Context-free Grammars

Def: A **Context-free Grammar** (CFG) is a 4-tuple

$$G=(N, \Sigma, P, S)$$

where:

1. N is a finite, nonempty set of symbols (non-terminals)
2. Σ is a finite set of symbols (terminals)
3. $N \cap \Sigma =$ [illegible]
4. $V \equiv N \cup \Sigma$ (vocabulary)
5. $S \in N$ (Goal symbol or start symbol)
6. P is a finite subset of $N \times V^*$ (Production rules).

Sometimes written as $G=(V, \Sigma, P, S)$, $N = V \setminus \Sigma$.

Example Grammar: Arithmetic Expressions

G = (N, Σ, P, E) where:

N = { E, T, F}

Σ = { (,), +, *, <u>id</u>}

P = { $E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow F$
$T \rightarrow T*F$
$F \rightarrow$ <u>id</u>
$F \rightarrow (E)$ }

Note: $P \subseteq N x V^*$, where
$V = N \cup \Sigma$ = { E,T,F,C,(,),+,*,<u>id</u>}

Note: $(A, \alpha) \in P$ is usually written
$A \rightarrow \alpha$
or $A ::= \alpha$
or $A : \alpha$

Derivations of a Grammar

Directly Derives or $\Rightarrow$:

If α and β are strings in V^* (vocabulary), then α directly derives β (written $\alpha \Rightarrow \beta$) *iff* there is a production $A \rightarrow \delta$ s.t.

- A is a symbol in α
- Substituting string δ for A in α produces the string β

Canonical Derivation Step:

The above derivation step is called rightmost if A is the rightmost non-terminal in α. (Similarly, leftmost.)

A rightmost derivation step is also called canonical.

Derivations and Sentential Forms

Derivation:

A sequence of steps $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ where $\alpha_0 = S$ is called a *derivation*. It is written $S \overset{*}{\Rightarrow} \alpha_n$

If every derivation step is rightmost, then this is a *canonical derivation*.

Sentential Form

Each α_i in a derivation is called a *sentential form* of G.

Sentences and the Language L(G)

A sentential form α_i consisting only of tokens (i.e., terminals) is called a *sentence* of *G*.

The *language generated by* G is the set of all sentences of *G*. It is denoted *L(G)*.

Parse Trees of a Grammar

A **Parse Tree** for a grammar G is any tree in which:

- The root is labeled with *S*
- Each leaf is labeled with a token *a* ($a \in \Sigma$) or ε (the empty string)
- Each interior node is labeled by a non-terminal.
- If an interior node is labeled *A* and has children labeled $X_1 \ldots X_n$, then $A \rightarrow X_1 \ldots X_n$ is a production of G
- If $A \rightarrow \varepsilon$ is a production in G, then a node labeled *A* may have a single child labeled ε

The string formed by the leaf labels (left to right) is the **yield** of the parse tree.

Parse Trees (continued)

- An **intermediate parse tree** is the same as a parse tree except the leaves can be non-terminals.

Notes:

- Every $\alpha \in L(G)$ is the yield of *some* parse tree. **Why?**
- Consider a derivation, $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, where $\alpha_n \in L(G)$

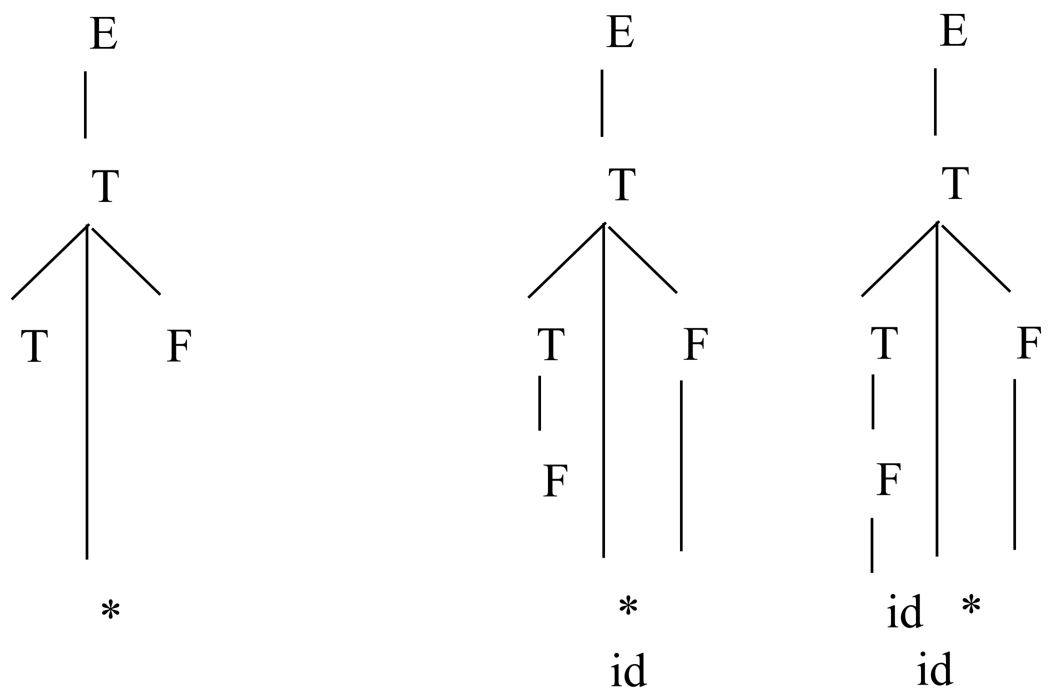
  For each α_i, we can construct an intermediate parse tree.

  The last one will be the parse tree for the sentence α_n.
- A parse tree ignores the order in which symbols are replaced to derive a string.

Derivations and Parse Trees

$\underline{\text{id}} * \underline{\text{id}}$

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \underline{\text{id}} \Rightarrow F * \underline{\text{id}} \Rightarrow \underline{\text{id}} * \underline{\text{id}}$$



Uniqueness of Derivations

Derivations and Parse Trees

- Every *parse tree* has a unique *derivation*: Yes? No?
- Every *parse tree* has a unique *rightmost derivation*: Yes? No?
- Every *parse tree* has a unique *leftmost derivation*: Yes? No?

Derivations and Strings of the Language

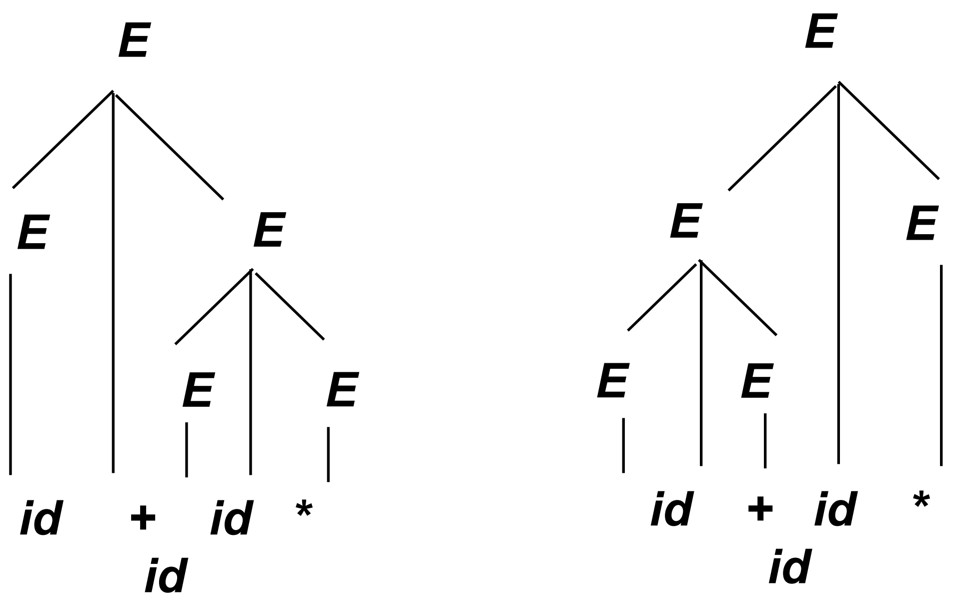
- Every $u \in L(G)$ has a unique *derivation*: Yes? No?
- Every $u \in L(G)$ has a unique *rightmost derivation*: Yes? No?
- Every $u \in L(G)$ has a unique *leftmost derivation*: Yes? No?

Ambiguity

Def. A grammar, G, is said to be **unambiguous** if $\forall$ u $\in$ L(G), $\exists$ exactly one canonical derivation S $\Rightarrow^*$ u. Otherwise, G is said to be **ambiguous**.

E.g., *Grammar*: E $\rightarrow$ E + E | E * E | (E) | id

Two parse trees for u = id + id * id



These are different syntactic interpretations of the input code

Order of Evaluation of Parse Tree

Note: These are conventions, not theorems

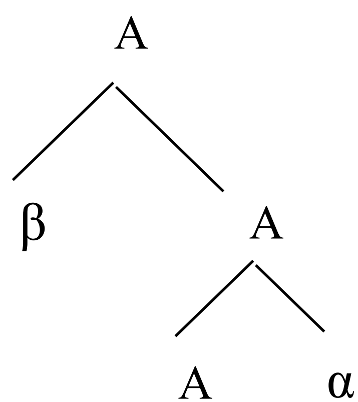
- Code for a non-terminal is evaluated as a single "block"
  - I.e., cannot partially execute it, then execute something else, then evaluate the rest
  - A different parse tree would be needed to achieve that
  - E.g. 1: Non-terminal T enforces precedence of * over +
  - E.g. 2: E $\rightarrow$ E + T enforces left-associativity,
    E $\rightarrow$ T + E enforces right-associativity.
- Parse tree does *not* specify order of execution of code blocks
  - Must be enforced by the code generated for parent block. Obey:
    - » Operator (e.g, +) cannot be evaluated before operands
    - » Associativity rules

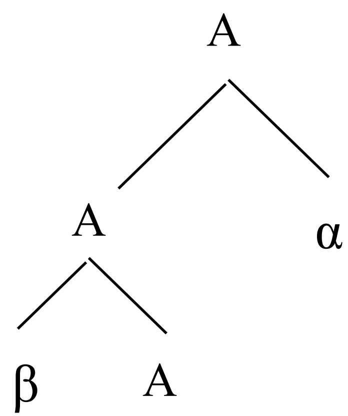
Detecting Ambiguity

<u>*Caution*</u>: There is no mechanical algorithm to decide whether an arbitrary CFG is ambiguous.

But one common kind of ambiguity <u>can</u> be detected:

If a symbol, $A \in N$ is both left-recursive (I.e., $A \Rightarrow^{+} A\alpha$, $|\alpha| \geq 0$) *and* right-recursive (i.e., $A \Rightarrow^{+} \beta A$, $|\beta| \geq 0$), then G is ambiguous, provided that G is "reduced" (i.e., has no "redundant" symbols).





Removal of Ambiguity: Example 1

1. Enforce higher precedence for *

$$E \rightarrow E + E \mid T$$
$$T \rightarrow T * T \mid \underline{id} \mid (E)$$

2. Eliminate right-recursion for $E \rightarrow E + E$ and $T \rightarrow T * T$.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * \underline{id} \mid T * (E) \mid \underline{id} \mid (E)$$

Removal of Ambiguity: Example 2

The Infamous *Dangling-Else* Grammar:

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

Solution**:** Introduce new non-terminals to distinguish matched **then/else**

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*