Context-free grammars (CFGs)

- Example: Simple Arithmetic Expressions
  - In English:
    - An integer is an arithmetic expression.
    - If exp_1 and exp_2 are arithmetic expressions, then so are the following:

      exp_1 - exp_2

      exp_1 / exp_2

      (exp_1)

- the corresponding CFG:

  E → *integer*

  E → E - E

  E → E / E

  E → (E)

Reading the CFG

- The grammar has five terminal symbols:
  - *integer*, **-**, **/**, **(**, **)**
  - terminals of a grammar = tokens returned by the scanner.
- The grammar has one non-terminal symbol:
  - **E**
  - non-terminals describe valid sequences of tokens
- The grammar has four productions or rules,
  - each of the form: E → α
    - left-hand side E = a single non-terminal.
    - right-hand side α = either
      - a sequence of one or more terminals and/or non-terminals, or
      - ε (the empty string)

Example, revisited

- Note:
  - a more compact way to write previous grammar:

    E → *integer* | E - E | E / E | (E)

    or

    E → *integer*
    | E - E
    | E / E
    | (E)

A formal definition of CFGs

- A CFG consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S (one of the non-terminals)
  - A set of *productions:*

$$X \rightarrow Y_1 Y_2 \cdots Y_n$$

$$\text{where } X \in N \text{ and } Y_i \in T \cup N \cup \{\varepsilon\}$$

Notational Conventions

- In these lecture notes
  - Non-terminals are written in upper-case
  - Terminals are written in lower-case
  - The start symbol is the left-hand side of the first production (unless specified otherwise)

The Language of a CFG

The language defined by a CFG is the set of strings that can be derived from the start symbol of the grammar.

Derivation: Read productions as rules:

$$X \to Y_1 \cdots Y_n$$

Means X can be replaced by $Y_1 \cdots Y_n$

Derivation: key idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by the right-hand side of some production

$$X \rightarrow Y_1 \cdots Y_n$$

3. Repeat (2) until there are no non-terminals in the string

Derivation: an example

CFG:

- E → id
- E → E + E
- E → E * E
- E → (E)

String id * id + id is in the language defined by the grammar.

Derivation:

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

Terminals

- Terminals are so called because there are no rules for replacing them
- Once generated, terminals are permanent
- Therefore,
  terminals are the tokens of the language

The Language of a CFG (continued)

More formally, we can write

$$X_1 \cdots X_i \cdots X_n \rightarrow X_1 \cdots X_{i-1} Y_1 \cdots Y_m X_{i+1} \cdots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \cdots Y_m$$

The Language of a CFG (continued)

Write

$$X_1 \cdots X_n \overset{*}{\rightarrow} Y_1 \cdots Y_m$$

if

$$X_1 \cdots X_n \rightarrow \cdots \rightarrow \cdots \rightarrow Y_1 \cdots Y_m$$

using a sequence of 0 or more replacement steps

The Language of a CFG

Let G be a context-free grammar with start symbol S. Then the language of G is:

$$\left\{ a_1 \ldots a_n \mid S \overset{*}{\rightarrow} a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \right\}$$

Example

Strings of balanced parentheses $\left\{ (^{i})^{i} \mid i \geq 0 \right\}$

The grammar:

$$S \rightarrow (S) \qquad S \rightarrow \varepsilon$$

Which is the same as

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

Another Example

A simple arithmetic expression grammar:

$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

Some strings in the language of this grammar:

id	id + id
(id)	id * id
(id) * id	id * (id)

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \rightarrow \cdots \rightarrow \cdots \rightarrow \cdots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \cdots Y_n$ add children $Y_1 \cdots Y_n$ to node X

Derivation Example

- Grammar

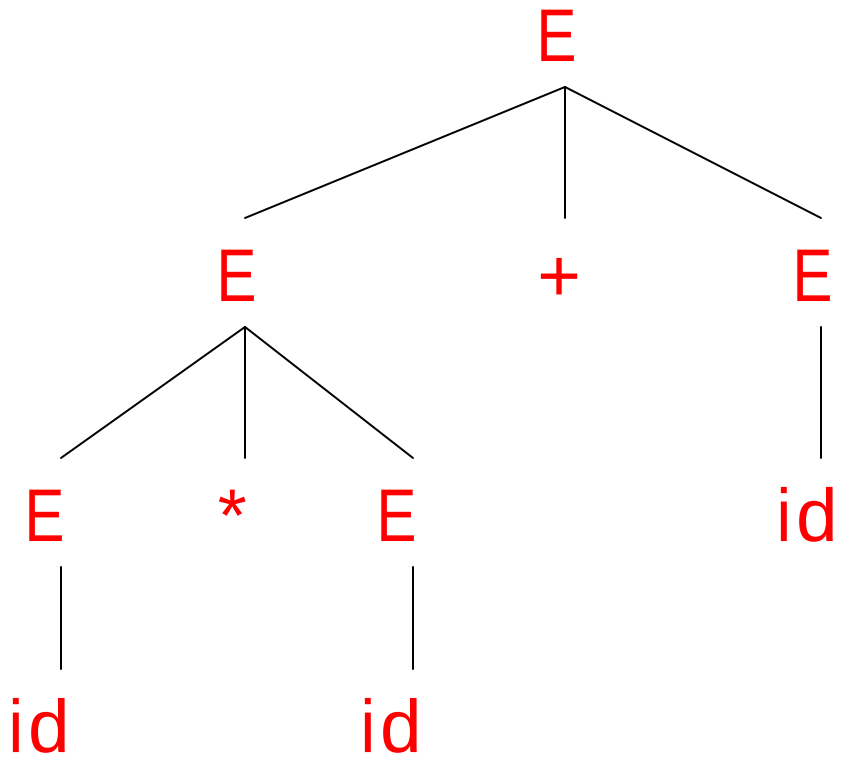
$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (continued)

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



Notes on Derivations

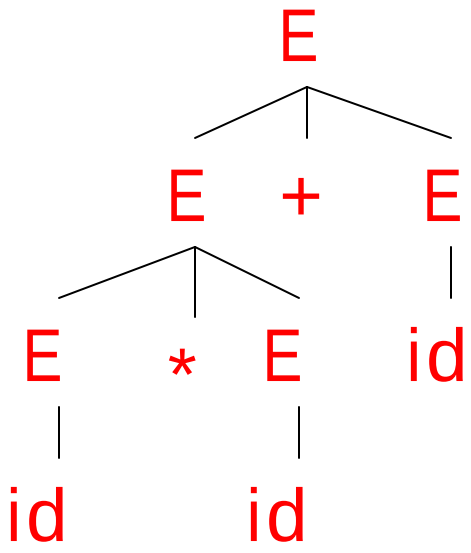
- A syntax tree or parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves yields the original input string
- As in the preceding example, we usually show a left-most derivation, that is, replace the left-most non-terminal remaining at each step

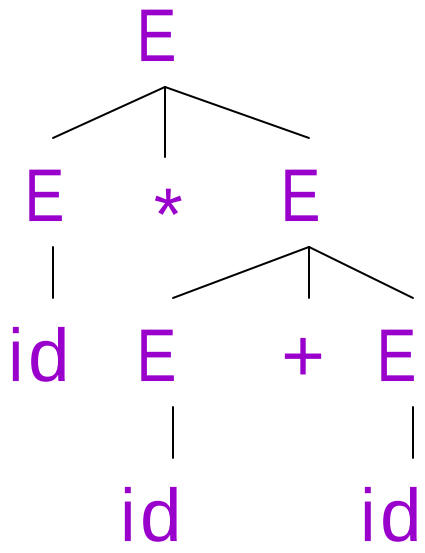
Ambiguity

- Grammar $E \rightarrow E+E \mid E * E \mid (E) \mid id$
- String $id * id + id$

Ambiguity (continued)

This string has two parse trees for this grammar





Test yourself

Question 1:

- for each of the two parse trees, write the corresponding left-most derivation

Question 2:

- for each of the two parse trees, write the corresponding right-most derivation

Ambiguity (continued)

- A grammar is *ambiguous* if for at least one string:
  - the string has more than one parse tree
  - the string has more than one left-most derivation
  - the string has more than one right-most derivation
- Note that these three conditions are equivalent

- Ambiguity is **BAD**
  - because if the grammar is ambiguous then the meaning of some programs is not well-defined

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously
- For example, enforce precedence of * and / over + and -

Enforcing Correct Precedence

- Rewrite the grammar
  - use a different nonterminal for each precedence level
  - start with the lowest precedence (minus)

E → E - E | E / E | (E) | id

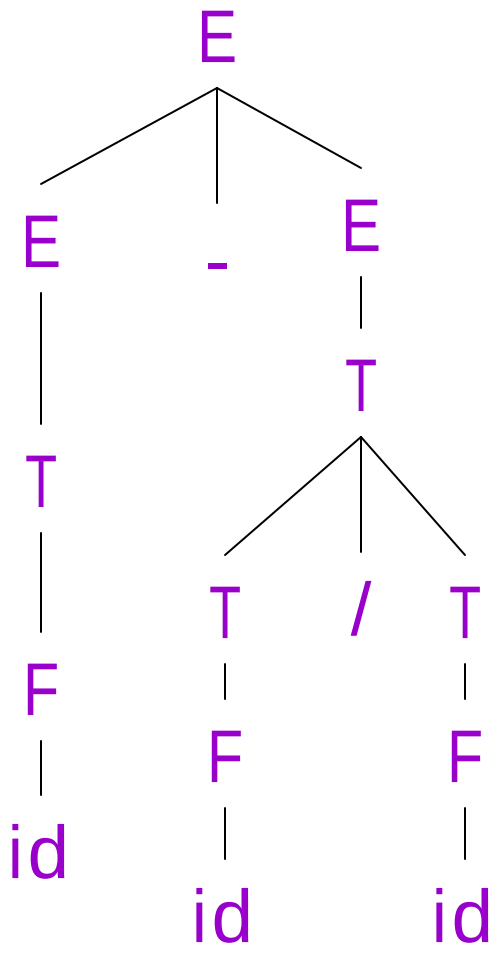
rewrite to

E → E - E | T
T → T / T | F
F → id | (E)

Example

parse tree for id - id / id

E → E - E | T
T → T / T | F
F → id | (E)



Test yourself

Question 3:

- Attempt to construct a parse tree for id-id/id that shows the *wrong* precedence.
  - Why do you fail to construct such a parse tree?

Question 4:

- Draw two parse trees for the expression a-b-c
  - One should correctly group ((a-b)-c), and one should incorrectly group (a-(b-c))

Enforcing Correct Associativity

- The grammar captures operator precedence, but it is still ambiguous
  - fails to express that both subtraction and division are *left* associative;
    - 5-3-2 is equivalent to: ((5-3)-2) but *not* to: (5-(3-2)).
    - 8/4/2 is equivalent to: ((8/4)/2) but *not* to: (8/(4/2)).

Recursion

- A grammar is **recursive in nonterminal X** if:
  - X $\rightarrow^+$ … X …
    - the notation $\rightarrow^+$ means "after one or more steps, X derives a sequence of symbols that includes another X"
- A grammar is **left** recursive in X if:
  - X $\rightarrow^+$ X …
    - after one or more steps, X derives a sequence of symbols that *starts* with an X
- A grammar is **right** recursive in X if:
  - X $\rightarrow^+$ … X
    - after one or more steps, X derives a sequence of symbols that *ends* with an X

How to fix associativity

- The grammar given above is both left and right recursive in non-terminals exp and term
  - Try at home: write the derivation steps that show this.

- To correctly express operator associativity:
  - For left associativity, use only left recursion.
  - For right associativity, use only right recursion.

- Here's the correct grammar:

  E → E – T | T
  T → T / F | F
  F → id | (E)

Ambiguity: The Dangling Else Problem

- Consider the grammar

$$S \rightarrow \text{if } E \text{ then } S$$
$$\mid \text{if } E \text{ then } S \text{ else } S$$
$$\mid a$$
$$E \rightarrow b$$

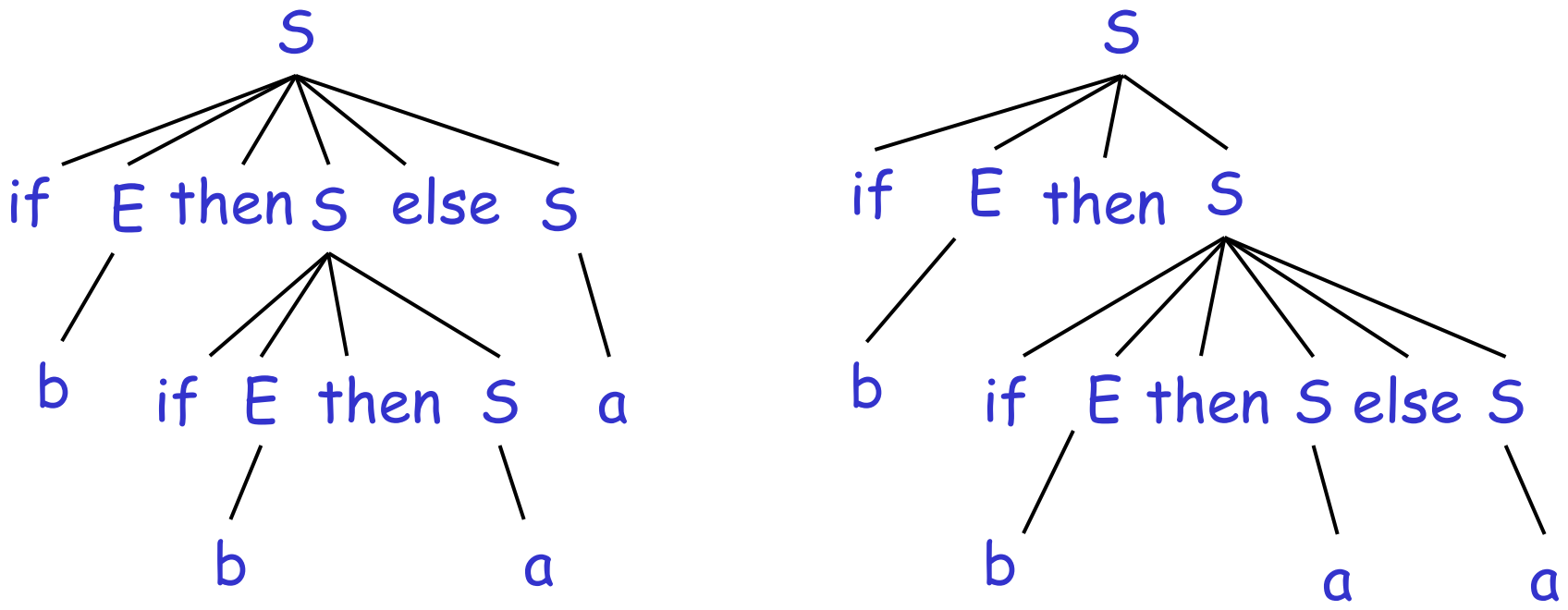
- This grammar is ambiguous

The Dangling Else Problem: Example

- The input string

  if b then if b then a else a

  has two different parse trees:



The Dangling Else Problem: How to Fix

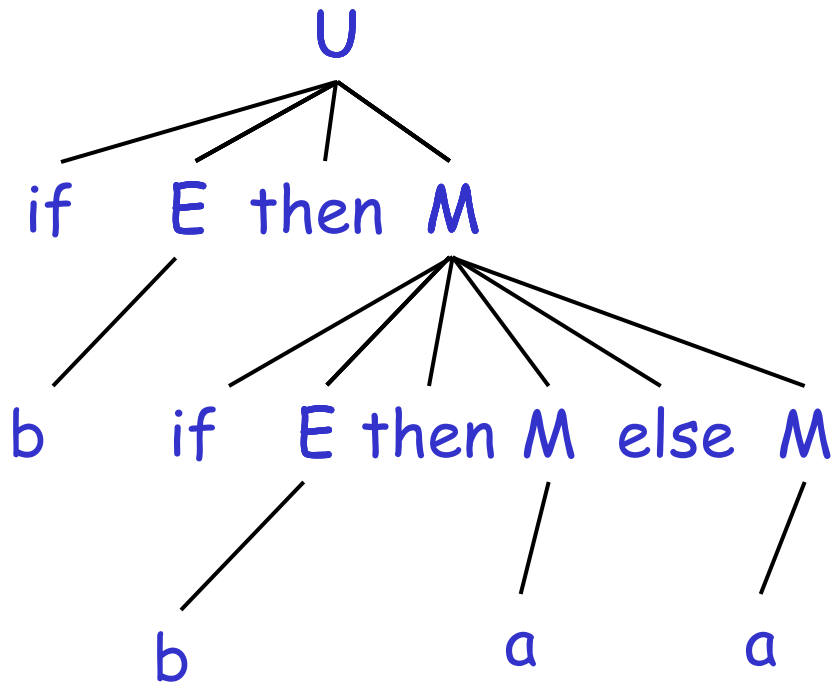
- else should match the closest unmatched then
- We can enforce this in a grammar:

```
S → M                          /* all then are matched by else */
  | U                          /* some then are unmatched */

M → if E then M else M
  | a

U → if E then S
  | if E then M else U
```

- Note: still generates the same set of strings

The Dangling Else Problem: Example Revisited

- Consider: if b then if b then a else a



- There is now only one possible parse tree for this string
- Try to draw a different parse tree and you should see why this is true

Backus-Naur Form

BNF is a special syntax for writing a CFG:

CFG

E → E+T | E–T | T
T → T*F | T/F | F
F → id | (E)

BNF

<Expr> ::= <Expr> + <Term> | <Expr> – <Term> | <Term>
<Term> ::= <Term> * <Fact> | <Term> / <Fact> | <Fact>
<Fact> ::= id | (<Expr>)

Extended BNF

EBNF permits any regular expression on right side of productions:

BNF

<Expr> ::= <Expr> + <Term> | <Expr> – <Term> | <Term>
<Term> ::= <Term> * <Fact> | <Term> / <Fact> | <Fact>
<Fact> ::= id | (<Expr>)

EBNF

<Expr> ::= <Term> (“+” <Term> | “–” <Term>)*
<Term> ::= <Fact> (“*” <Fact> | “/” <Fact>)*
<Fact> ::= id | “(” <Expr> “)”

Test yourself

Question 5:

- Write a CFG, BNF, or EBNF for each of these languages:
  - Strings of the form $a^n b^n$. Example: aaabbb
  - Strings $a^m b^n$ such that m>n. Example: aaaabb
  - Strings $a^m b^n$ such that m<n. Example: aabbbb
  - Strings over {a, b} such that the number of a's equals the number of b's. Example: baabba
  - Strings of the form $a^m b^n c^p$. Example: aabbbcccc
  - Strings of the form $a^m b^n c^p$ such that either m=n or n=p. Examples: aabbcccc, aabbbbcccc