WRITING A GRAMMAR

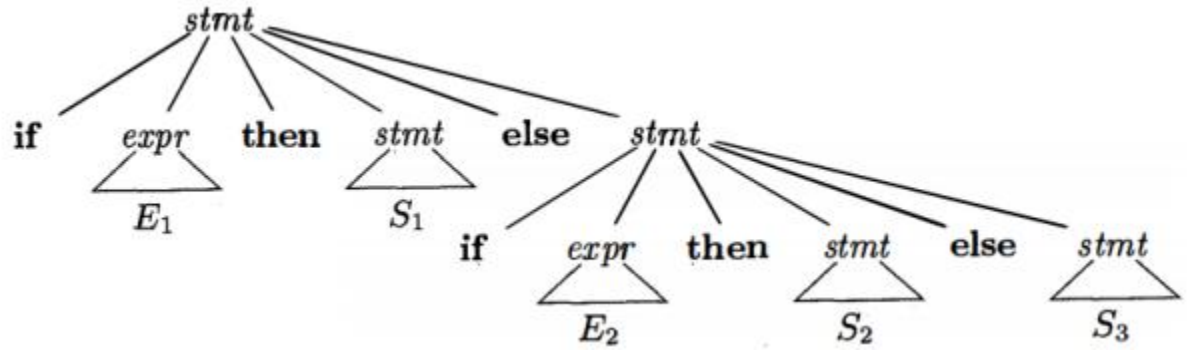
- Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar.
- Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

Eliminating Ambiguity

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar. Consider the following "dangling else" grammar.

stmt → if expr then stmt | if expr then stmt else stmt | other

Here "other" stands for any other statement. According to this grammar, the compound conditional statement if E1 then S1 else if E2 then S2 else S3

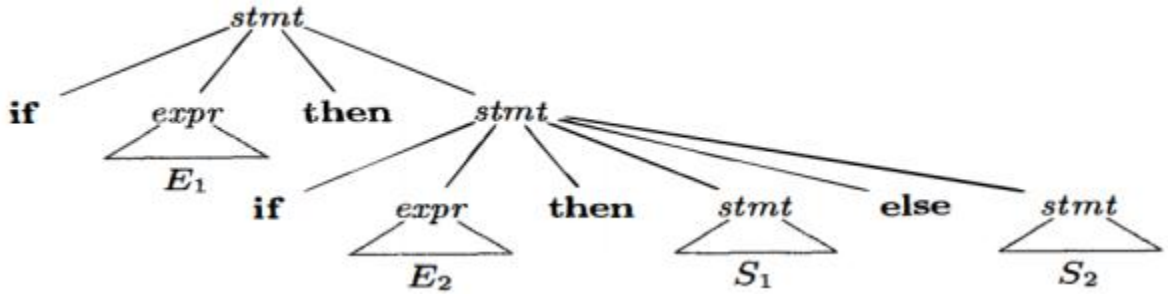


But the grammar is ambiguous since the string

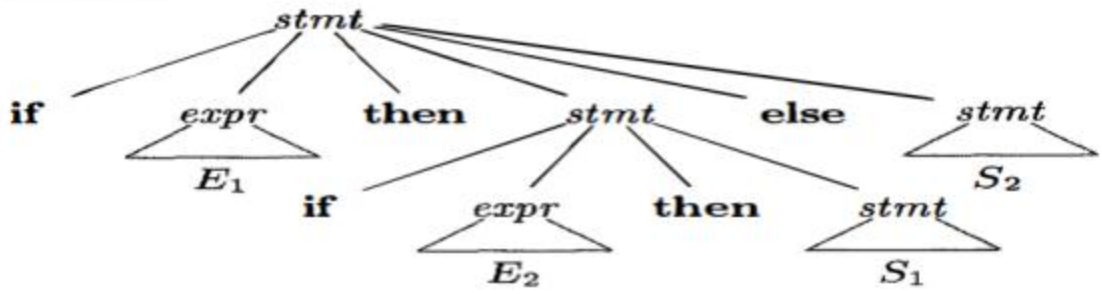
if E1 then if E2 then S1 else S2

Has the following two parse trees for leftmost derivation:

Parse Tree-1



Parse Tree-2



Elimination of Left Recursion:

- A grammar is left recursive if it has a non-terminal A such that there is a derivation

  A=> Aα
- Top down parsing methods can't handle left-recursive grammars
- A simple rule for immediate left recursion elimination for: A => A α|β
- We may replace it with

  A -> β A'

  A' -> α A' | ε

Example to eliminate Immediate left recursion: Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

Answer: Eliminate left recursive productions E and T by applying the left recursion

First eliminate the left recursion for E as

E → TE'

E' → +TE' | ε

Then eliminate for T as

T → FT'

T'→ *FT'| ε

Thus the obtained grammar after eliminating left recursion is

E → TE'

E' → +TE'| ε

T → FT'

T' → *FT'| ε

F → (E) | id

Left factoring

- Left factoring is a process of factoring out the common prefixes of two or more production alternates for the same nonterminal.
- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.

Consider following grammar:

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

On seeing input if it is not clear for the parser which production to use. So, can perform left factoring, where the general form is:

If we have **A → αβ1 | αβ2** then we replace it with

A → αA'

A' → β1 | β2

Example: Eliminate left factors from the given grammar. S → T + S | T

After left factoring, the grammar becomes,

S →T S'

S'→ + S | ε

Example: Left factor the following grammar.

S → aBcDeF | aBcDgg | F , B→ x, D→ y, F→ z

After left factoring, the grammar becomes,

S → aBcDS' | F

S' → eF | gg

B→ x

D→ y

F→ z

TOP DOWN PARSING

- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first).
- Top down parsing can be viewed as finding a leftmost derivation for an input string.
- Parsers are generally distinguished by whether they work top-down (start with the grammar's start symbol and construct the parse tree from the top) or bottom-up (start with the terminal symbols that form the leaves of the parse tree and build the tree from the bottom).
- Top down parsers include recursive-descent and LL parsers, while the most common forms of bottom up parsers are LR parsers