Principles of Programming Languages

COMP251: Syntax and Grammars

Prof. Dekai Wu

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China

Fall 2006

Part II

Grammar

Grammar: Motivation

What do the following sentences really mean?

- 路不通行不得在此小便
- "I saw a small kid on the beach with a binocular."
- What is the final value of x?

```
x = 15
if (x > 20) then
if (x > 30) then
x = 8
else
x = 9
```

- 楊 乃 武 與 小 白 菜

Ambiguity in semantics is often caused by ambiguous grammar of the language.

A Formal Description: Example 7

1. $<$ *real-number* $>$ ::= $<$ *integer-part* $>$. $<$ *fraction* $>$
2. $<$ *integer-part* $>$ ::= $<$empty$>$ | $<$ *digit-sequence* $>$
3. $<$ *fraction* $>$ ::= $<$ *digit-sequence* $>$
4. $<$ *digit-sequence* $>$ ::= $<$ *digit* $>$ | $<$ *digit* $><$ *digit-sequence* $>$
5. $<$ *digit* $>$ ::= 0|1|2|3|4|5|6|7|8|9

This is the context-free grammar of real numbers written in the Backus-Naur Form.

Context Free Grammar (CFG)

A context-free grammar has **4** components:

1. **A set of tokens or terminals**:
   atomic symbols of the language.

   English : `a, b, c, ...., z`
   Reals : `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .`

2. **A set of nonterminals**:
   variables denoting language constructs.

   English : $< \textit{Noun} >, < \textit{Verb} >, < \textit{Adjective} >, \ldots$
   Reals : $< \textit{real-number} >, < \textit{integer-part} >, < \textit{fraction} >,$
   $< \textit{digit-sequence} >, < \textit{digit} >$

Context Free Grammar ..

3. **A set of rules called productions**:
   for generating expressions of the language.

   nonterminal ::= a string of terminals and nonterminals

   English : $< \textit{Sentence} > ::= < \textit{Noun} > < \textit{Verb} > < \textit{Noun} >$
   Reals : $< \textit{integer-part} > ::= <\text{empty}> | < \textit{digit-sequence} >$

   Notice that CFGs allow only a single non-terminal on the left-hand side of any production rules.

4. **A nonterminal chosen as the start symbol**:
   represents the main construct of the language.

   English : $< \textit{Sentence} >$
   Reals : $< \textit{real-number} >$

The set of strings that can be generated by a CFG makes up a context-free language.

Backus-Naur Form (BNF)

One way to write context-free grammar.

- Terminals appear as they are.
- Nonterminals are enclosed by $<$ and $>$.
  e.g.: $<$ *real-number* $>$, $<$ *digit* $>$.
- The special empty string is written as <empty>.
- Productions with a common nonterminal may be abbreviated using the special "or" symbol "|".

  e.g. `X ::= W1, X ::= W2, ..., X ::= Wn`

  may be abbreviated as `X ::= W1 | W2 | ··· | Wn`

Top-Down Parsing: Example 8

- A parser checks to see if a given expression or program can be derived from a given grammar.

Check if ".5" is a valid real number by finding from the CFG of Example 6 a leftmost derivation of ".5":

< *real-number* >
=> < *integer-part* > . < *fraction* > [Production 1]
=> <empty> . < *fraction* > [Production 2]
=> .< *fraction* > [By definition]
=> .< *digit-sequence* > [Production 3]
=> .< *digit* > [Production 4]
=> .5 [Production 5]

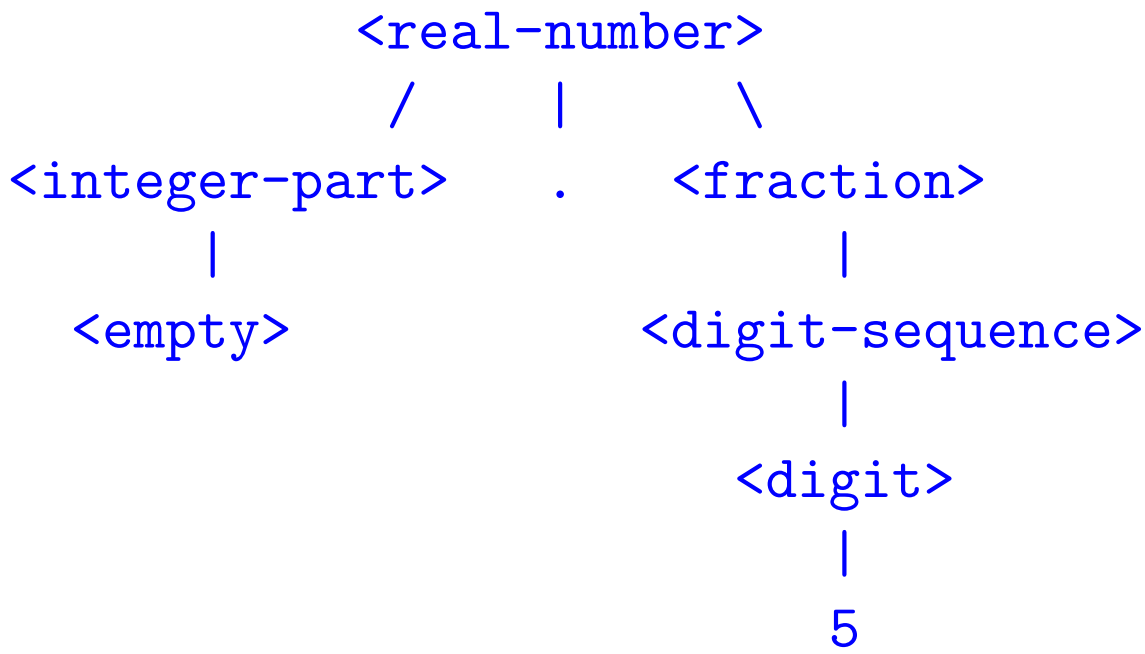
Bottom-Up Parsing: Example 9

Check if ".5" is a valid real number by finding from the CFG of Example 6 a rightmost derivation of ".5" in reverse:

.5 = <empty>.5 [By definition]
=> < *integer-part* > .5 [Production 2]
=> < *integer-part* > . < *digit* > [Production 5]
=> < *integer-part* > . < *digit-sequence* > [Production 4]
=> < *integer-part* > . < *fraction* > [Production 3]
=> < *real-number* > [Production 1]

Parse Tree: Example 10 [Real Numbers]

A parse tree of ".5" generated by the CFG of Example 6.



Parse Tree

A parse tree shows how a string is generated by a CFG — the concrete syntax in a tree representation.

- Root = start symbol.
- Leaf nodes = terminals or <empty>.
- Non-leaf nodes = nonterminals
- For any subtree, the root is the left-side nonterminal of some production, while its children, if read from left to right, make up the right side of the production.
- The leaf nodes, read from left to right, make up a string of the language defined by the CFG.

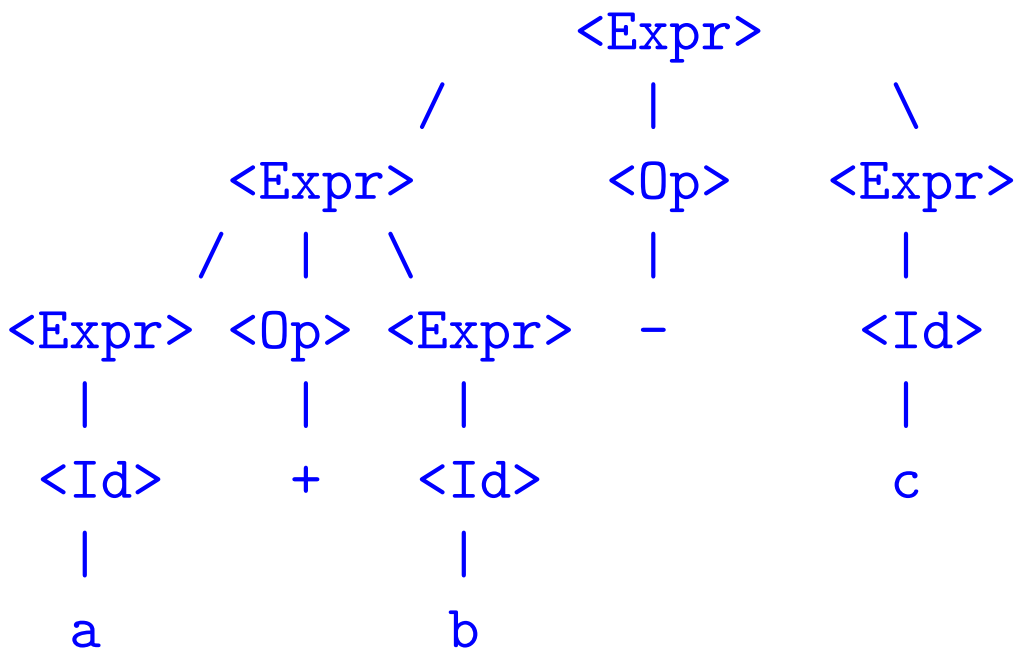
Example 11: CFG/BNF [Expression]

$$
\begin{array}{rcl}
< \textit{Expr} > & ::= & < \textit{Expr} >< \textit{Op} >< \textit{Expr} > \\
< \textit{Expr} > & ::= & (< \textit{Expr} >) \\
< \textit{Expr} > & ::= & < \textit{Id} > \\
< \textit{Op} > & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=} \\
< \textit{Id} > & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{array}
$$

1. Terminals: `a, b, c, +, -, *, /, =, (, )`
2. Nonterminals: *Expr*, *Op*, *Id*
3. Start symbol: *Expr*

Parse Tree : Example 12 [Expression]

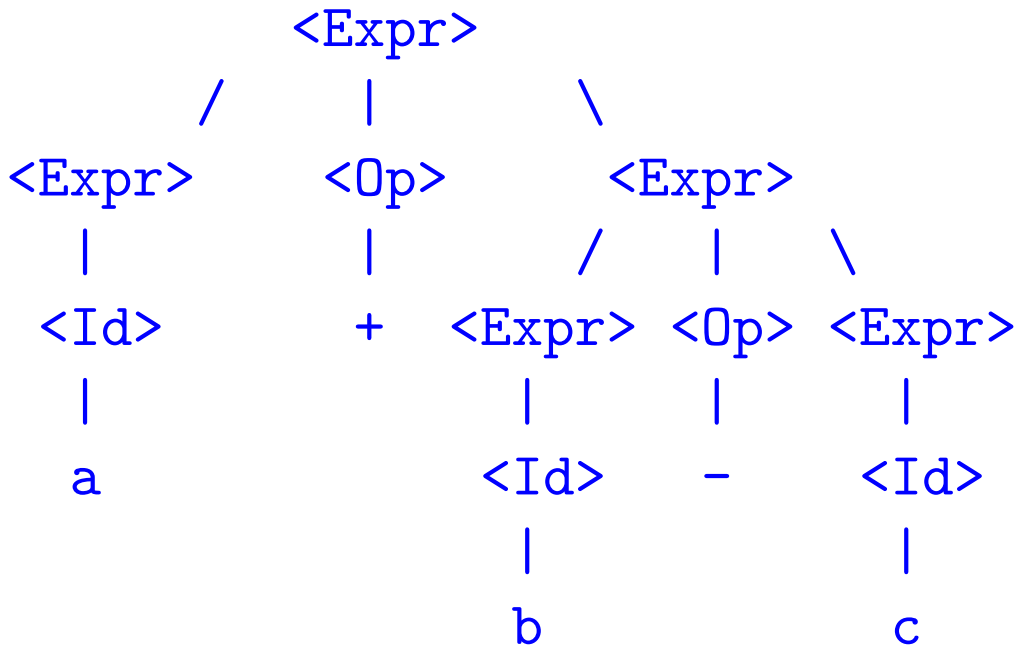
A parse tree of "$a + b - c$" generated by the CFG of Example 10:



Question: What is the difference between a parse tree and an abstract syntax tree?

Ambiguous Grammar: Example 13

A grammar is (syntactically) ambiguous if some string in its language is generated by more than one parse tree.



Solution: Rewrite the grammar to make it unambiguous.

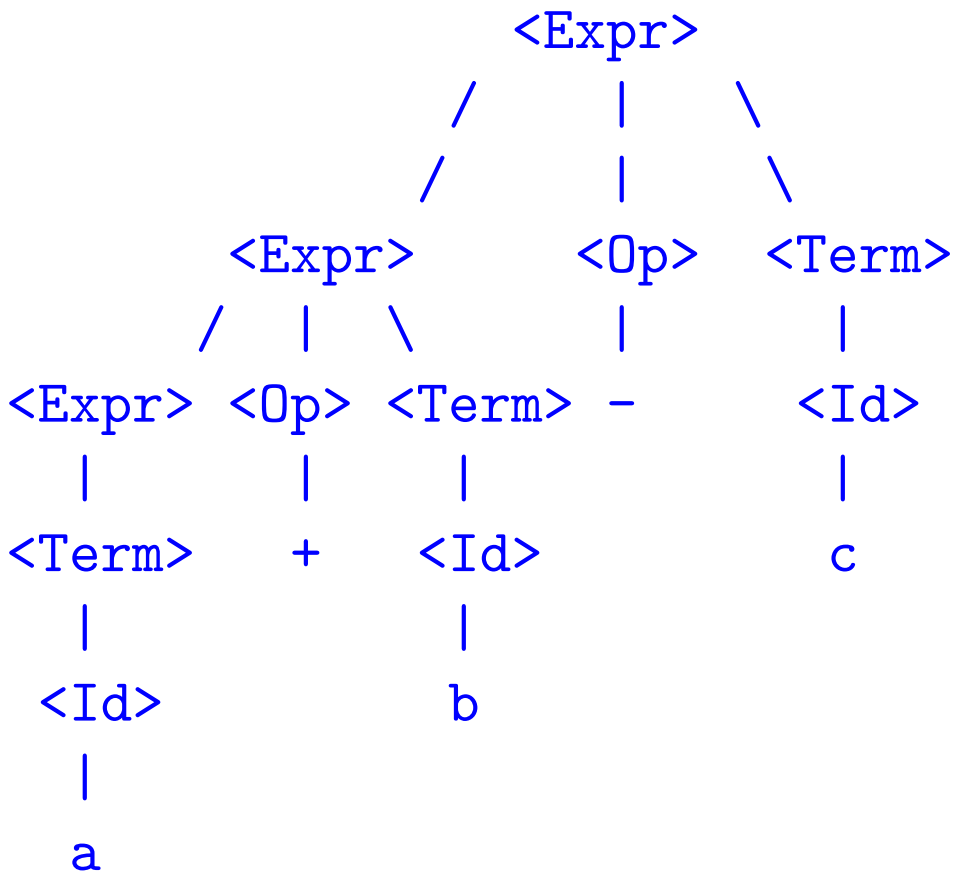
Handle Left Associativity: Example 14

CFG of Example 10 cannot handle "$a + b - c$" correctly.
$\Rightarrow$ Add a left recursive production.

$$
\begin{array}{rcl}
< Expr > & ::= & < Expr >< Op >< Term > \\
< Expr > & ::= & < Term > \\
< Term > & ::= & (< Expr >) | < Id > \\
< Op > & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=} \\
< Id > & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{array}
$$

Handle Left Associativity ..

Now there is only one parse tree for "$a + b - c$":



Handling Right Associativity: Example 15

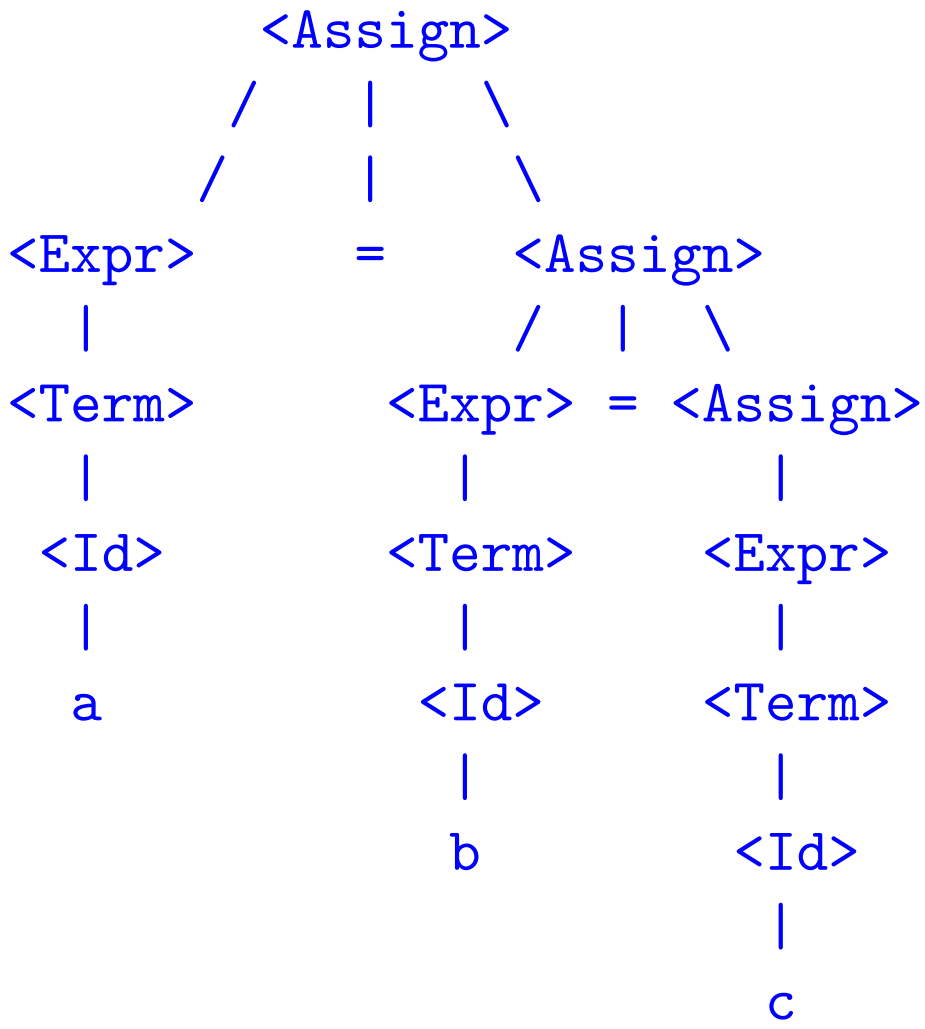
CFG of Example 10 cannot handle "$a = b = c$" correctly.
$\Rightarrow$ Add a right recursive production.

$$
\begin{array}{rcl}
< \textit{Assign} > & ::= & < \textit{Expr} > \texttt{=} < \textit{Assign} > \\
< \textit{Assign} > & ::= & < \textit{Expr} > \\
< \textit{Expr} > & ::= & < \textit{Expr} >< \textit{Op} >< \textit{Term} > \mid < \textit{Term} > \\
< \textit{Term} > & ::= & (< \textit{Expr} >) \mid < \textit{Id} > \\
< \textit{Op} > & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
< \textit{Id} > & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{array}
$$

Question: this grammar will accept strings like " a + b = c - d ".
Try to correct it.

Handling Right Associativity ..

Now there is only one parse tree for "$a = b = c$":



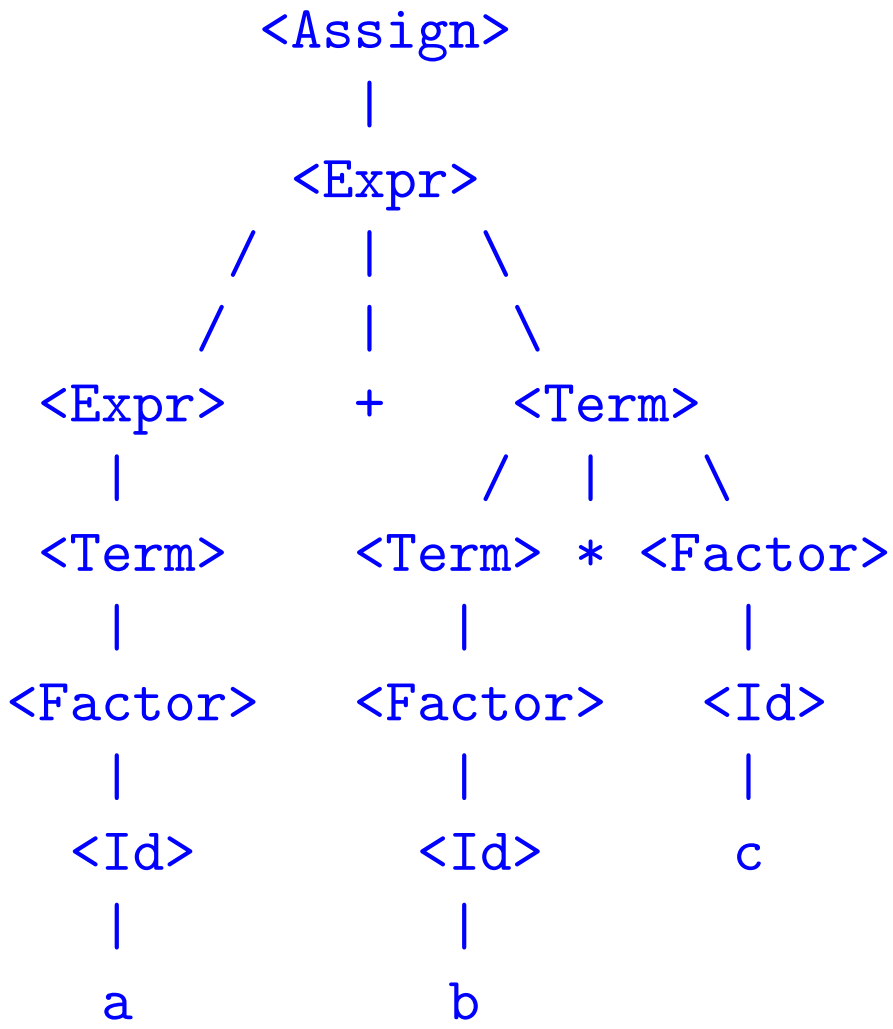
Handling Precedence: Example 16

CFG of Example 10 cannot handle "$a + b * c$" correctly.
$\Rightarrow$ Add one nonterminal (plus appropriate productions) for each precedence level.

$$
\begin{array}{rcl}
< Assign > & ::= & < Expr > \texttt{=} < Assign > \mid < Expr > \\
< Expr > & ::= & < Expr > + < Term > \\
< Expr > & ::= & < Expr > - < Term > \mid < Term > \\
< Term > & ::= & < Term > * < Factor > \\
< Term > & ::= & < Term > / < Factor > \mid < Factor > \\
< Factor > & ::= & (< Expr >) \mid < Id > \\
< Id > & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{array}
$$

Handling Precedence ..

Now there is only one parse tree for "$a + b * c$":



Tips on Handling Precedence/Associativity

- left associativity $\Rightarrow$ left-recursive production
- right associativity $\Rightarrow$ right-recursive production
- n levels of precedence
  - divide the operators into n groups
  - write productions for each group of operators
  - start with operators with the lowest precedence
- In all cases, introduce new non-terminals whenever necessary.
- In general, one needs a new non-terminal for each new group of operators of different associativity and different precedence.

Dangling-Else: Example 17

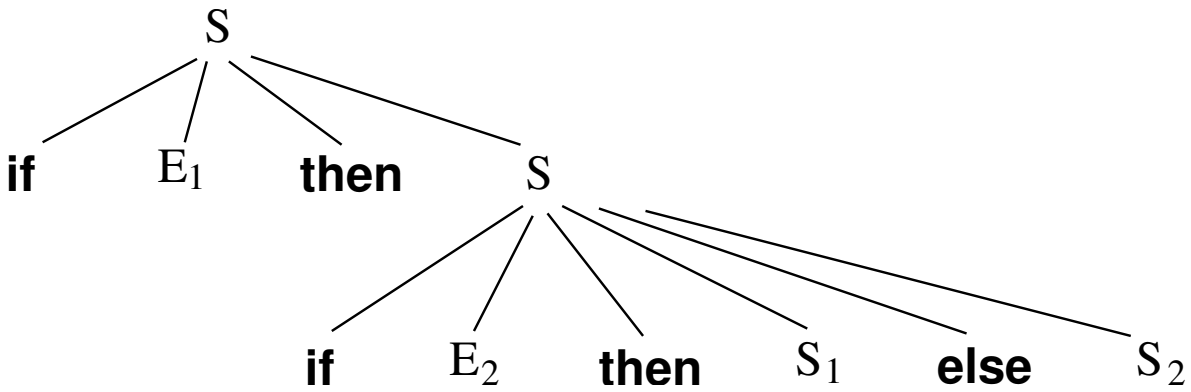
Consider the following grammar:

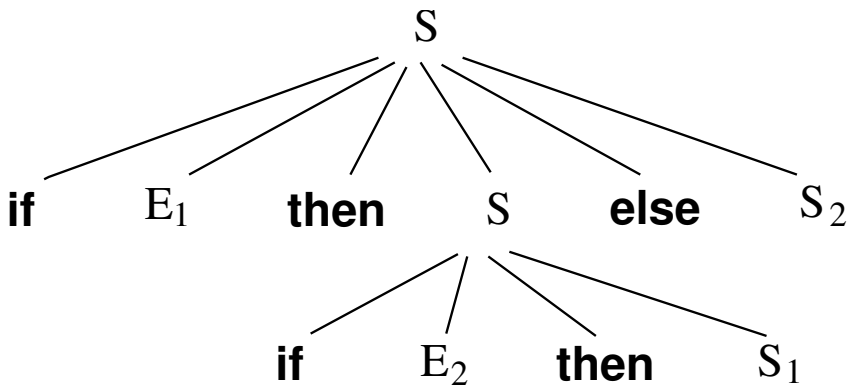
$$< S > \quad ::= \quad \textbf{if} < E > \textbf{then} < S >$$
$$< S > \quad ::= \quad \textbf{if} < E > \textbf{then} < S > \textbf{else} < S >$$

- How many parse trees can you find for the statement:

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

Dangling-Else ..





Dangling-Else ...

- Ambiguity is often a property of a grammar, not of a language.

Solution: matching an "**else**" with the nearest unmatched "**if**" .
i.e. the first case.

More CFG Examples

1.
$$
\begin{array}{lcl}
< S > & ::= & < A >< B >< C > \\
< A > & ::= & \mathrm{a}< A > \mid \mathrm{a} \\
< B > & ::= & \mathrm{b}< B > \mid \mathrm{b} \\
< C > & ::= & \mathrm{c}< C > \mid \mathrm{c}
\end{array}
$$

2.
$$
\begin{array}{lcl}
< S > & ::= & < A > \mathrm{a} < B > \mathrm{b} \\
< A > & ::= & < A > \mathrm{b} \mid \mathrm{b} \\
< B > & ::= & \mathrm{a}< B > \mid \mathrm{a}
\end{array}
$$

3.
$$
\begin{array}{lcl}
\text{<stmts>} & ::= & \text{<empty>} \mid \text{<stmt>} \ ; \ \text{<stmts>} \\
\text{<stmt>} & ::= & \text{<id>} := \text{<expr>} \\
 & & \mid \text{if <expr> then <stmt>} \\
 & & \mid \text{if <expr> then <stmt> else <stmt>} \\
 & & \mid \text{while <expr> do <stmt>} \\
 & & \mid \text{begin <stmts> end}
\end{array}
$$

Non-Context Free Grammars: Examples

1.
$$
\begin{array}{rcl}
< S > & ::= & < B >< A >< C > \mid < C >< A >< B > \\
\mathtt{b}< A > & ::= & \mathtt{c}< A >< B > \mid < B > \\
\mathtt{c}< A > & ::= & \mathtt{b}< A >< C > \mid < C > \\
< B > & ::= & \mathtt{b} \\
< C > & ::= & \mathtt{c}
\end{array}
$$

$\Rightarrow L = \{ (cb)^n,\ b(cb)^n,\ (bc)^n,\ c(bc)^n \}$.

2. $L = \{ wcw | w$ is a string of a's or b's $\}$.

This language abstracts the problem of checking that an identifier is declared before its use in a program.
The first w = declaration of the identifier, and
the second w = its use in the program.

Summary on Grammar

- √ Context-free grammar (CFG) is commonly used to specify most of the syntax of a programming language.
- √ However, most programming languages are not CFL!
- √ CFG is commonly written in Backus-Naur Form (BNF).
- √ CFG = (Terminals, Nonterminals, Productions, Start Symbol)
- √ A program is valid if we may construct a parse tree, or a derivation from the grammar.
- √ Associativity and precedence of operations are part of the design of a CFG.
- √ Avoid ambiguous grammars by rewriting them or imposing parsing rules.