have a common prefix

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

  $\Rightarrow$

  $A \rightarrow \alpha A'$

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

- General case

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \alpha\beta_n \mid \gamma$

  $\Rightarrow$

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

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

- Example

  S->iEtS| iEtSeS | a

  E ->b

  S -> iEtSS' | a

  S' -> eS | ε

  E -> b