Compilers

module of the course
“Professional English”

Yulia Burkatovskaya

Department of Computer Engineering

Associate professor

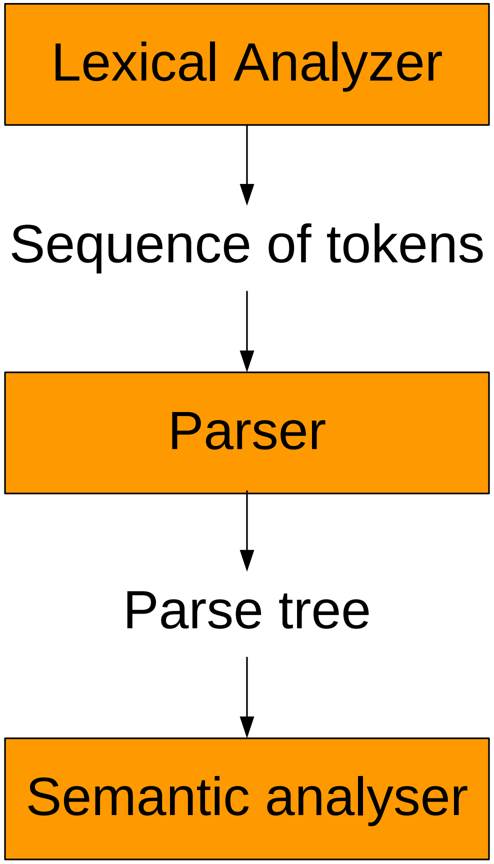
3. Parsing

- Basics of parsing
- Context-free grammar
- Parse tree
- Ambiguity
- Associativity and precedence
- Left-recursion
- Error-recovery

3.1. Basics of parsing

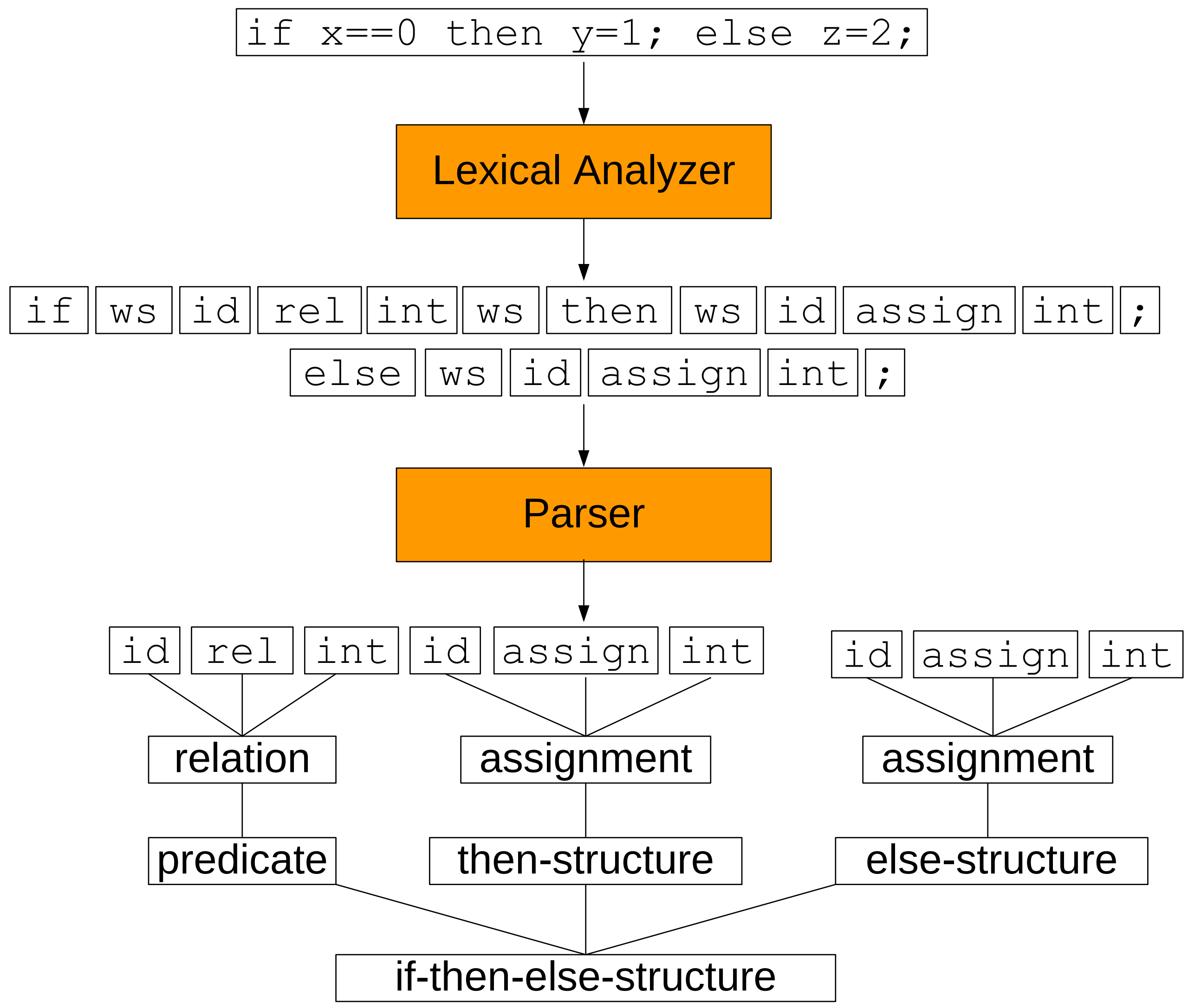
Syntax analysis tasks:

- To understand the meaning of the sequence of tokens
- To distinguish between valid and invalid strings of tokens
- To construct a parse tree for the semantic analyzer



Basics of parsing

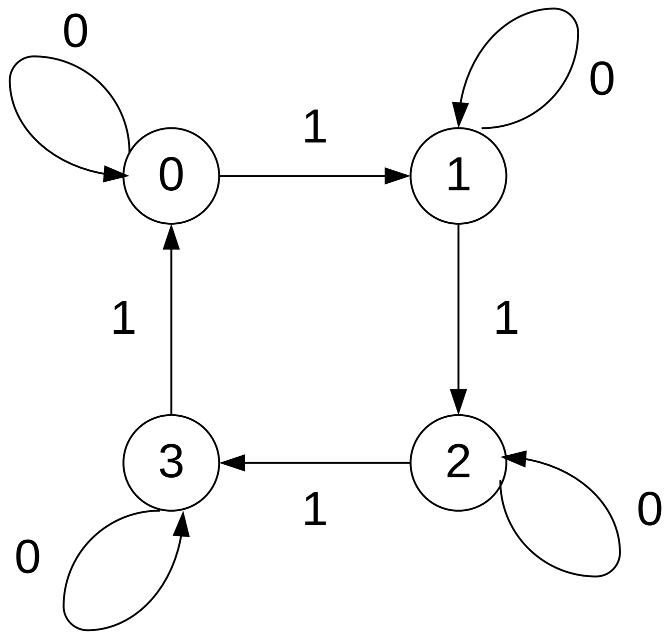
Example



Basics of parsing

- **Regular languages**

Counting “mod k”.



- **Context-free languages**

Recursive structures.

```
if …
    then …
        if …
            then …
                …
            else …
    else …
```

Basics of parsing

Not every string of tokens is a program!

- **Regular expressions** describe tokens.
- **Context-free grammars** describe valid strings of tokens.

3.2. Context-free grammars

Context-free grammar G=<A,T,N,P>

- A – a *start symbol*;
- T – a set of *terminals*;
- N – a set of *non-terminals*;
- P – a set of *productions*.

Production: $X \rightarrow Y1Y2\ldots Yk$;

$X \in N$;

$Yi \in T \cup N \cup \{\varepsilon\}$.

A set of productions: $X \rightarrow S1 \mid S2 \mid \ldots \mid Sj$.

Context-free grammars

- Example (if-else structure)

EXP → if EXP then ST;

EXP → if EXP then ST; else ST;

EXP → ID

EXP → ID COM ID

EXP → ID COM INT

ST → ID = ID | ID = INT

COM → == | < | <= | > | >=

ID → LET | ID LET | ID DIG

INT → DIG | INT DIG

LET → a | … | z | A | … | Z

DIG → 0 | … | 9

- EXP=EXPRESSION
- ST=STATEMENT
- ID=IDENTIFIER
- COM=COMPARISON
- INT=INTEGER
- LET=LETTER
- DIG=DIGIT

Context-free grammars

Derivation

Let G=<A,T,N,P> be a CFG and:

- $S_1S_2\ldots S_k\ldots S_n \in (T \cup N \cup \{\varepsilon\})^*$;
- $S_k \rightarrow Y_1\ldots Y_j \in P$,

then $S_1S_2\ldots S_k\ldots S_n \rightarrow S_1S_2\ldots Y_1\ldots Y_j \ldots S_n$ is *a step of derivation*.

$\alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_n$: α_n derives from α_0 in n steps ($\alpha_0 \rightarrow^* \alpha_n$) .

Context-free grammars

EXP → if EXP then ST;

EXP → if EXP then ST; else ST;

EXP → ID

EXP → ID COM ID

EXP → ID COM INT

ST → ID = ID | ID = INT

COM → == | < | <= | > | >=

ID → LET | IDLET | IDDIG

INT → DIG | INTDIG

LET → a | … | z | A | … | Z

DIG → 0 | … | 9

EXP → if EXP then ST; → if if EXP then ST; else ST; then ST; →

if if ID COM INT then ST; else ST; then ST; →

if if ID COM INT then ST; else ST; then ID = ID; → …

if if x1 <= 12 then x2 = 23; else y = z; then z = x2;

Context-free grammars

- Leftmost derivation: the leftmost non-terminal is always chosen to replace.

EXP → if EXP then ST; → if if EXP then ST; else ST; then ST; →

if if ID COM INT then ST; else ST; then ST; → ….

- Rightmost derivation: the rightmost non-terminal is always chosen to replace.

EXP → if EXP then ST; → if EXP then ID = ID; →

if EXP then ID = IDDIG; → if EXP then ID = ID2;

Context-free grammars

$L(G)=\{\alpha\colon \alpha \in T^* \cup \{\varepsilon\},\ A \rightarrow^* \alpha\}$ – the **language generated by G** (context-free language).

G(A,T,N,P) and G'(A',T,N',P') are **equivalent** if they generate the same language, i.e. L(G)=L(G').

Implementing tools are sensitive to grammar.

$E \rightarrow (E)$

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow ID$

$E \rightarrow T$

$T \rightarrow E+T$

$T \rightarrow E*T$

$T \rightarrow ID$

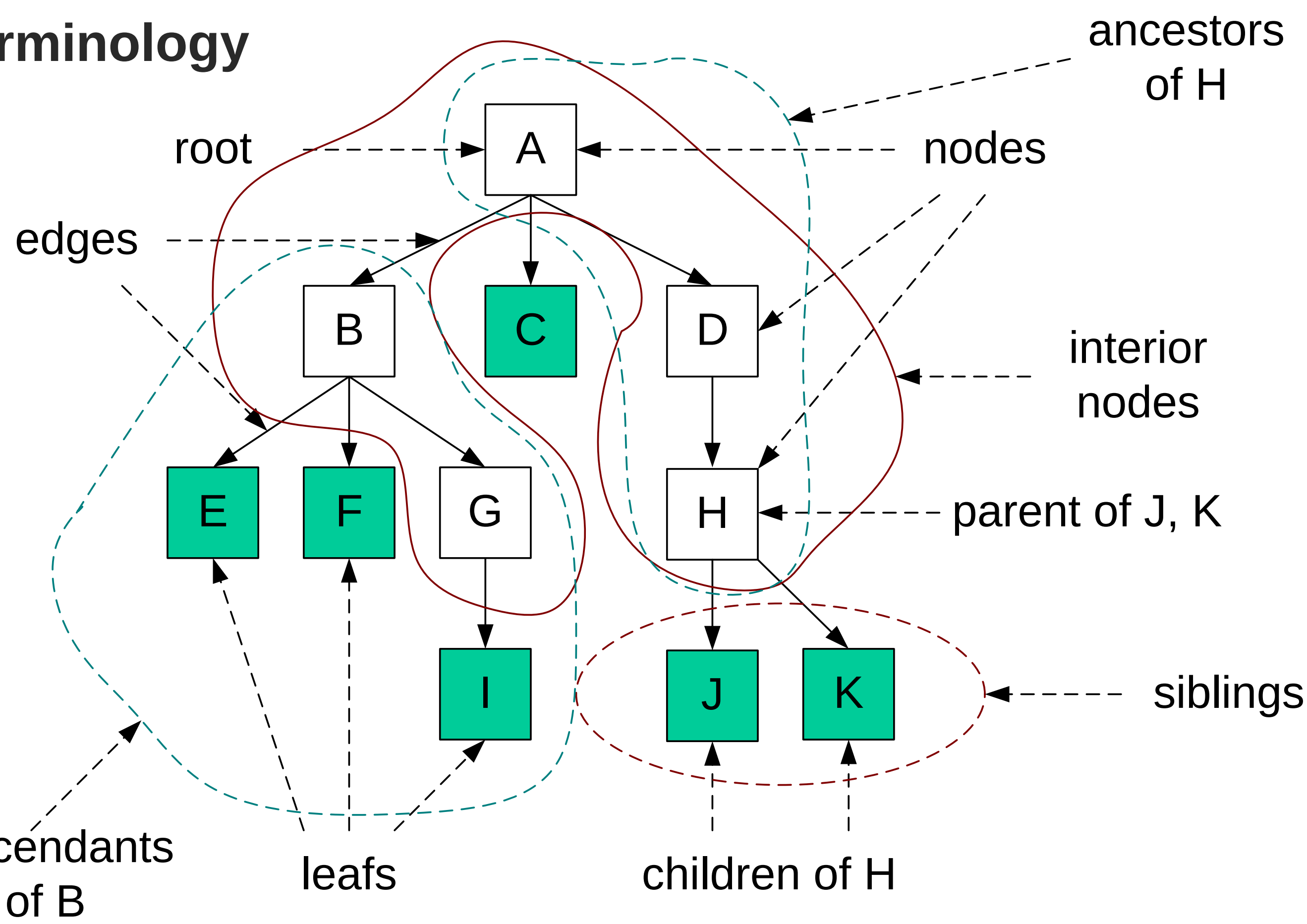
$T \rightarrow (E)$

CFG vs Rexp

- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
- The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
- Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
- More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

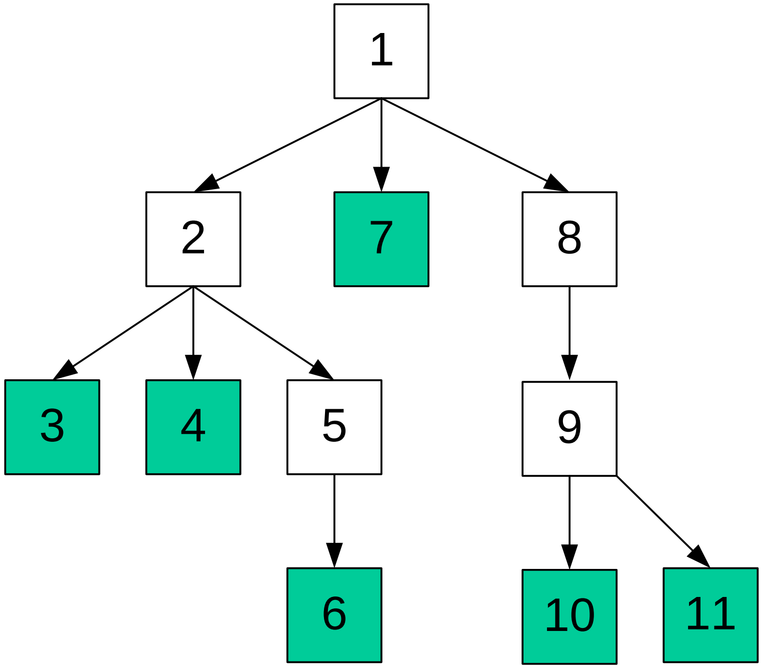
3.3. Parse tree

Tree terminology

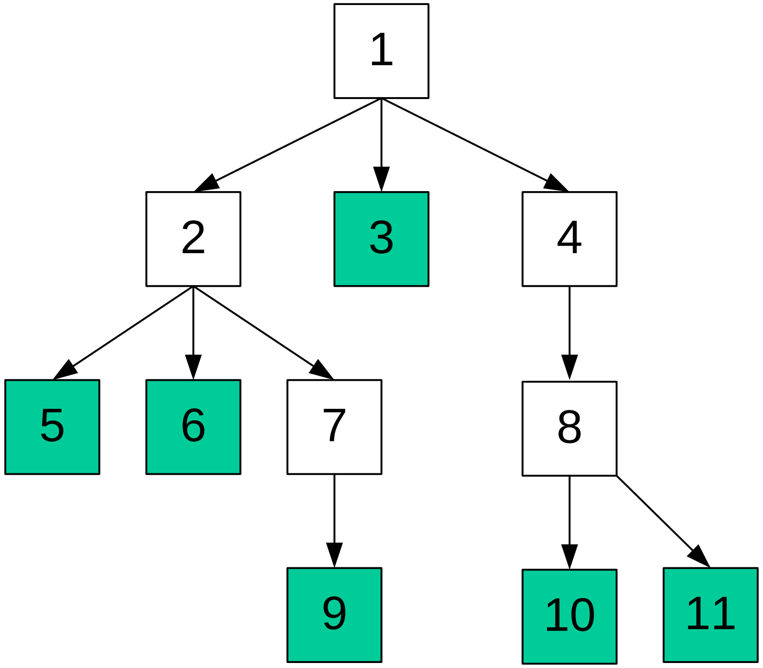


Parse tree

depth-first search



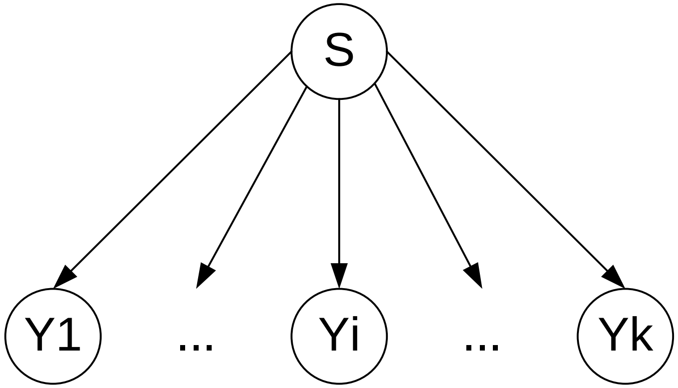
breadth-first search



Parse tree

Parse tree is a graphical representation of a derivation.

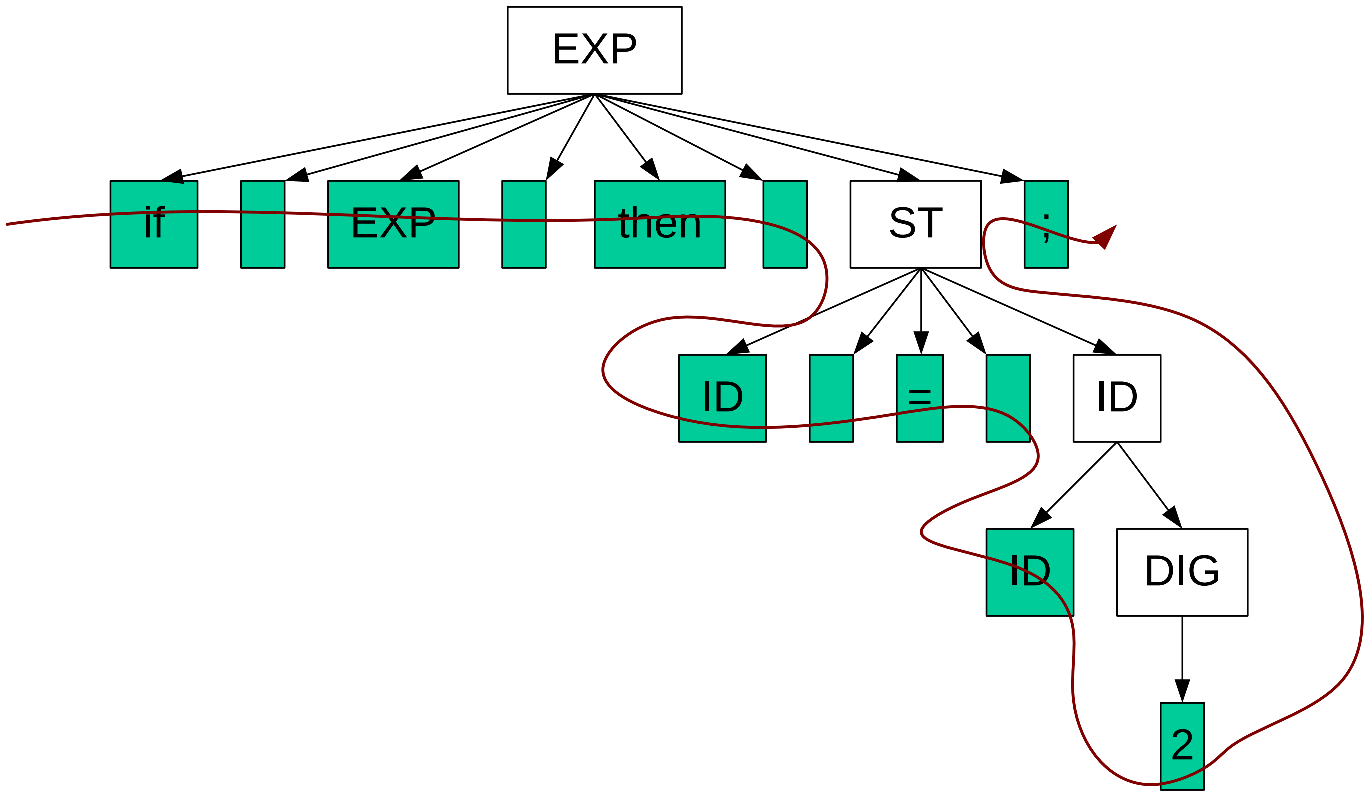
$S \rightarrow Y1...Yi...Yk$



- The **root** is labeled by the start symbol.
- **Leaves** are labeled by terminals. These labels read from left to right constitute a sequential form called the **yield** or **frontier** of the tree.

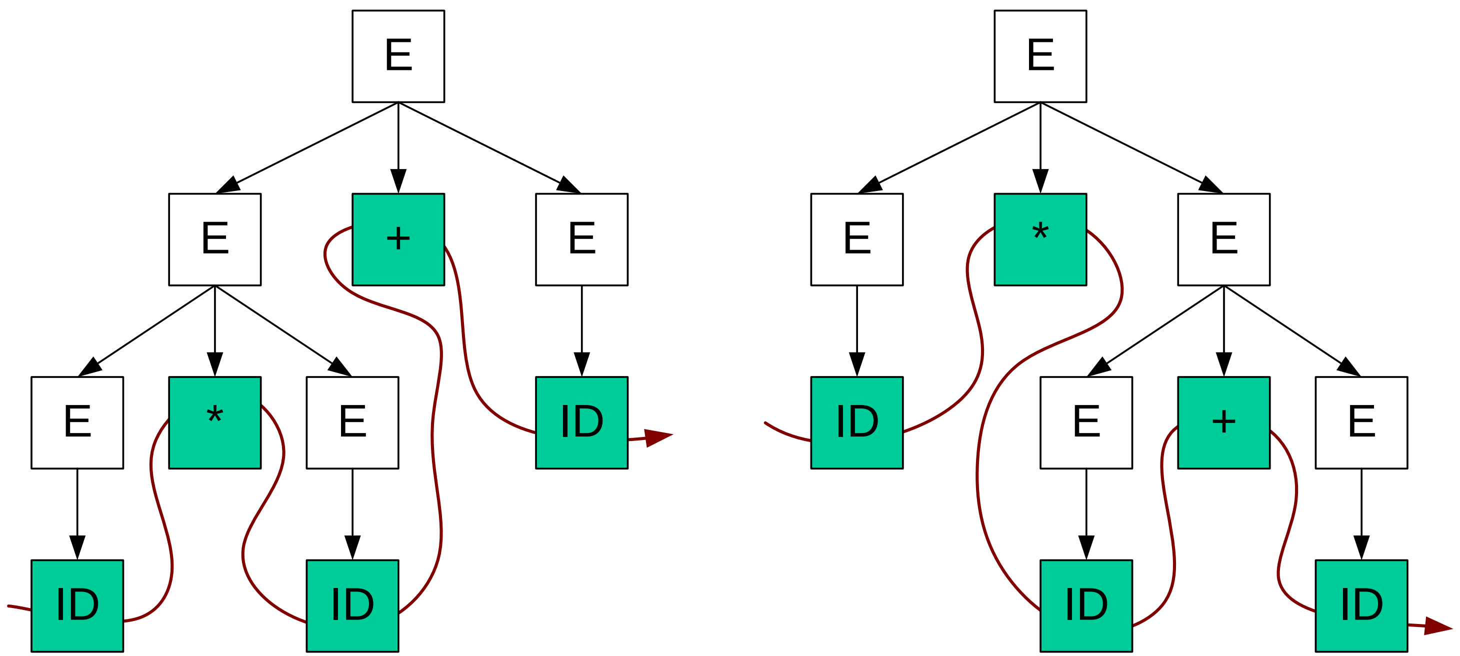
Parse tree

EXP → if EXP then ST; → if EXP then ID = ID; → if EXP then ID = IDDIG; → if EXP then ID = ID2;



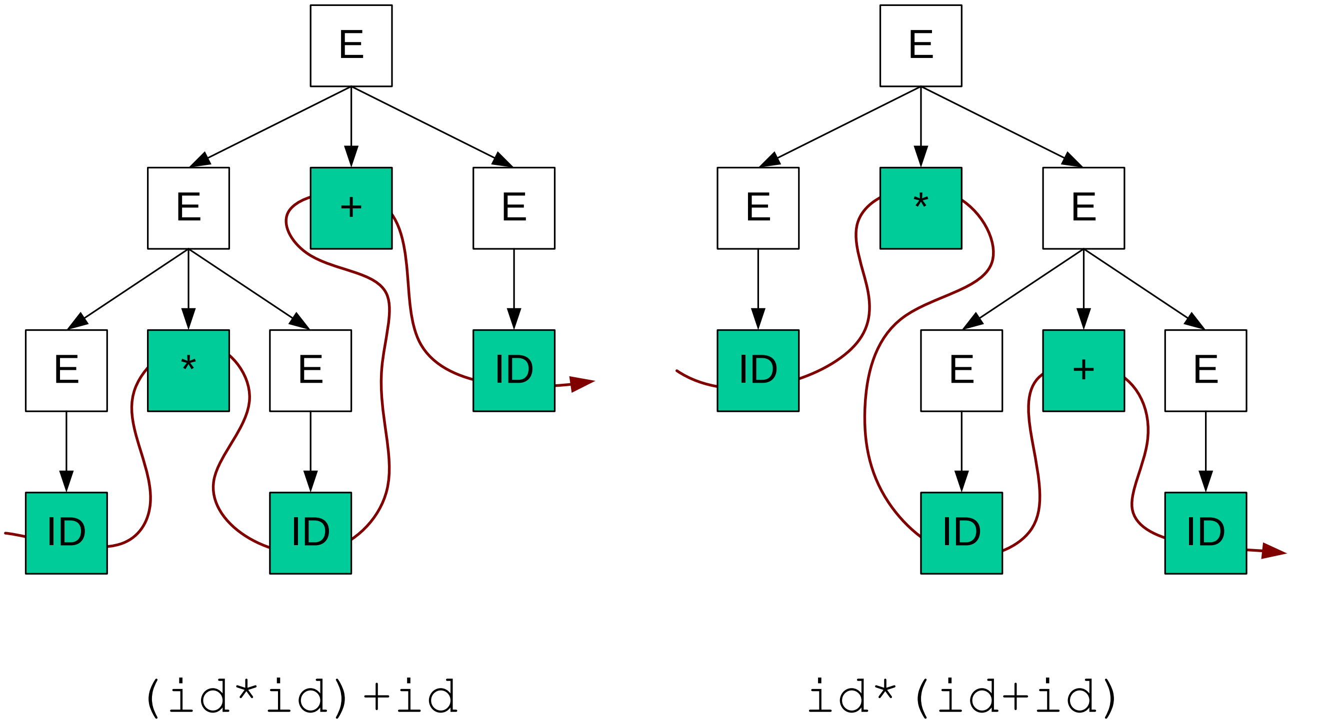
3.4. Ambiguity

- Grammar: E → E+E|E*E|(E)|ID
- String: ID*ID+ID



Ambiguity

- A grammar is **ambiguous** if it has more than one parse tree for some string.
- It can cause different interpretations of a program!



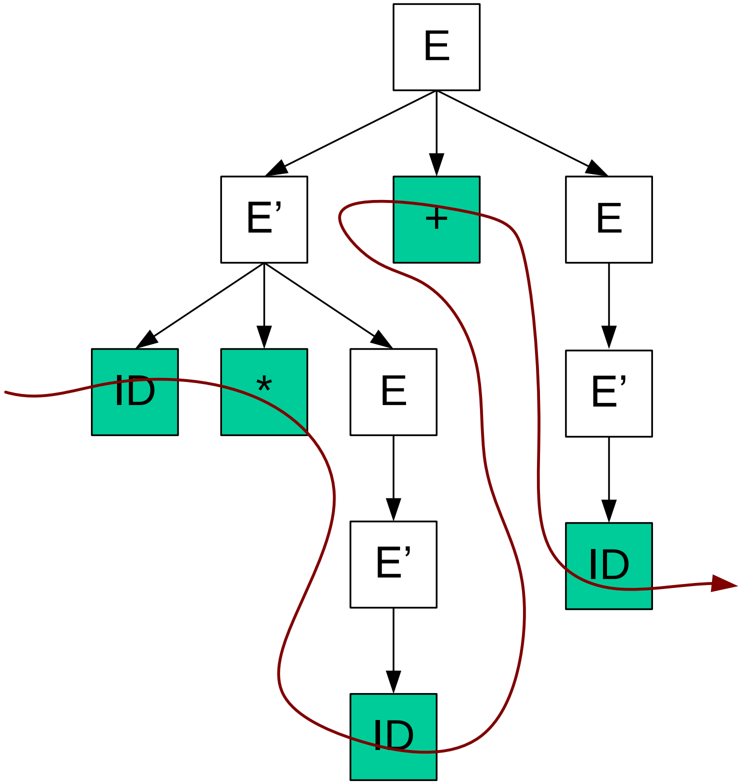
Ambiguity

To eliminate ambiguity:

- to rewrite a grammar;

E $\longrightarrow$ E'+E|E'

E' $\longrightarrow$ ID*E'|(E)*E'|(E)|D



(id*id)+id

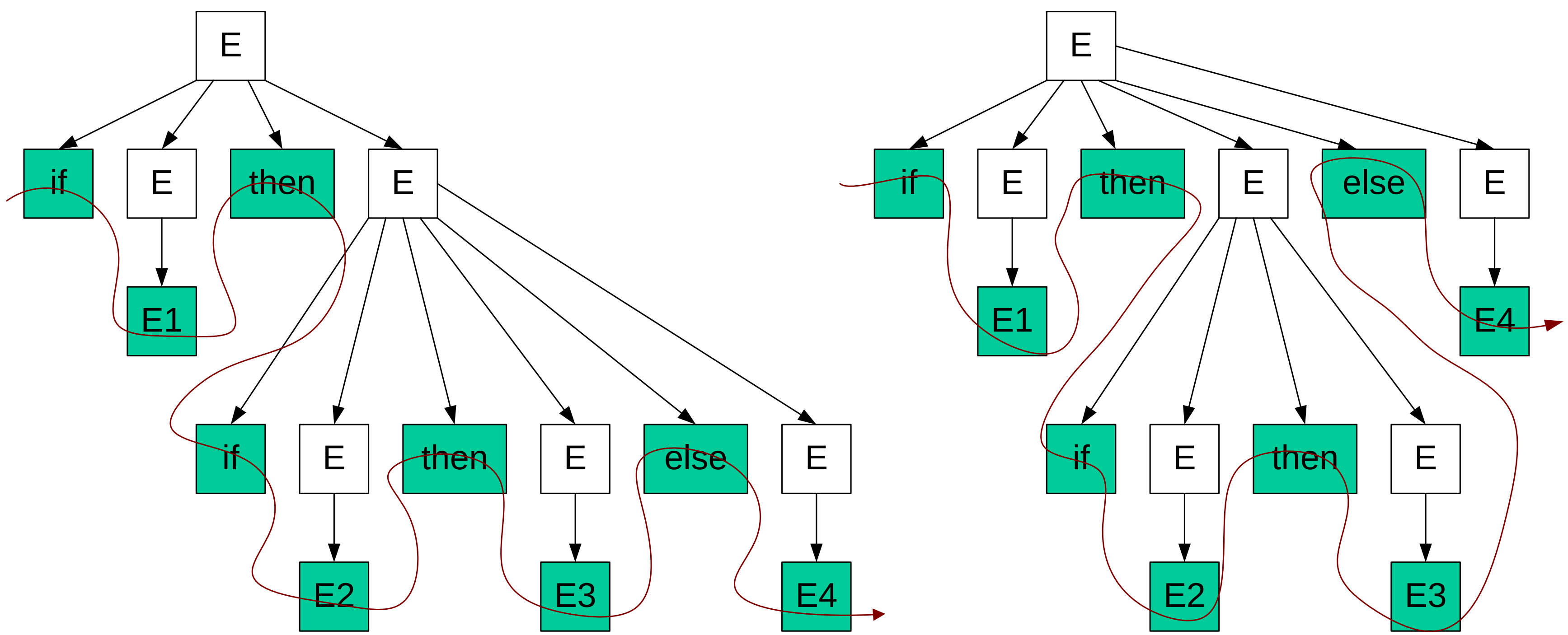
Ambiguity

- to use additional rules to resolve the ambiguities;

If-else grammar:

- E → if E then E | if E then E else E | ID

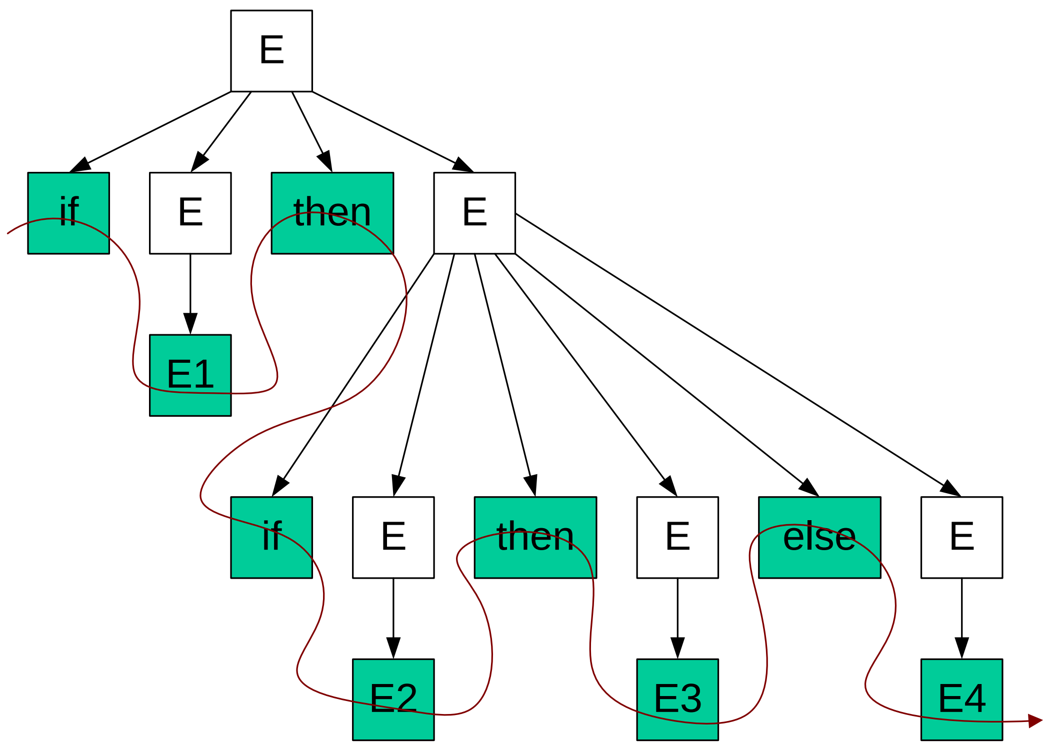
if E1 then if E2 then E3 else E4



Ambiguity

- **General rule:** *"Match each **else** with the closest unmatched **then**".*

if E1 then if E2 then E3 else E4



Ambiguity

Unambiguous if-else grammar:

- STMT → MATCHEDSTMT | OPENSTMT
- MATHEDSTMT → if EXPR then MATHEDSTMT else MATCHEDSTMT
- OPENSTMT → if EXPR then STMT | if EXPR then MATCHEDSTMT else OPENSTMT

3.5. Associativity and precedence

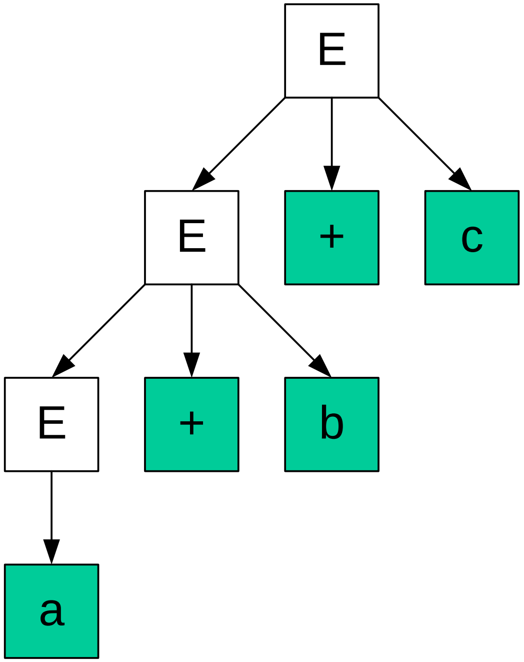
Associativity of operators:

- left-associated operators (-,+,*,/,…) – an operand with operator signs on both sides of it belongs to the operator to its left;
- right-associated operators (=,exponentiation,…) – an operand with operator signs on both sides of it belongs to the operator to its right.

.

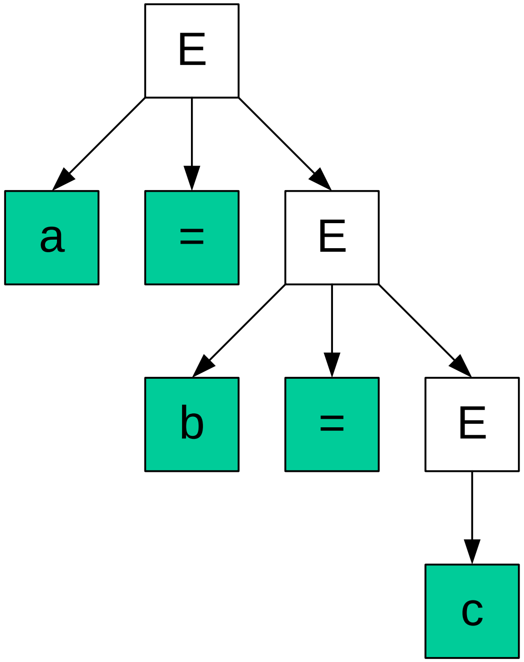
Associativity and precedence

$E \rightarrow E{+}a \mid \ldots \mid E{+}z \mid a \mid \ldots \mid z$



`(a+b)+c`

$E \rightarrow a{=}E \mid \ldots \mid z{=}E \mid a \mid \ldots \mid z$



`a=(b=c)`

Associativity and precedence

We say that * has **higher precedence** than + if * takes its operands before + does.

Operators *, / have higher precedence than +, -

$$\begin{tikzcd} & \texttt{a+b*c} \arrow[dl] \arrow[dr] & \\ \texttt{a+(b*c)} & & \cancel{\texttt{(a+b)*c}} \end{tikzcd}$$

Associativity and precedence

Arithmetic grammar

- E → LET = E | LET = EXPR | LET
- EXP → EXP+TERM | EXP-TERM | TERM
- TERM → TERM*FACTOR | TERM/FACTOR | FACTOR
- FACTOR → DIG | (EXP)
- DIG → 0 | … | 9
- FACTOR → LET | (EXP)
- LET → a | … | z

3.6. Left-recursion

- **Left-recursive grammar:** A → Aa

Left-recursive arithmetic grammar:

- E → E+T | T
- T → T*F | F
- F → (E) | ID

Non-left-recursive arithmetic grammar:

- E → TE’ | T
- E’ → +TE’ | +T
- T → FT’ | F
- T’ → *FT’ | *F
- F → (E) | ID

Left-recursion

Left-recursive grammar:

- $A \rightarrow Aw1 \mid \ldots \mid Awn$
- $A \rightarrow v1 \mid \ldots \mid vm$

Non-left-recursive grammar:

- $A \rightarrow v1A' \mid \ldots \mid vmA'$
- $A \rightarrow v1 \mid \ldots \mid vm$
- $A' \rightarrow w1A' \mid \ldots \mid wnA'$
- $A' \rightarrow w1 \mid \ldots \mid wn$

3.7. Left-factoring

- When the choice between two alternative productions for a non-terminal is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

If-else grammar:

- S → if E then S | if E then S else S | ID

If-else grammar after left-factoring:

- S → if E then S S’ | ID
- S’ → ε | else S

Left-factoring

Before:

- $A \rightarrow \sigma\alpha$
- $A \rightarrow \sigma\beta$

After:

- $A \rightarrow \sigma A'$
- $A' \rightarrow \alpha$
- $A' \rightarrow \beta$

σ is the longest common prefix

3.8. Error-recovery

Syntactic errors include

- misplaced semicolons or extra or missing braces; that is, "{" or " } . " ;
- in C or Java, the appearance of a case statement without an enclosing switch.

Error-recovery strategies:

- panic-mode;
- phrase-level;
- error-productions;
- global-correction.

Error-recovery

Panic-mode: the parser discards input symbols one at a time until one of a designated set of *synchronizing tokens* (semicolon ;, brace }, …) is found.

- Panic-mode correction often skips a considerable amount of input without checking it for additional errors (-);
- it has the advantage of simplicity (+);
- it is guaranteed not to go into an infinite loop (+).

`(1++`**`2`**`)*3`

Skip ahead to next integer (2) and then continue.

Error-recovery

Phrase-level: a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue (to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon).

- It can lead to infinite loops (for example, if we always inserted something on the input ahead of the current input symbol) (-).
- Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string (+).
- It has the difficulty in coping with situations in which the actual error has occurred before the point of detection (-).

Error-recovery

Error Productions: known common mistakes are specified in the grammar. A parser constructed from a grammar detects the anticipated errors when an error production is used during parsing.

- The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input (+).
- It complicates the grammar (-).

- 5x instead of 5*x: add the production $E \rightarrow EE$

Error-recovery

Global Correction: find a correct “nearby” program trying token insertions and deletions and other changes in tokens.

- Hard to implement (-).
- Slows down parsing of correct programs (-).
- “Nearby” is not necessarily “the intended” program (-).

Error-recovery

Past

- Slow recompilation cycle (even once a day)
- Find as many errors in one cycle as possible

Present

- Quick recompilation cycle
- Users tend to correct one error/cycle
- Complex error recovery is less compelling