Syntactic Analysis (Parsing)

José Miguel Rivero

rivero@cs.upc.edu

Barcelona School of Informatics (FIB)

Universitat Politècnica de Catalunya BarcelonaTech (UPC)

FIB

Summary

- Objectives of Syntax Analysis
- Context Free Grammars. Applications
- Parsing in Compilers / Interpreters
- Syntax vs. Semantics
- Derivation. Parse Tree
- Cocke-Younger-Kasami Parsing Algorithm
- Parse Tree and Abstract Syntax Tree (AST)
- Ambiguous Grammars
- Linear Parsing Algorithms
  - Top-down $LL(1)$ parsers
  - Bottom-up $LR(1)$ parsers

FIB

Objectives of Syntax Analysis

- Given a CFG (Context Free Grammar) G, with start symbol S, and a word w (a sequence of tokens), can w be generated by G ?

$$w \in \mathcal{L}(G) \; ? \qquad S \Rightarrow^* w \; ?$$

- Analyze the sequence of tokens to determine their grammatical structure with respect to G.
  Compute the *parse tree* corresponding to the input
- Detect, diagnose, and recovery from *syntax errors*
- Accepts some invalid constructs, filtered out by the semantic analysis

FIB

Context Free Grammars

- Expressive power
  - $\mathcal{L}_1 = \{\ a^n\, b^n \mid n \geq 0\ \}$
  - $\mathcal{L}_2 = \{\ w\, c\, w^{-1} \mid w \in (a|b)^*\ \}$
  - $\mathcal{L}_3 = \{\ w\, c\, w \mid w \in (a|b)^*\ \}$
  - $\mathcal{L}_4 = \{\ a^n\, b^m\, c^n\, d^m \mid n, m \geq 0\ \}$
  - Algebraic expressions involving numbers, operations $+$ and $*$, and left and right parentheses
  - Constructions of programming languages such as declarations, statements (assignment, `if`, `while`, ...), expressions, etc.

FIB

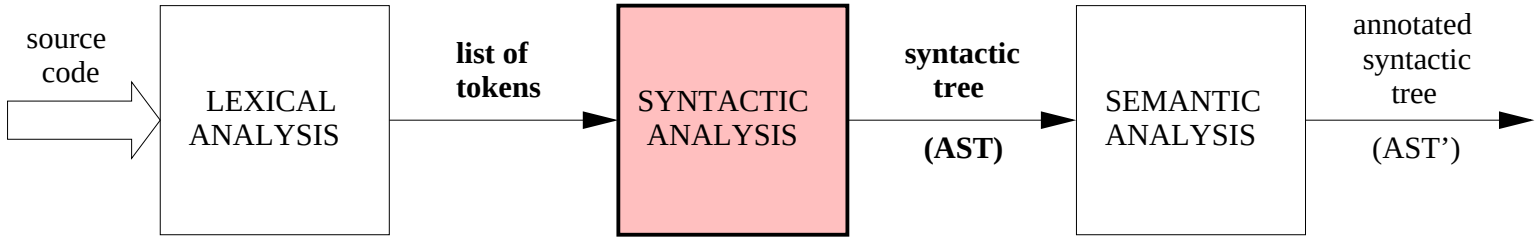
Context Free Grammars

- Applications
  - Natural language processing (NLP)
  - Type-setting languages (nroff, postscript), document and extensible markup languages (LaTeX, SGML, XML, ...)
  - Query for databases and information systems (SQL, DataLog, LDAP, ...)
  - Logical synthesis and simulation of electronic circuits (VHDL)
  - Graphical or modelling languages (UML, Energy Systems Language)
  - Many applications come with their built-in language. For example, the scripting language for Adobe Flash (ActionScript)

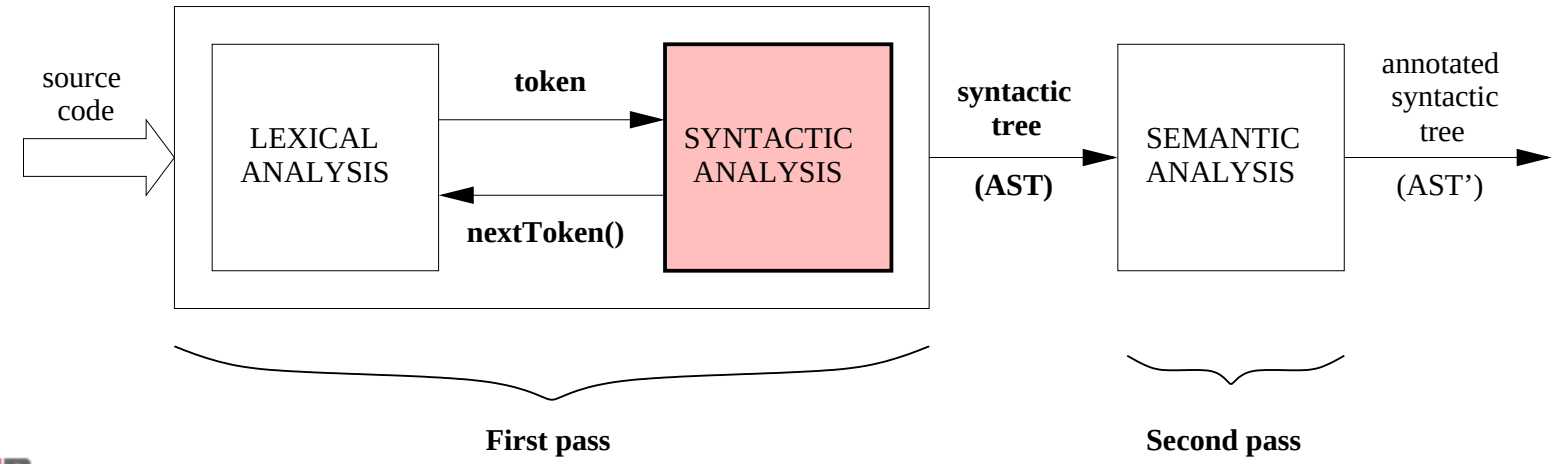
FIB

Parsing in Compilers / Interpreters

Conceptual structure:

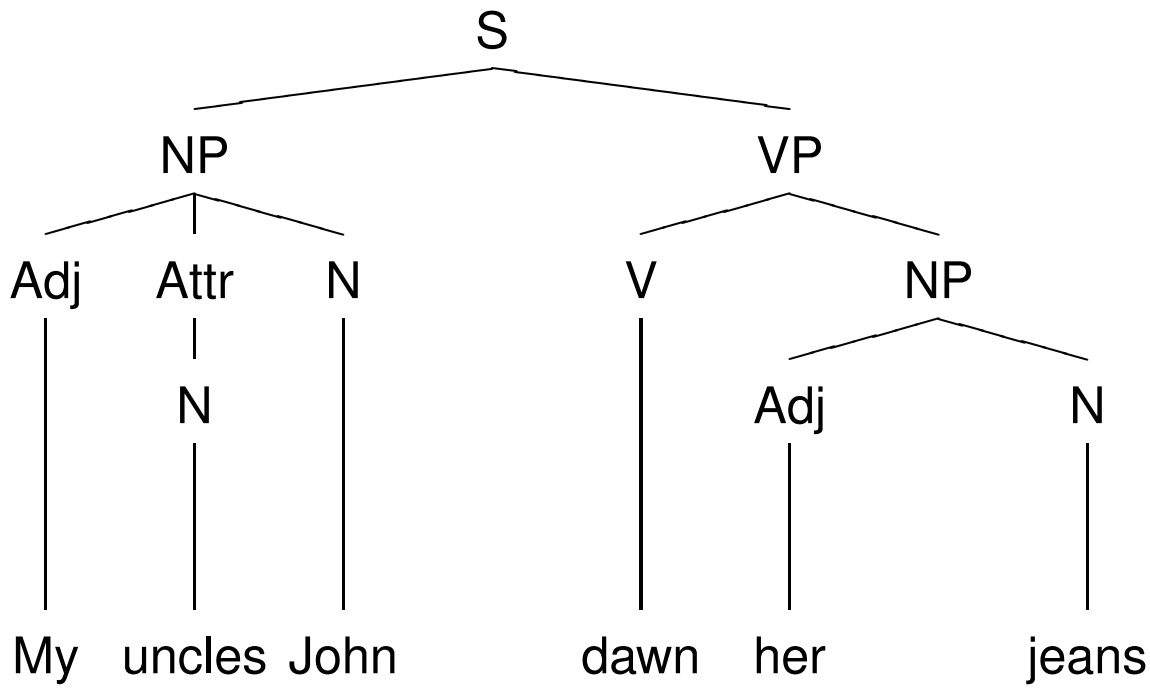


Usual structure:



Syntax vs. Semantics

- “My uncles John dawn her jeans”



- “Marketplace for : input the a .”
- “They are hunting dogs”

Syntax vs. Semantics

- Programming languages.

  Syntax errors:

```
public class {
    int static i;
    boolean j;
    public double i(UnknownClass k {
        i..x = "hello";
        if (i / 2)
            i + 1 = j;
        return i > ;
    }
    }
}
```

Syntax vs. Semantics

- Programming languages.

  Semantic errors:

```
public class MyFirstClass {
    static int i;
    boolean j;
    public double i(UnknownClass k) {
        i.x[10] = "hello";
        if (i / 2)
            i + 1 = j;
        return i > 3.14;
    }
}
```

FIB

Syntax vs. Semantics

- Programming languages.

  Ambiguity:

```
class Bar {
    public float foo( ) {
        return 1;
    }

    public int foo( ) {
        return 2;
    }
};

float x = Bar::foo( );
```

FIB

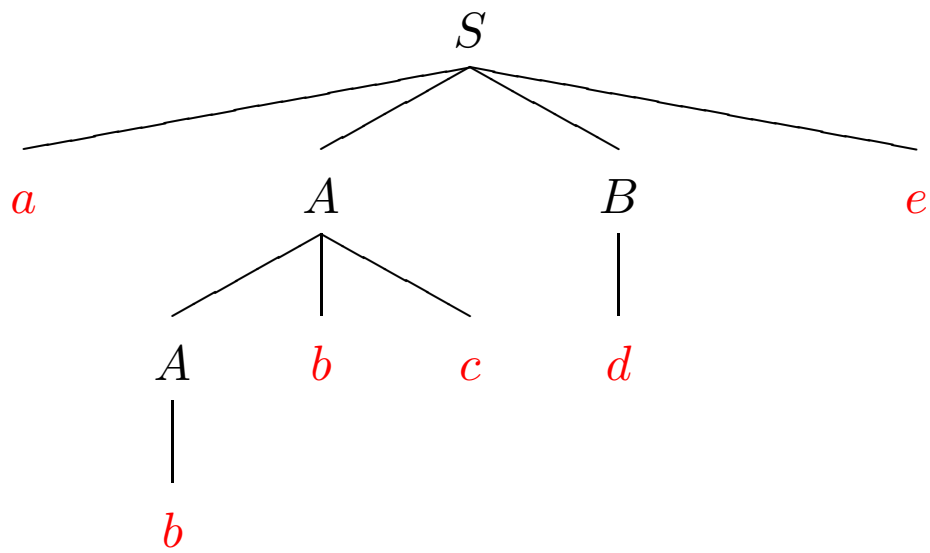
Derivation. Parse Tree

Grammar G :

$$\begin{cases} S \to aABe \\ A \to Abc \mid b \\ B \to d \end{cases} \qquad w = abbcde$$

$$\boxed{S \Rightarrow aABe \Rightarrow aAbcBe \Rightarrow abbcBe \Rightarrow abbcde = w}$$

Parse tree:



Cocke-Younger-Kasami Algorithm

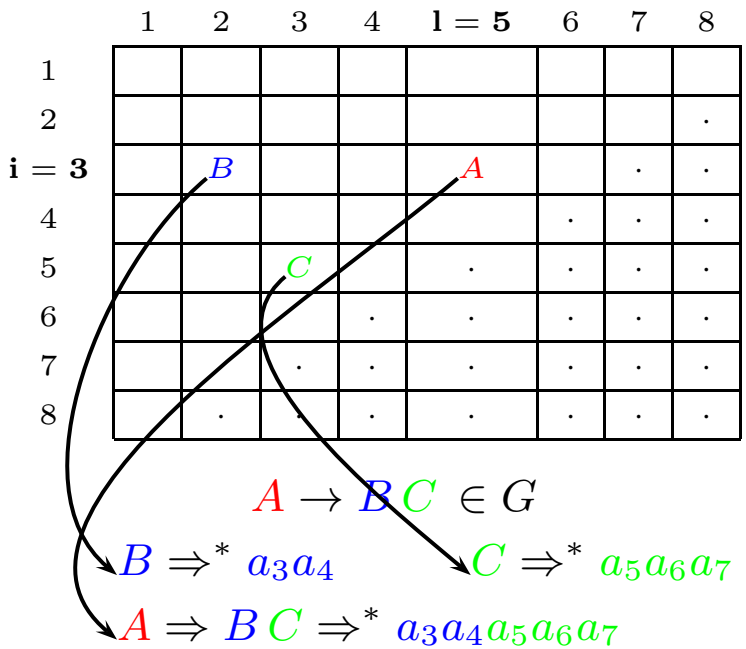
Input : A CFG in *Chomsky Normal Form* G, and a word $w = a_1 \dots a_n$

Only rules of the form $A \to a$ or $A \to B\,C$

Output : An $n \times n$ table T, where $T[i,\ l]$ is the set of nonterminals A

that generate the substring $a_i \dots a_{i+l-1}$

Example : $w = a_1 a_2 \dots a_8$



Cocke-Younger-Kasami Algorithm

Input : A CFG in *Chomsky Normal Form* G, and a word $w = a_1 \dots a_n$

Only rules of the form $A \to a$ or $A \to B\,C$

Output : An $n \times n$ table T, where $T[i,\ l]$ is the set of nonterminals A

that generate the substring $a_i \dots a_{i+l-1}$

Algorithm :

for all $i \in [1\,..\,n]$: $T[i,\ 1] := \{A \mid A \to a_i \in G\}$

for all $l \in [2\,..\,n]$:

  for all $i \in [1\,..\,n-l]$:

    for every $A \to B\,C \in G$:

      if there is a $l_1 \in [1\,..\,l-1]$ such that

        $B \in T[i,\ l_1]$ and $C \in T[i+l_1,\ l-l_1]$

      then

        $T[i,\ l] := T[i,\ l] \cup \{A\}$

Using dynamic programming, each table entry can be filled in $O(n)$ time.

The algorithm runs in $O(n^3)$ time.

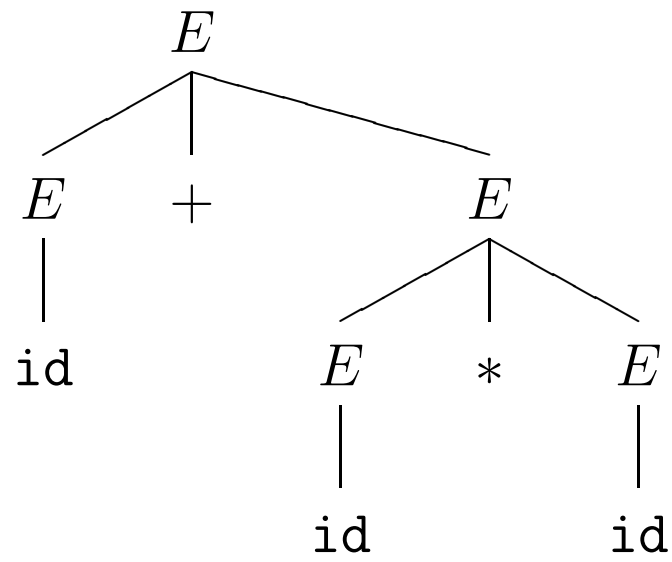
Parsing expressions

$G: \quad E \rightarrow E + E \mid E * E \mid (E) \mid \texttt{id}$ $\qquad\qquad$ $w = id_1 + id_2 * id_3$

(Leftmost) Derivation #1 :

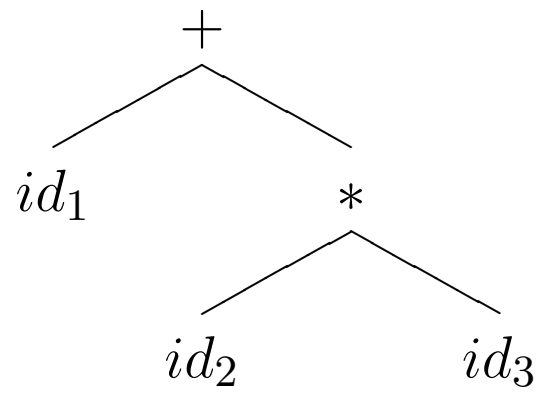
$E \Rightarrow_{ld} E + E \Rightarrow_{ld} \texttt{id} + E \Rightarrow_{ld} \texttt{id} + E * E \Rightarrow_{ld} \texttt{id} + \texttt{id} * E \Rightarrow_{ld} \texttt{id} + \texttt{id} * \texttt{id}$

Parse Tree:



$id_1 + (id_2 * id_3)$

Abstract Syntax Tree (AST):



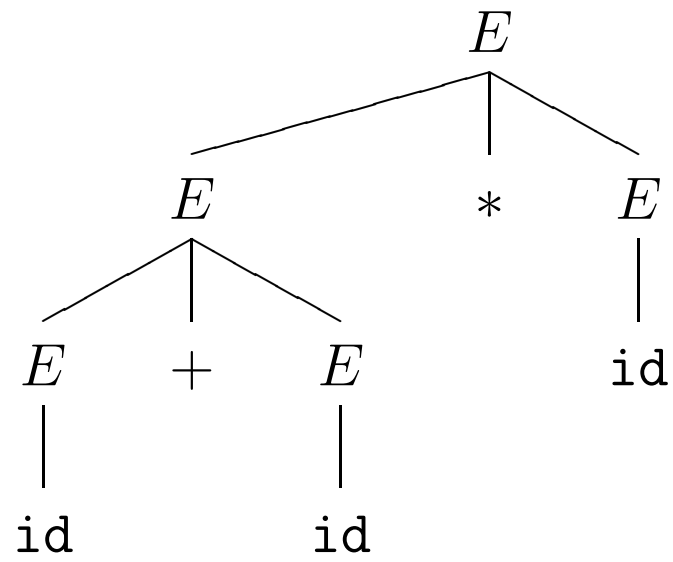
Parsing expressions

$G: \quad E \rightarrow E + E \mid E * E \mid (E) \mid \texttt{id}$ $\qquad\qquad$ $w = id_1 + id_2 * id_3$

(Leftmost) Derivation #2 :

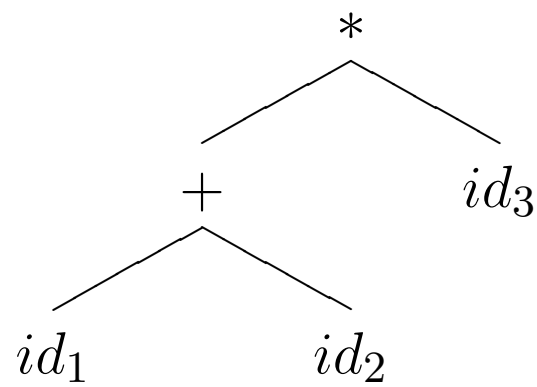
$E \Rightarrow_{ld} E*E \Rightarrow_{ld} E+E*E \Rightarrow_{ld} \texttt{id}+E*E \Rightarrow_{ld} \texttt{id}+\texttt{id}*E \Rightarrow_{ld} \texttt{id}+\texttt{id}*\texttt{id}$

Parse Tree:



$(id_1 + id_2) * id_3$

Abstract Syntax Tree (AST):



Ambiguous grammar

A grammar G is said to be *ambiguous* if there is some string w that has more than one parse tree or more than one leftmost derivation.

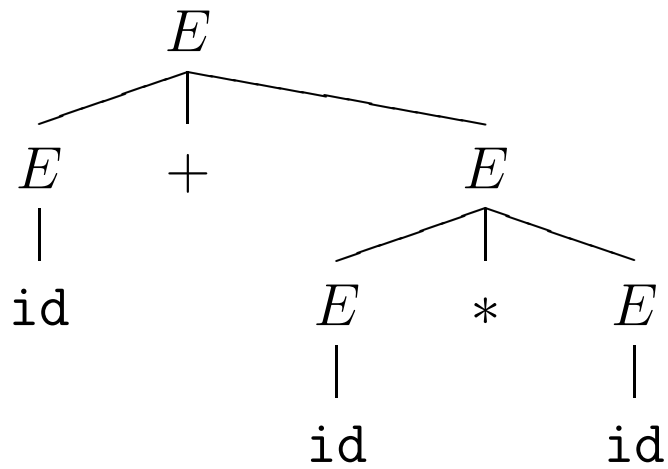
FIB

Ambiguous grammar

$G: \quad E \rightarrow E + E \mid E * E \mid (E) \mid \texttt{id}$

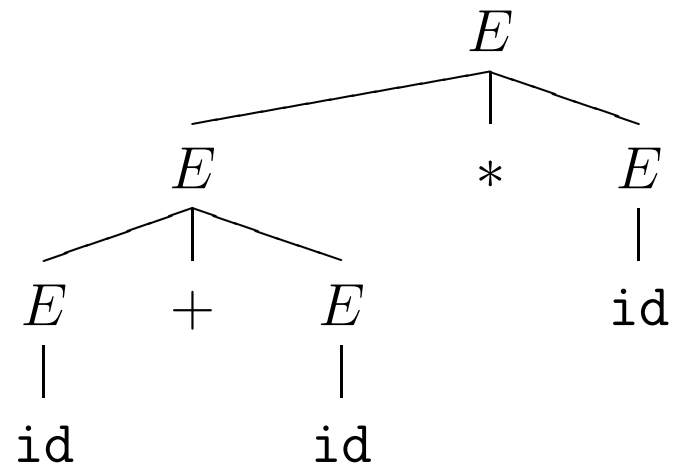
$w = id_1 + id_2 * id_3$

Parse Tree A:



$id_1 + (id_2 * id_3)$

Parse Tree B:



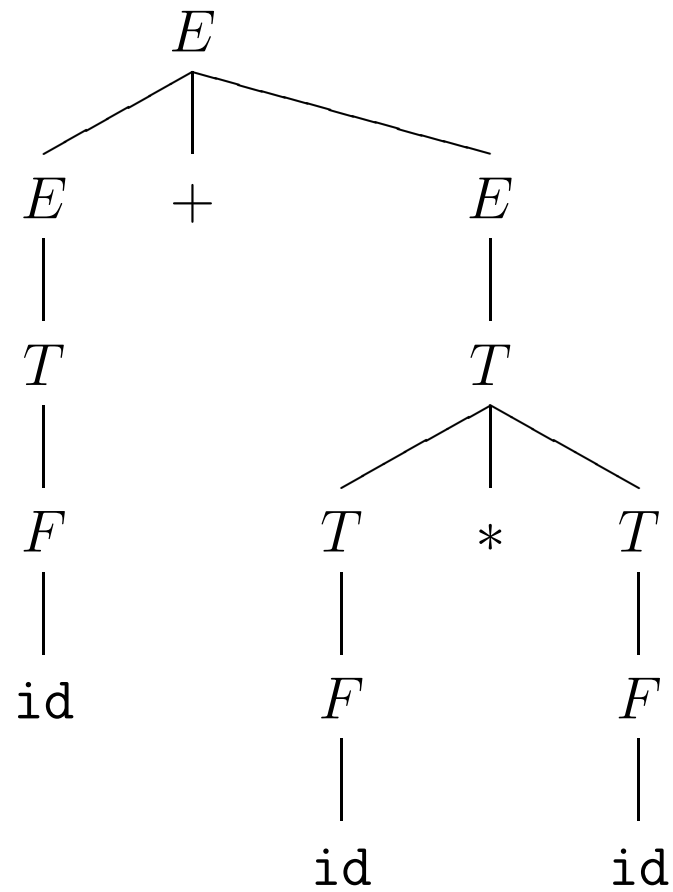
$(id_1 + id_2) * id_3$

Operator's precedence

- $id_1 + (id_2 * id_3 * id_4) + id_5$
- Precedence: $+ \prec_p *$
- This other grammar G' reflects the operator's precedence:

$$E \to E + E \mid T$$
$$T \to T * T \mid F$$
$$F \to (E) \mid \texttt{id}$$

- Now $w = id_1 + id_2 * id_3$ only has this parse tree:



$$id_1 + (id_2 * id_3)$$

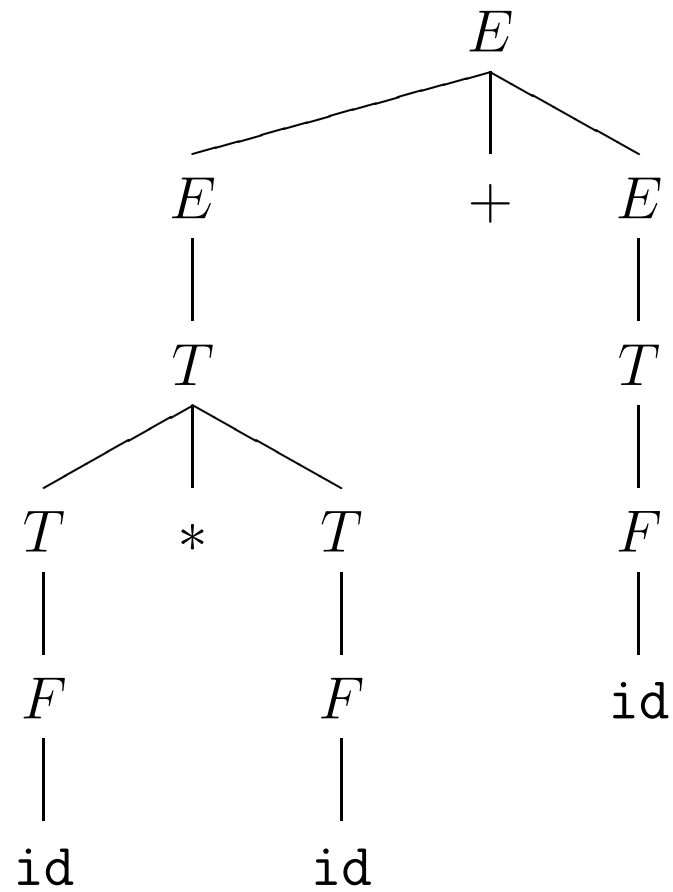
Operator's precedence

- $id_1 + (id_2 * id_3 * id_4) + id_5$
- Precedence: $+ \prec_p *$
- This other grammar G' reflects the operator's precedence:

$$E \to E + E \mid T$$
$$T \to T * T \mid F$$
$$F \to (E) \mid \texttt{id}$$

- Now $w = id_1 * id_2 + id_3$ only has this parse tree:

$$(id_1 * id_2) + id_3$$



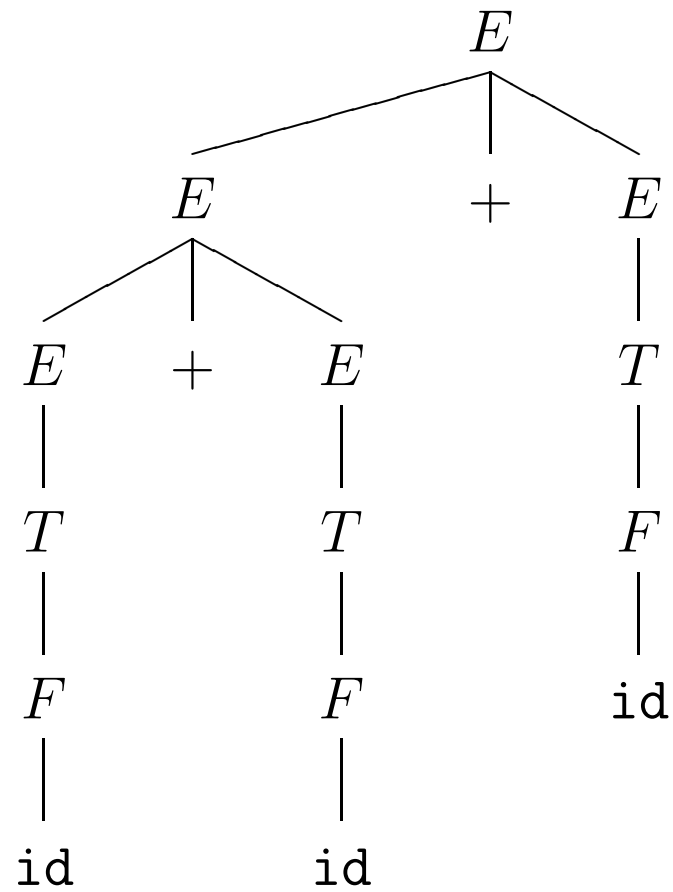
Operator's precedence

- $id_1 + (id_2 * id_3 * id_4) + id_5$
- Precedence: $+ \prec_p *$
- This other grammar G' reflects the operator's precedence:

$$E \to E + E \mid T$$
$$T \to T * T \mid F$$
$$F \to (E) \mid \texttt{id}$$

- But $w = id_1 + id_2 + id_3$ still has two parse trees:

$$(id_1 + id_2) + id_3$$



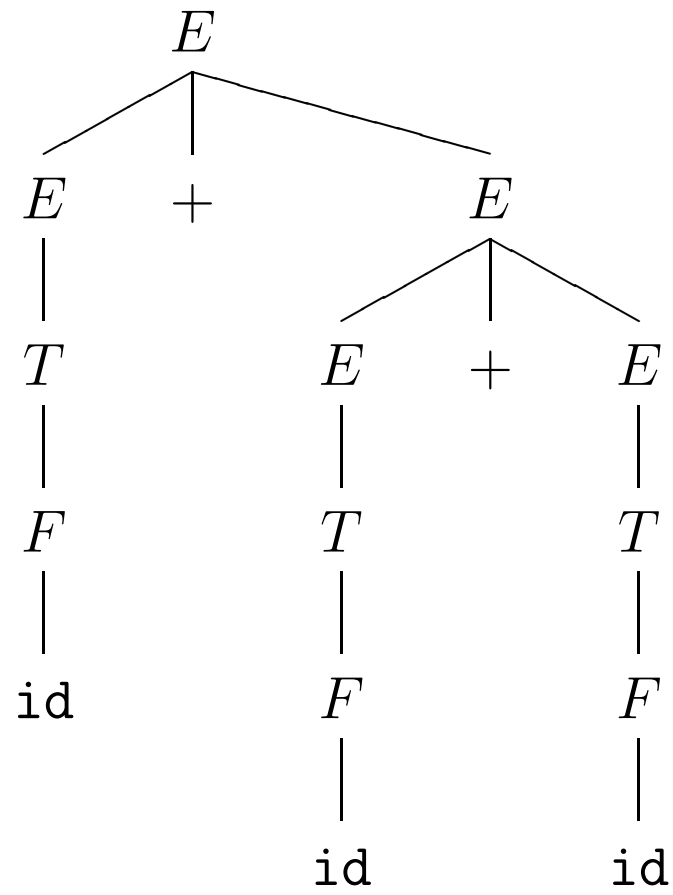
Operator's precedence

- $id_1 + (id_2 * id_3 * id_4) + id_5$
- Precedence: $+ \prec_p *$
- This other grammar G' reflects the operator's precedence:

$$E \to E + E \mid T$$
$$T \to T * T \mid F$$
$$F \to (E) \mid \texttt{id}$$

- <u>But</u> $w = id_1 + id_2 + id_3$ still has two parse trees:

$$id_1 + (id_2 + id_3)$$



Operator's associativity

Left-associative:

$e_1 \otimes e_2 \otimes e_3$ means $(e_1 \otimes e_2) \otimes e_3$

Right-associative:

$e_1 \otimes e_2 \otimes e_3$ means $e_1 \otimes (e_2 \otimes e_3)$

Non-associative:

$e_1 \otimes e_2 \otimes e_3$ is not correct

Usually arithmetic ($+$, $-$, $*$, $/$) and logical operators (`&&`, `||`) are left-associative, comparison operators ($>$, $=$,$<$, ...) are non-associative, and exponentiation (`^`) is right-associative.

FIB

Operator's associativity

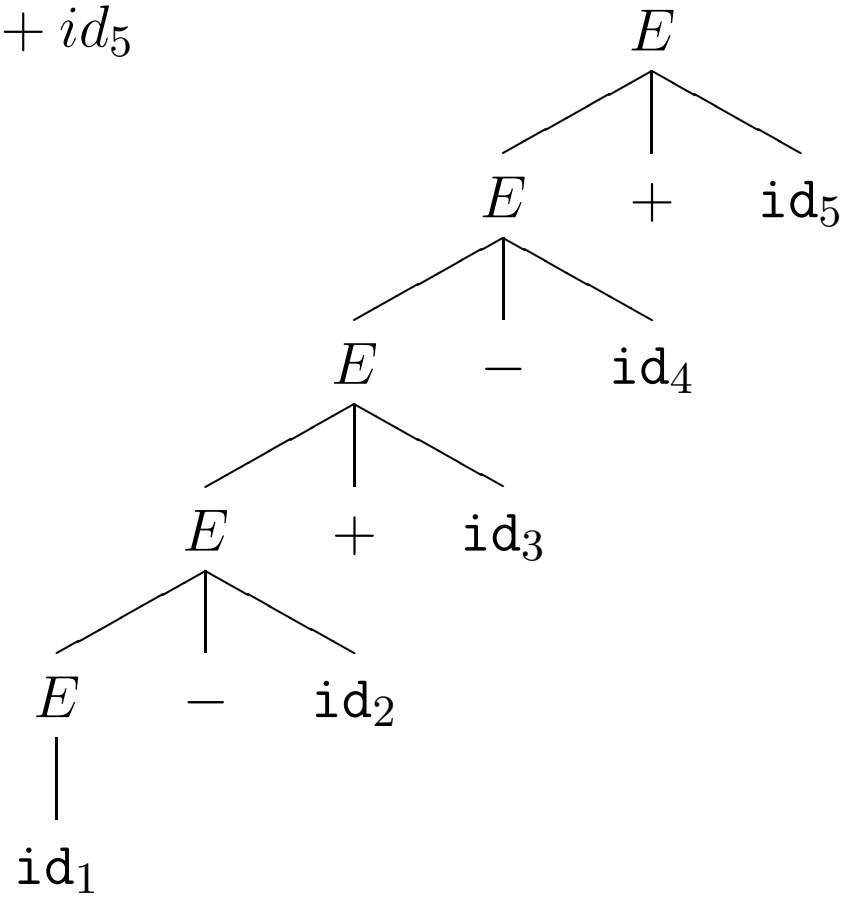
$$(((id_1 - id_2) + id_3) - id_4) + id_5$$

id_1 : E

$id_1 - id_2$: E

$id_1 - id_2 + id_3$: E

$id_1 - id_2 + id_3 - id_4$: E



Operator's associativity

Left-associativity ($+$, $-$):

$$\begin{array}{rcl} E & \rightarrow & E + \texttt{id} \\ & | & E - \texttt{id} \\ & | & \texttt{id} \end{array}$$

Right-associativity ($+$, $-$):

$$\begin{array}{rcl} E & \rightarrow & \texttt{id} + E \\ & | & \texttt{id} - E \\ & | & \texttt{id} \end{array}$$

Non-associativity ($>$):

$$\begin{array}{rcl} E & \rightarrow & T > T \mid T \\ T & \rightarrow & \texttt{id} \mid \texttt{num} \end{array}$$

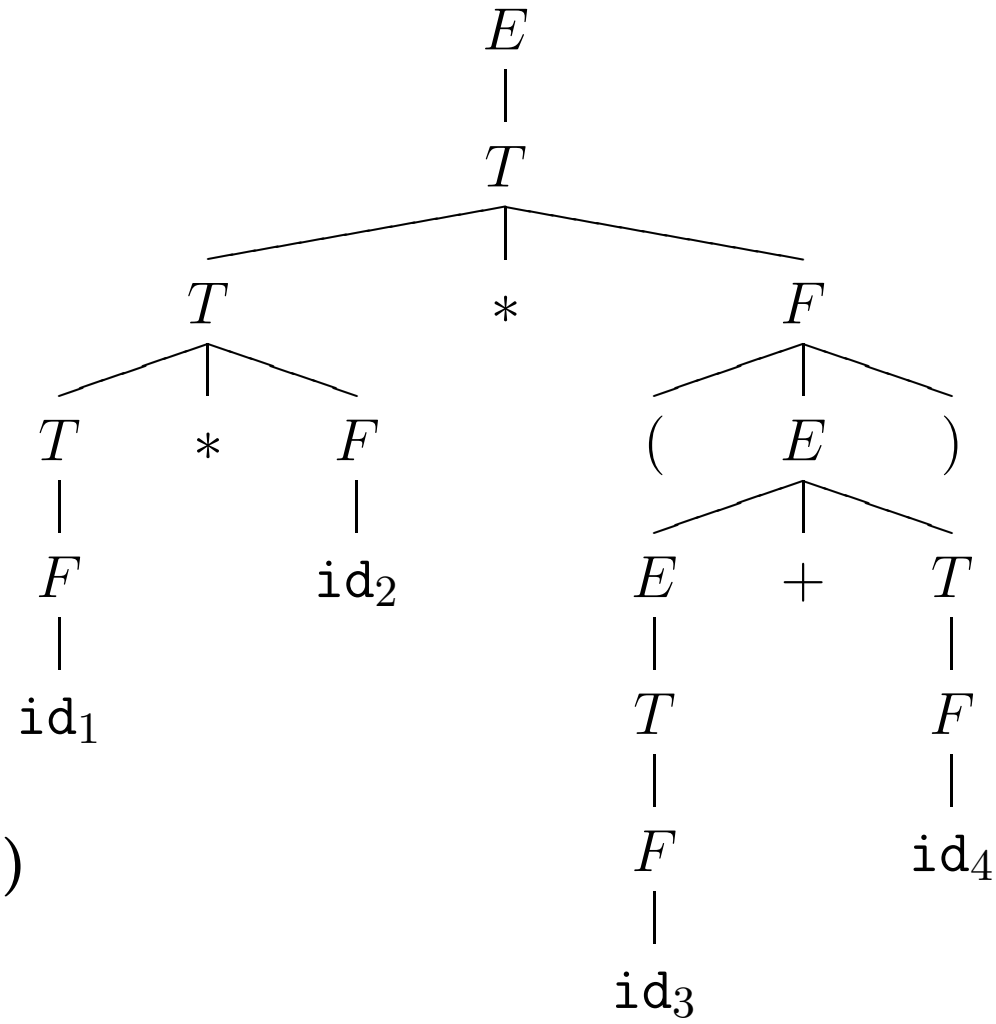
Left-associativity ($+$, $*$):

$$\begin{array}{rcl} E & \rightarrow & E + T \\ & | & T \\ T & \rightarrow & T * F \\ & | & F \\ F & \rightarrow & (E) \\ & | & \texttt{id} \end{array}$$

Grammar for expressions

Grammar G:

$$
\begin{array}{rcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{id}
\end{array}
$$



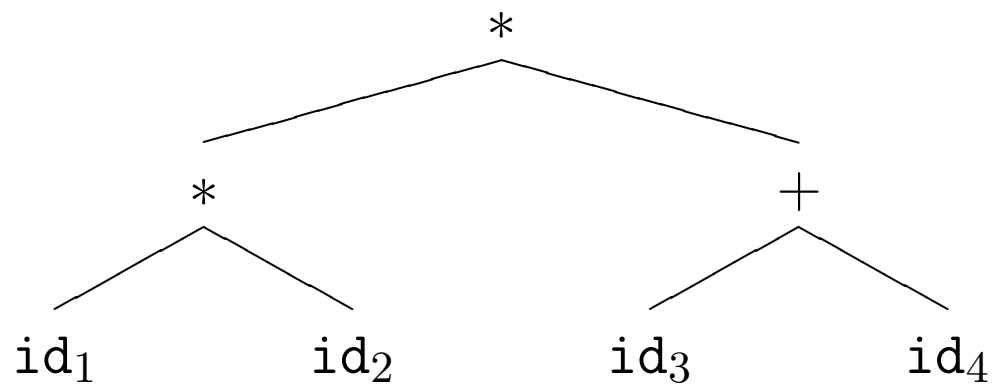
$$w = \underbrace{id_1 * id_2} * (\underbrace{id_3 + id_4})$$

Grammar for expressions

Grammar G:

$$
\begin{array}{rcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{id}
\end{array}
$$

Abstract Syntax Tree (AST) :



$$w = \underbrace{id_1 * id_2} * (\underbrace{id_3 + id_4})$$

Exercises

- Extend the grammar for expressions with the following operators and precedence:
  $$\{>, <, =\} \prec_p \{+, -\} \prec_p \{*, /\} \prec_p \{-_u\}$$
  where $-_u$ denotes unary minus ($-e_1$).
  All the binary operators are left-associative.
- Extend the previous grammar incorporating the access-to-array operator $[\,]$ (with the usual syntax ($e_1[e_2]$), the access-to-struct operator ., and the postfix access-to-pointed operator ^ (e_1^).
  All of them have maximal precedence.

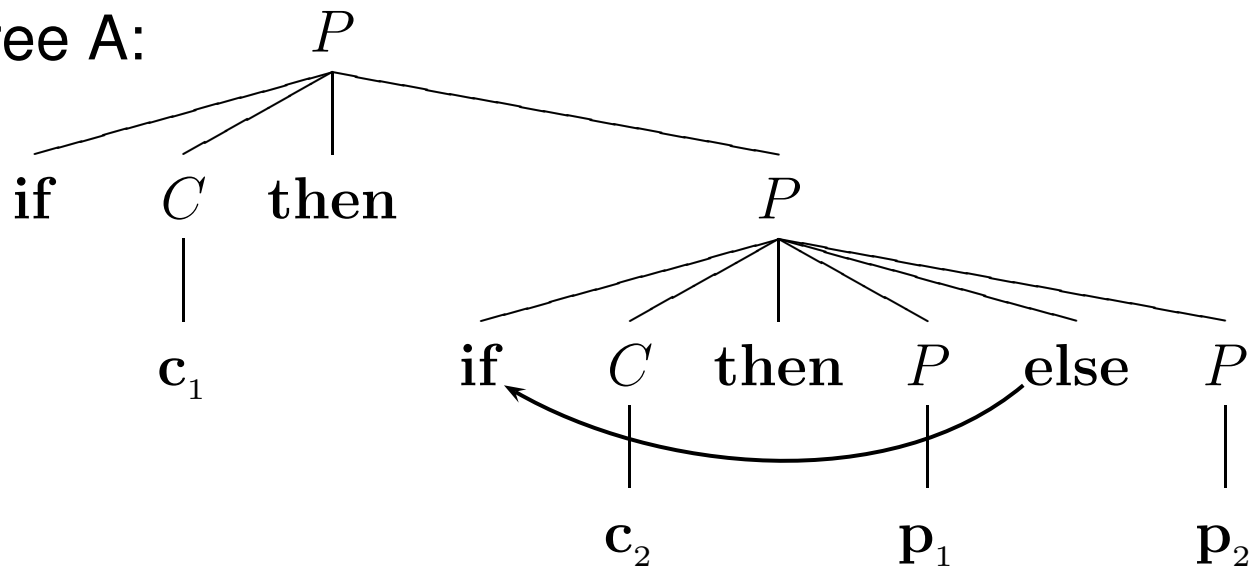
FIB

Ambiguous If-then-else (dangling else)

Grammar: $w = \textbf{if } \mathbf{c_1} \textbf{ then if } \mathbf{c_2} \textbf{ then } \mathbf{p_1} \textbf{ else } \mathbf{p_2}$

$$
\begin{aligned}
P &\rightarrow \textbf{if } C \textbf{ then } P \\
&\mid \textbf{if } C \textbf{ then } P \textbf{ else } P \\
&\mid \mathbf{p} \\
C &\rightarrow \mathbf{c}
\end{aligned}
$$

Parse Tree A:

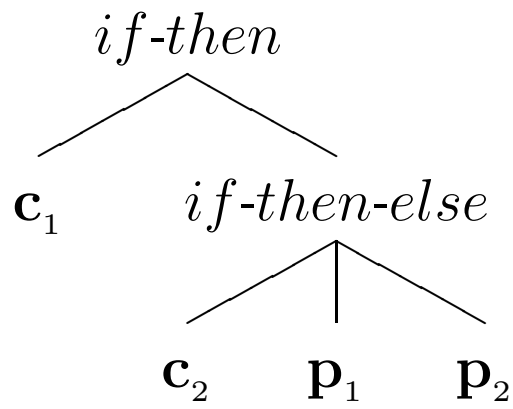


Ambiguous If-then-else (dangling else)

Grammar: $w = \mathbf{if}\ \mathbf{c_1}\ \mathbf{then}\ \mathbf{if}\ \mathbf{c_2}\ \mathbf{then}\ \mathbf{p_1}\ \mathbf{else}\ \mathbf{p_2}$

$$
\begin{array}{rcl}
P & \rightarrow & \mathbf{if}\ C\ \mathbf{then}\ P \\
 & | & \mathbf{if}\ C\ \mathbf{then}\ P\ \mathbf{else}\ P \\
 & | & \mathbf{p} \\
C & \rightarrow & \mathbf{c}
\end{array}
$$

Abstract Syntax Tree (AST) A:

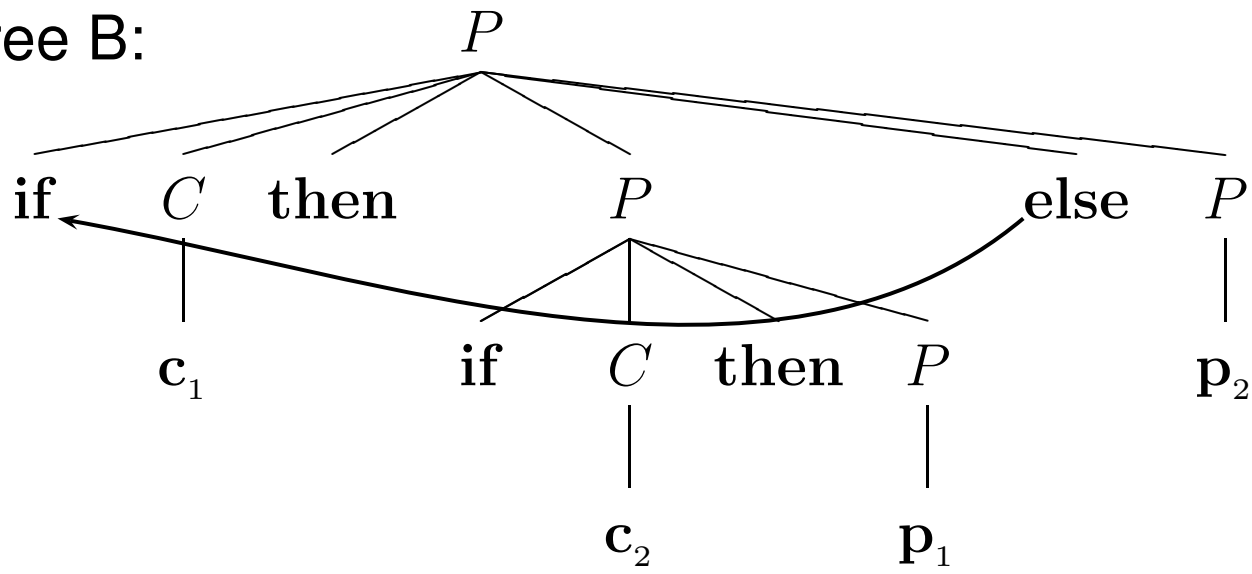


Ambiguous If-then-else (dangling else)

Grammar: $w = \textbf{if } \mathbf{c_1} \textbf{ then if } \mathbf{c_2} \textbf{ then } \mathbf{p_1} \textbf{ else } \mathbf{p_2}$

$$
\begin{aligned}
P &\rightarrow \textbf{if } C \textbf{ then } P \\
&\mid \textbf{if } C \textbf{ then } P \textbf{ else } P \\
&\mid \mathbf{p} \\
C &\rightarrow \mathbf{c}
\end{aligned}
$$

Parse Tree B:

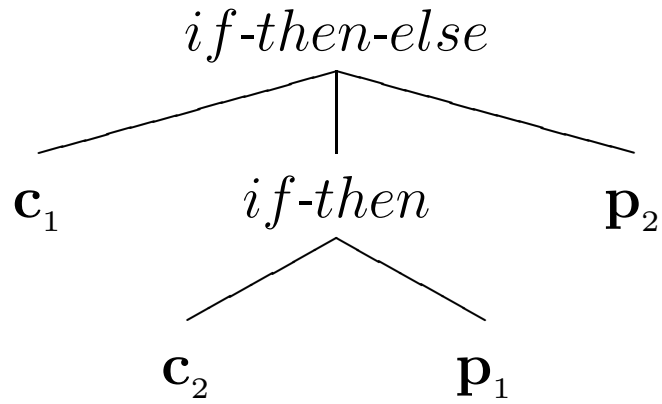


Ambiguous If-then-else (dangling else)

Grammar: $w = \textbf{if } \mathbf{c_1} \textbf{ then if } \mathbf{c_2} \textbf{ then } \mathbf{p_1} \textbf{ else } \mathbf{p_2}$

$$
\begin{array}{rcl}
P & \rightarrow & \textbf{if } C \textbf{ then } P \\
 & | & \textbf{if } C \textbf{ then } P \textbf{ else } P \\
 & | & \mathbf{p} \\
C & \rightarrow & \mathbf{c}
\end{array}
$$

Abstract Syntax Tree (AST) B:



Exercise

- Find an alternative non-ambiguous grammar for *this* language.
- Give a clue?
  There are two kind of propositions:
  - *closed*, where a following `else` cannot correspond to them.
  - *open*, where a following `else` will correspond to them (to their `if`).

FIB