ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 5a

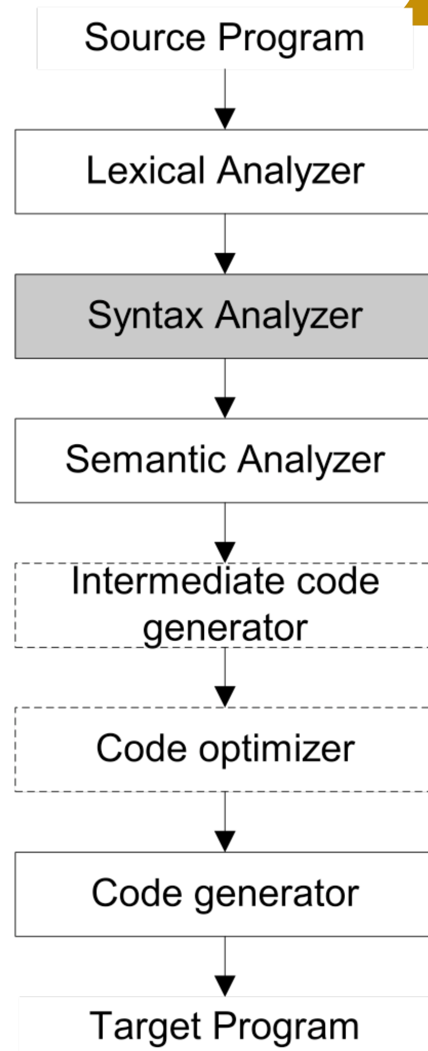
Syntax Analysis

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Syntax Analysis

Συντακτική Ανάλυση

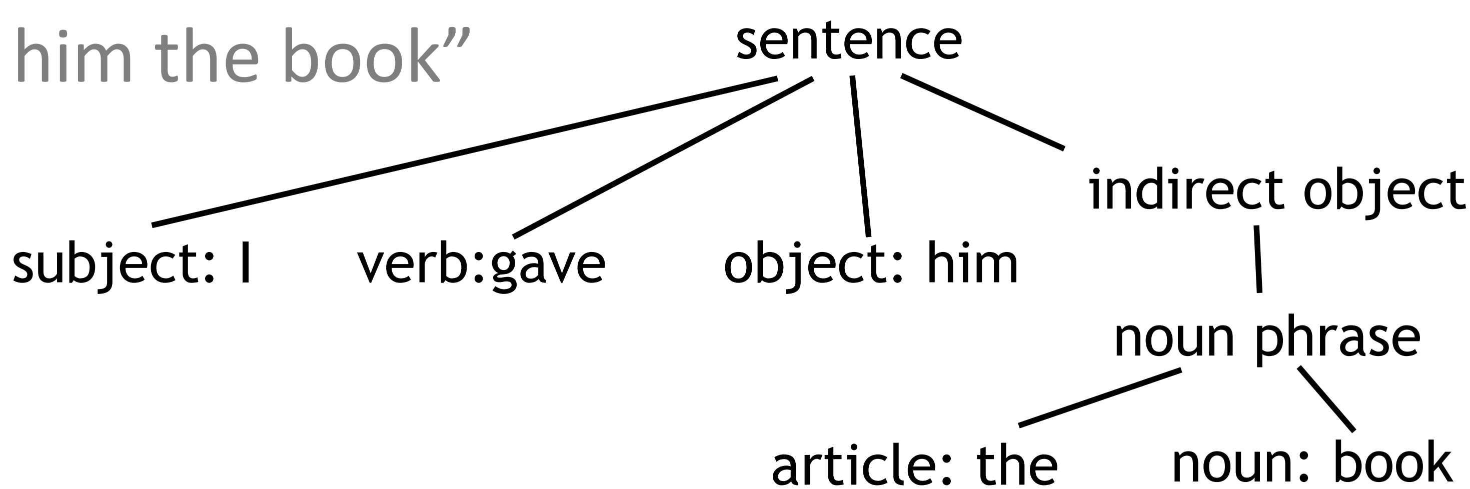
- Context-free Grammars (CFGs)
- Derivations
- Parse trees
- Top-down Parsing
- Ambiguities



Syntax Analysis

- Syntax analysis (parsing) is the process of determining if a string of tokens can be generated by a grammar

"I gave him the book"



Lexical-Syntax Analysis

Source code (character stream)

```
{
   if (b == 0) a = b;
   while (a != 1) { printf("%I ",I--); }
}
```

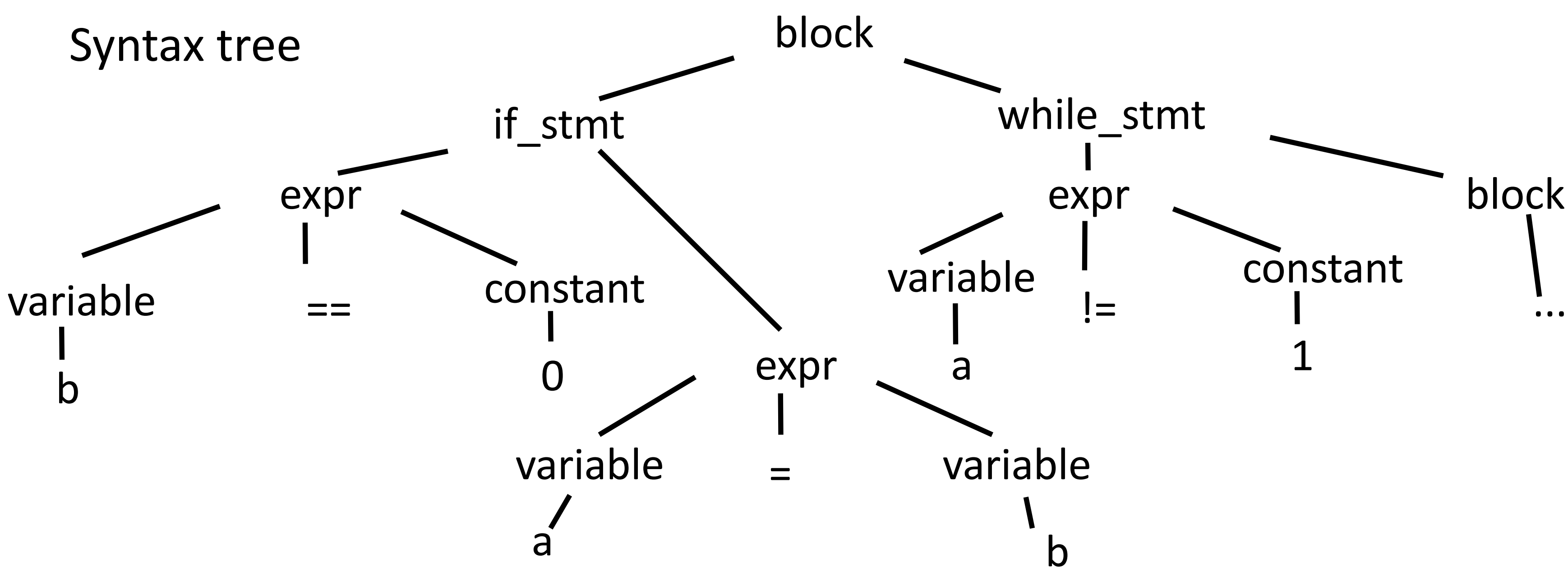
Lexical Analysis

Token stream

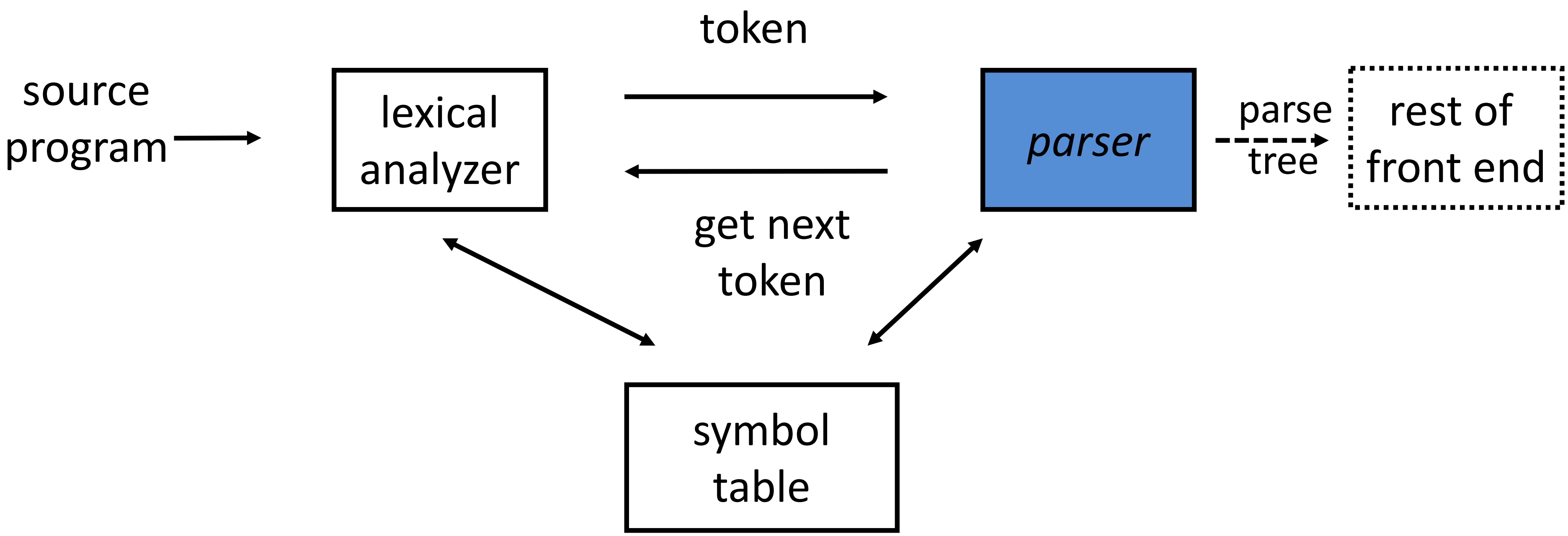
{	if	(	b	==	0	)	a	=	b	;	

Syntax Analysis

Syntax tree



The Role of the Parser



Syntax Analysis Operation

- Input
  - A stream of tokens taken from lexical analysis
- Output
  - Syntax tree which determines the token relations and the syntax correctness (are all parentheses balanced?)
- Semantic analysis takes care of types
  - `int x = true;`
  - `int y; z = f(y);`

Syntax Error Handling

- Lexical
  - Misspelling an identifier, keyword, or operator
- Syntactic
  - Arithmetic expression with unbalanced parenthesis
- Semantic
  - Operator applied to an incompatible operand
- Logical
  - Infinitely recursive call

Error Handler Requirements

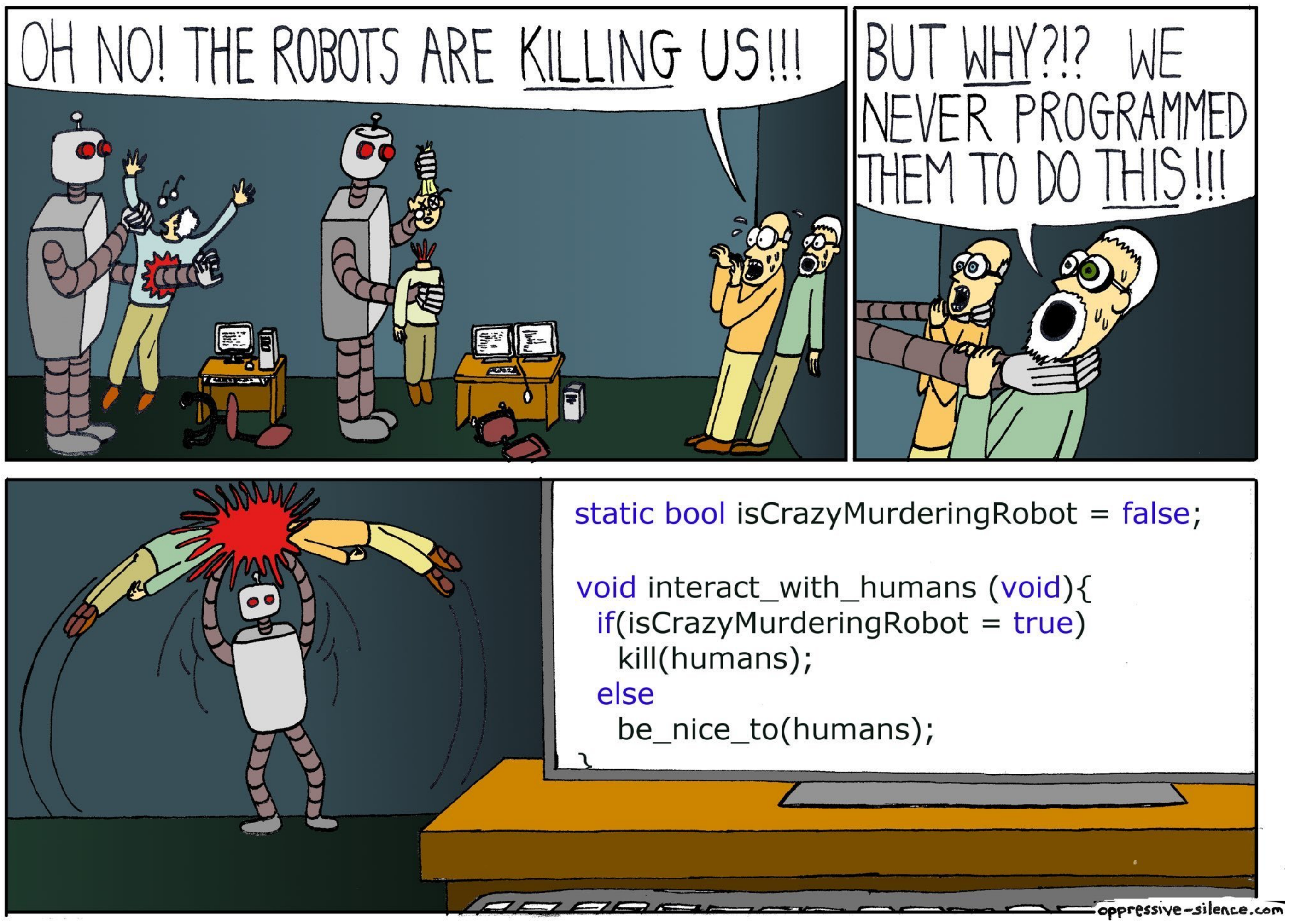
- It should report the presence of errors clearly and accurately
- It should recover from each error quickly enough to be able to detect subsequent errors
- It should not significantly slow down the processing of correct programs

What happens when an error is detected?

- Many strategies, none clearly dominates
- Not adequate for the parser to quit upon detecting the **first** error
  - Subsequent parsing may reveal additional errors
- Usually, the compiler attempts error recovery
  - Reasonable hope that the rest of the program can be parsed
- Error recovery should be realized correctly
  - Otherwise **many** errors can be generated

Example

- While recovering from an error a compiler may skip the declaration of a variable **`zap`**
- At a later point when **`zap`** is used the compiler should not generate a syntactic error, but just the missing declaration
  - Since, there should be no entry at the symbol table
- Conservative strategy
  - Once an error is detected, filter out close errors (consume enough tokens to exit the error area)


OH NO! THE ROBOTS ARE KILLING US!!!
BUT WHY?!? WE NEVER PROGRAMMED THEM TO DO THIS!!!
static bool isCrazyMurderingRobot = false;
void interact_with_humans (void){
if(isCrazyMurderingRobot = true)
kill(humans);
else
be_nice_to(humans);
}
oppressive-silence.com

Error-recovery Strategies

- Panic mode
  - Once an error is detected, consume tokens until a *synchronizing token* is detected
  - Synchronizing tokens are usually delimiters (**`end, ;`**), which have a clear meaning
  - Simple and cannot enter an infinite loop
- Phrase level
  - Attempt to correct the error by taking action
  - Insert a missing semicolon, replace a comma with a semicolon, etc.
  - Can create infinite loops if actions are not applied correctly
  - Hard to cope with cases where the error has occurred before the point of detection

Error-recovery Strategies

- Error productions
  - Common errors can be augmented to the grammar of the language
  - The parser can then detect errors, since these errors are part of the language
- Global correction
  - Attempt to correct an error with the least possible actions
  - Given an incorrect input string *x* and grammar *G*, find a valid *y*, which can be derived from *x* with the least amount of changes
  - The closest correct program may not be the one the programmer had in mind

Γραμματικές Χωρίς Συμφραζόμενα

CONTEXT-FREE GRAMMARS

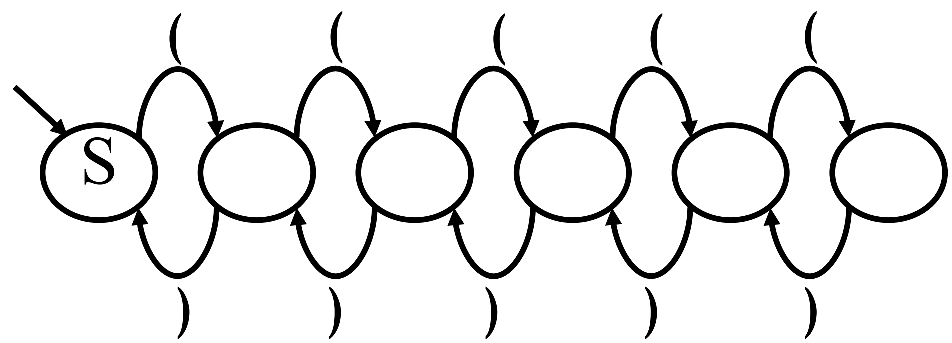
Regular Expressions Limitations

- Regular expressions can be transformed **easily** to NFA (and then to DFA)
- Discovering and classifying tokens using regular expressions is easy and efficient
- Regular expressions cannot be used for syntax analysis

Regular Expressions Limitations

- Match all balanced parentheses:
  - ( )   (( ))   ( ) ( ) ( )   (( )) ( ) ((( ) ( )))
- You need an NFA with an infinite number of states

For 5 nested parentheses you need the following NFA



Context-free Grammar (CFG)

Γραμματική Χωρίς Συμφραζόμενα

1. A set of tokens, known as *terminal* symbols.
   – Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about programming languages (e.g., tokens like **`if`**, **`then`**, and **`else`** are all terminals)
2. A set of nonterminals.
   – Nonterminals are syntactic variables that denote sets of strings. The nonterminals define sets of strings that help define the language generated by the grammar. They also impose a hierarchical structure on the language defined by the grammar.

Context-free Grammar (CFG)

Γραμματική Χωρίς Συμφραζόμενα

3. A set of *productions (κανόνες παραγωγής)* where each production consists of a nonterminal, called the *left side* of the production, an arrow, and a sequence of tokens and/or nonterminals, called the *right side* of the production.
   - The productions of the grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of a nonterminal, followed by an arrow (sometimes the symbol `::==` is used in place of the arrow), followed by a string of nonterminals and terminals.
4. A designation of one of the nonterminals as the *start* symbol
   - In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.

Example 1

- Expressions of digits separated by plus and minus signs
  - `9-5+2, 3-1, 7`

```
list  → list + digit              (2.2)
list  → list – digit              (2.3)
list  → digit                     (2.4)
digit → 0|1|2|3|4|5|6|7|8|9       (2.5)
```

The three first productions can be grouped:

```
list  → list + digit | list – digit | digit
```

Terminals/Tokens: + – 0 1 2 3 4 5 6 7 8 9
Nonterminals: *list*, *digit*
Sart symbol: *list*

Example 1

- The ten productions for the nonterminal *digit* allow it to stand for any of the tokens `0, 1, ..., 9`
- From 2.4 a single *digit* by itself is a *list*
- 2.2 and 2.3 express the fact that if we take any list and follow it by a plus or minus sign and then another *digit* we have a new *list*

9−5+2

- 9 is a list by production 2.4, since 9 is a digit
- 9−5 is a list by production 2.3, since 9 is a list and 5 is a digit
- 9−5+2 is a list by production 2.2, since 9−5 is a list and 2 is a digit

Example 2

- "**Begin End**" block in Pascal

```
begin
   ... (* Pascal code *)
end

block        → begin opt_stmts end
opt_stmts    → stmt_list | ε
stmt_list    → stmt_list ; stmt | stmt

( stmt  is not expanded at this point )
```

Example 3

- Simple arithmetic expressions

expr	→	expr op expr
expr	→	*(expr)*
expr	→	*–expr*
expr	→	**id**
op	→	+
op	→	–
op	→	*
op	→	/
op	→	^

Equal with:

E → *E A E* | (*E*) | –*E* | **id**

A → + | – | * | / | ^

Derivation

Παραγωγή

$$E \rightarrow E\ A\ E \mid (E) \mid -E \mid \mathbf{id}$$

- The production $E \rightarrow -E$ signifies that an expression preceded by a minus sign is also an expression
- We can thus generate more complex expressions from simpler expressions by just replacing E with $-E$

Derivation

Παραγωγή

$$E \Rightarrow -E$$

(E *derives* $-E$)

Examples

$E \rightarrow (E)$

$E*E \Rightarrow (E)*E$ or $E*(E)$

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$

$\Rightarrow$ Derives in one step
$\overset{*}{\Rightarrow}$ Derives in zero ore more steps
$\overset{+}{\Rightarrow}$ Derives in one or more steps

Leftmost - Rightmost

$$E \rightarrow E\ A\ E \mid (E) \mid -E \mid \textbf{id} \qquad \textbf{(G1)}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \hat{}$$

The string $-(\textbf{id} + \textbf{id})$ is a sentence of grammar G1

Leftmost derivation

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$

Rightmost derivation

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\textbf{id}) \underset{rm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$

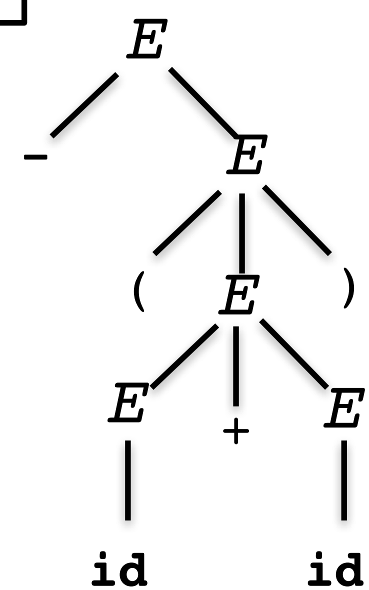
Grammars and Languages

- Given a grammar *G* with a start symbol *S*,
  - A string of **only terminals,** *w,* is in *L(G)* iff $S \Rightarrow^{+} w$
  - The string *w* is called a *sentence of G*
  - *L(G)* is the language generated by *G* and includes all *w* (strings composed by terminals of *G*)
- A language that can be generated by a grammar is a *context-free grammar*
- If two grammars generate the same language, then they are *equivalent*

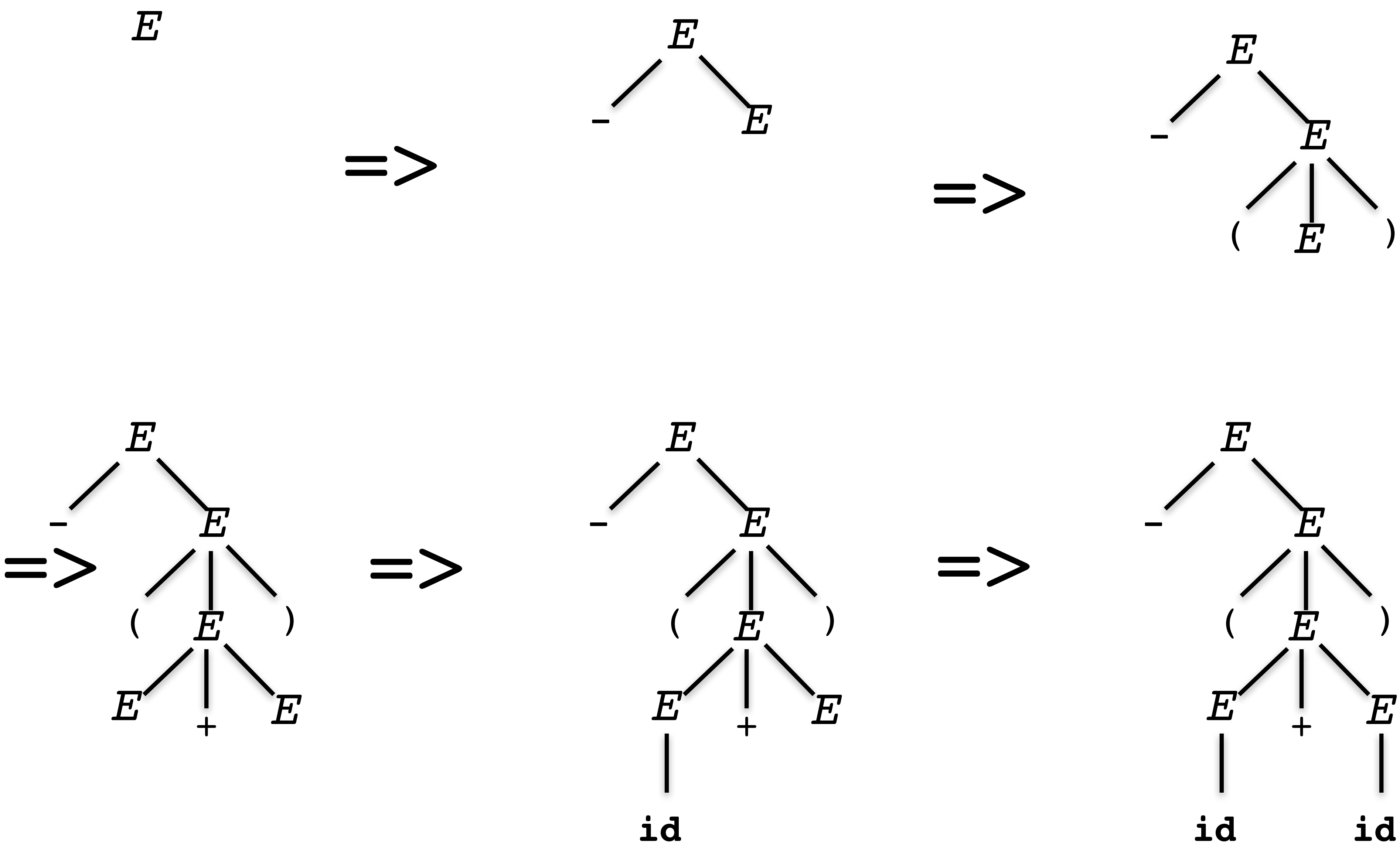
Parse Trees

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order.

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$



Constructing the Parse Tree



Ambiguity

Αμφισημία

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*
- For certain types of parsers, it is desirable that the grammar be made unambiguous
- For some applications we shall also consider methods whereby we can use certain ambiguous grammars, together with *disambiguating rules* that “throw away” undesirable parse trees