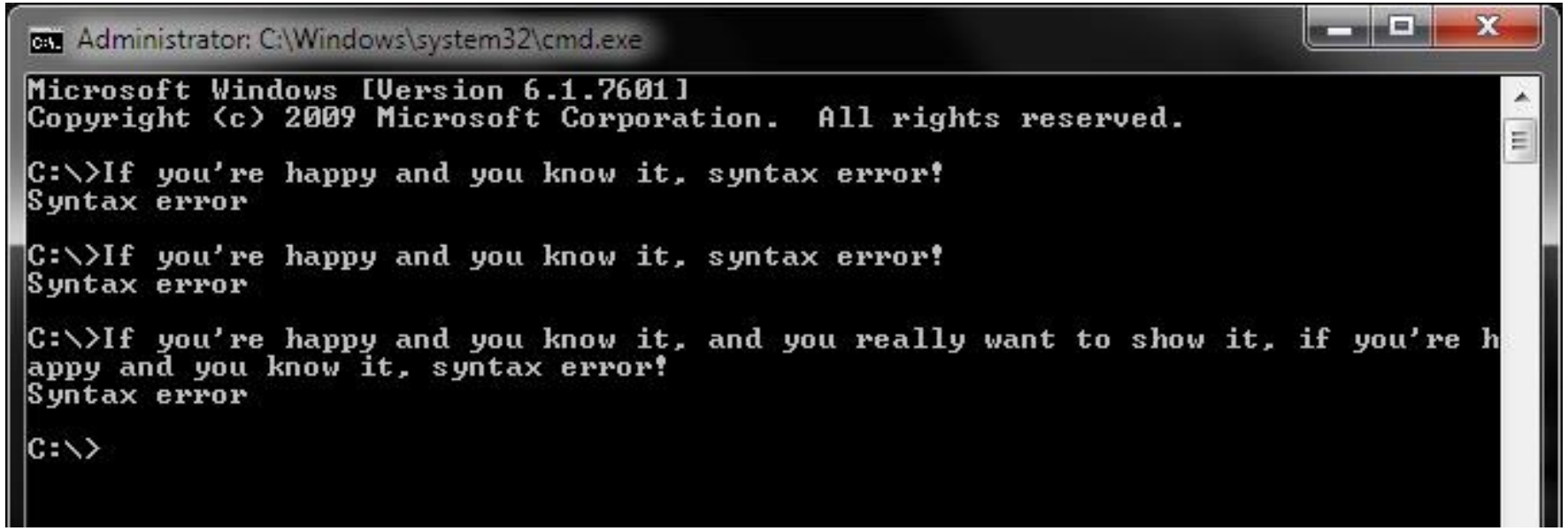
Syntax and Semantics



IN3040 – Programming Languages

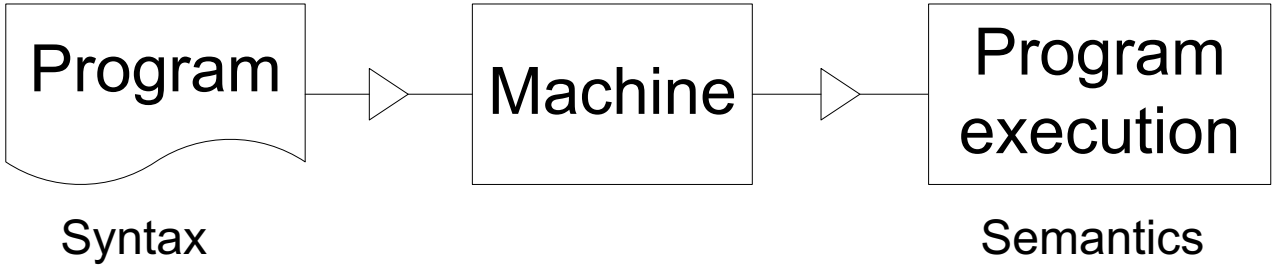
Eyvind W. Axelsen - eyvinda@ifi.uio.no

Slides adapted from previous years' slides made
by Birger Møller-Pedersen - birger@ifi.uio.no

Outline

- Program != program execution
- Compiler/interpreter
  - This is not a compiler course
  - …but some basic knowledge of language constructs is needed
- Syntax
  - Grammars
  - BNF
  - Scanning/Parsing

Program != program execution



- Important topics for this course:
  - Understand design tradeoffs in PL design
  - Understand how programs are executed
  - Understand how languages are implemented (though this is not a compiler course)

```
int x = 1;

procedure a() {
  int x;
  b();
}

procedure b() {
  x = 2;
}

a();
print x;
```

Which x are we writing to?

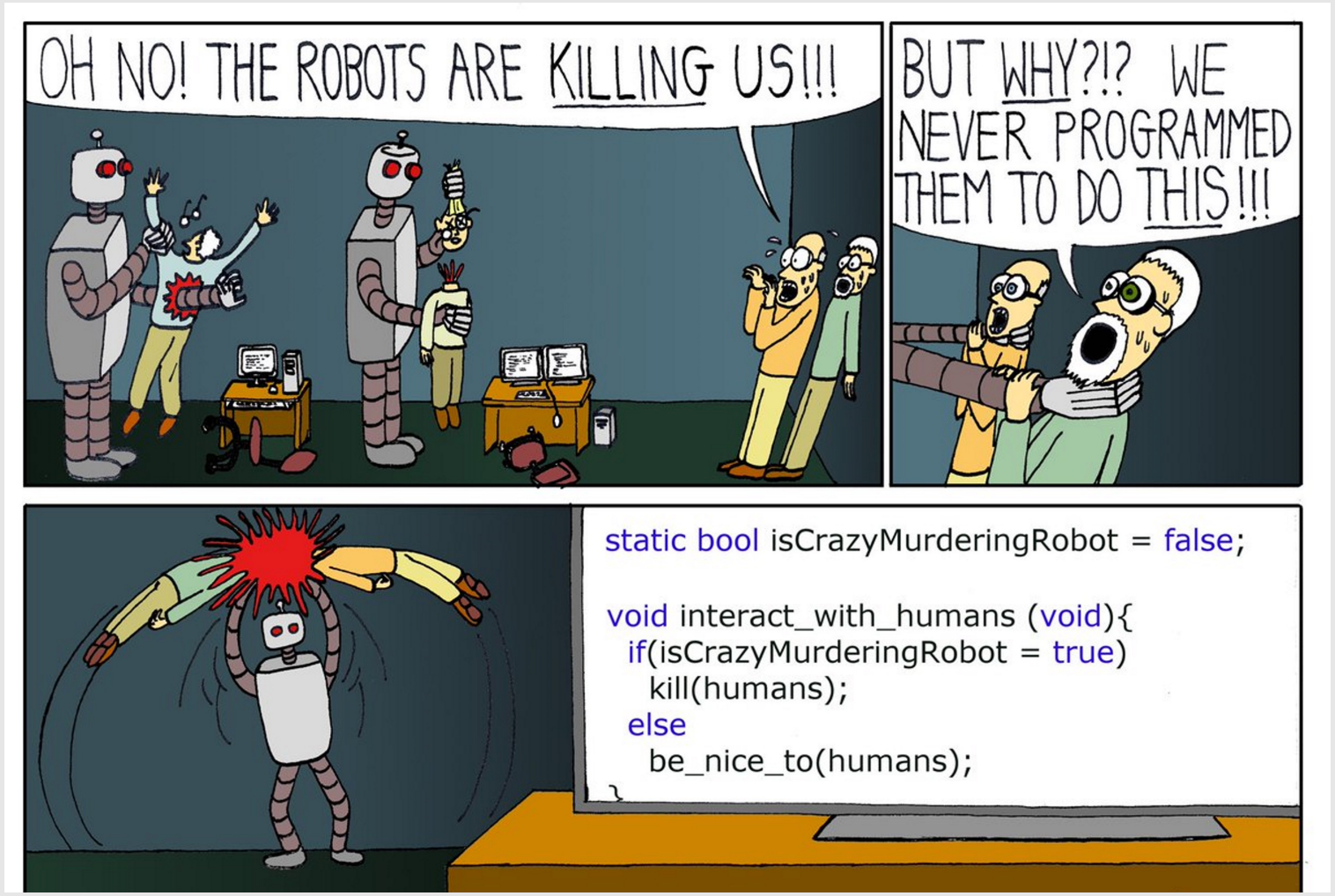
What will be printed?

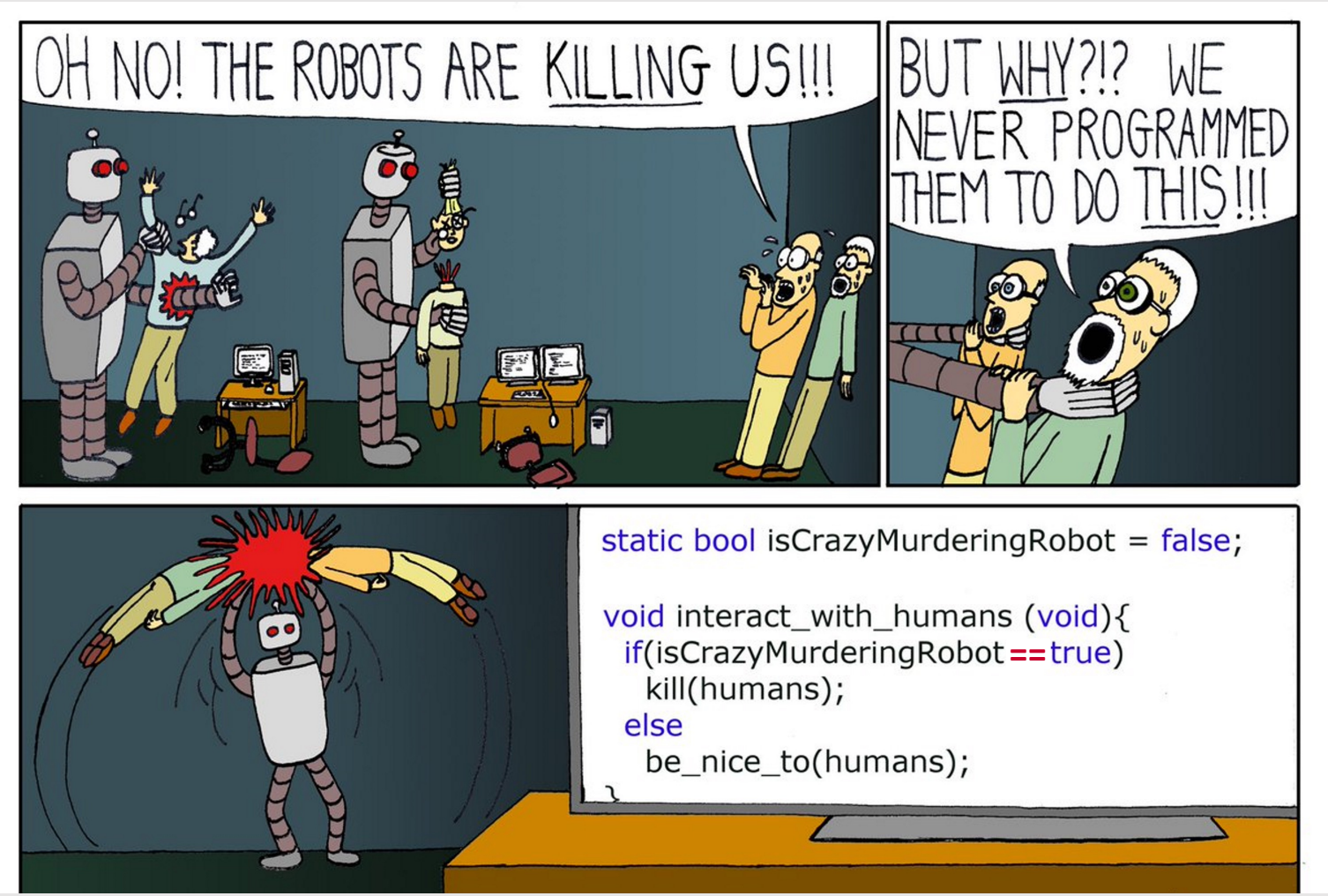
Syntax != Semantics

A description of a programming language consists of two main components:

- *Syntactic* rules
  - What *form* does a legal program have.
- *Semantic* rules:
  - Which programs are meaningful?
  - What do the sentences (of meaningful programs) in the language mean?

- *Static* semantics: rules that may be checked *before* the execution of the program, e.g.:
  - All variables must be declared.
  - Declaration and use of variables coincide (type check).
  - Different languages have different rules!
- *Dynamic* semantics: What shall happen during the *execution* of the program?
  - *Operational* semantics, that is a semantics that describes the behavior of an (idealized) abstract machine performing a program,
  - or, mapping to something else (but well-known and well-defined) - *denotational* semantics.

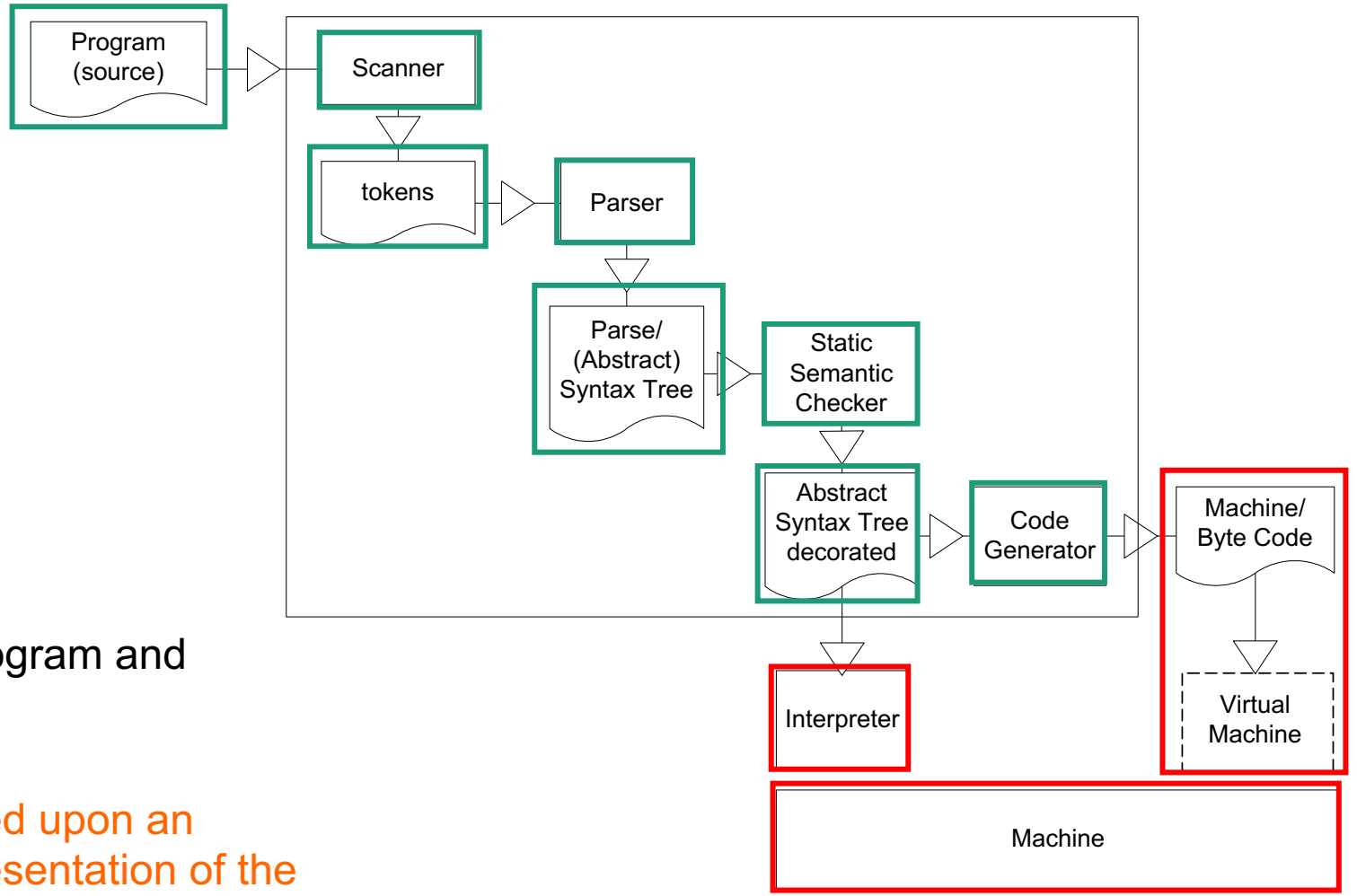

OH NO! THE ROBOTS ARE KILLING US!!!
BUT WHY?!? WE NEVER PROGRAMMED THEM TO DO THIS!!!
static bool isCrazyMurderingRobot = false;
void interact_with_humans (void){
if(isCrazyMurderingRobot = true)
kill(humans);
else
be_nice_to(humans);
}



Syntax matters!

Compiler/interpreter

- A *compiler* translates a program to another language
  - Typically a machine language
  - Or a language for a virtual machine
  - Do you know any such languges?



- An *interpreter* reads a program and simulates its operations.

- Both are (or can be) based upon an *abstract syntax tree* representation of the program

Interpreters

- You will make one in the obligs!

- An interpreter is a program that will
  - Parse the source code of a given language (the *parser*)
    - We skip this step in the oblig
  - Execute the instructions/sentences one by one, at runtime
    - (without first compiling to machine code)

- Examples of (typically) interpreted languages:
  - Python, Ruby, Perl, Basic, JavaScript, ...
  - The latter currently ”rules the web” – interesting development with transpilers

- Why interpreters?
  - Allows very dynamic languages that can do things a compiler cannot check
  - Very quick write → run cycle (no compiling, just run the thing)
  - Relatively easy to implement (as you will see for yourselves!)

- Why not?
  - Execution speed
  - Compiler feedback

Syntax

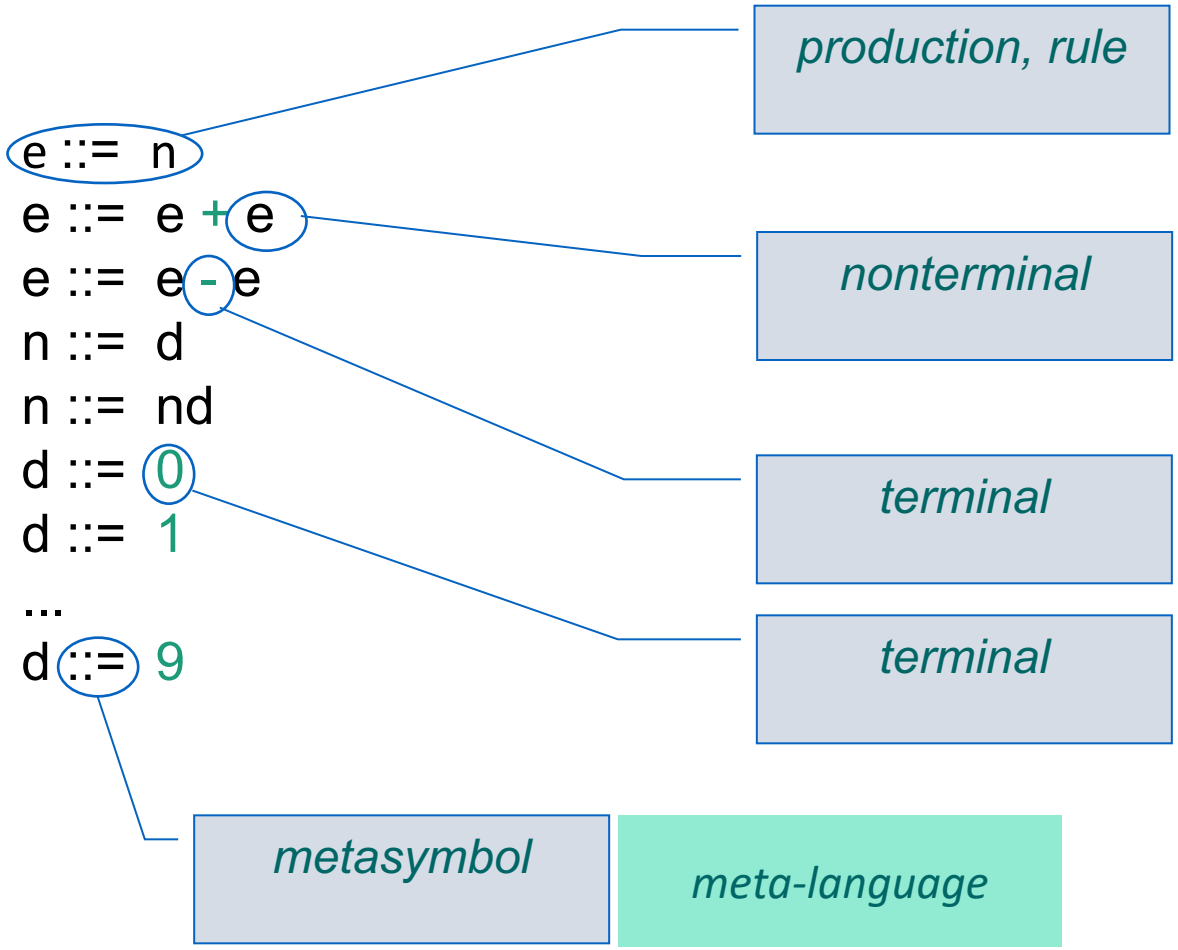
Imagine a very simple language, that allows us to write simple numeric expressions, e.g.

1 + 2
42 + 42
11 – 9 + 2

etc.

How do we express the grammar (the syntax) for this language?

Syntax: described by BNF grammars



Terminals are found in the program text

Non-terminals are not

Extended BNF

- In Extended BNF (eBNF) we can use the following *metasymbols* on the righthand side:

|	alternatives
[…]	optionality (0 or 1 time)
*	zero or more times (from regular expressions – alternatively {...})
+	one or more times (from regular expressions)
(…)	grouping symbols (sometimes {...} is used)

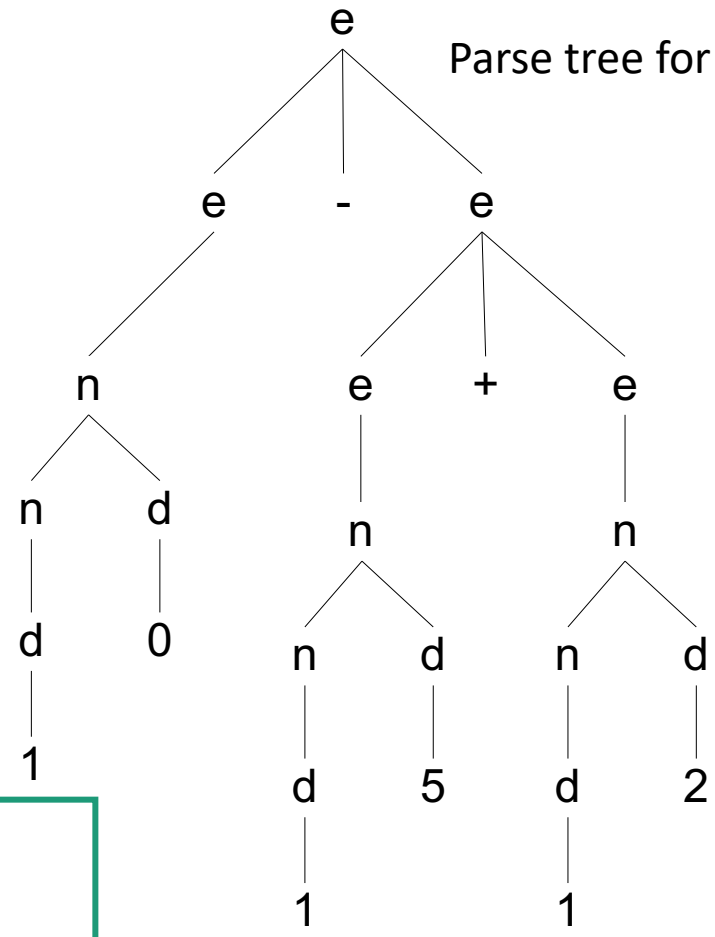
Grammar from previous slide expressed more concisely with eBNF

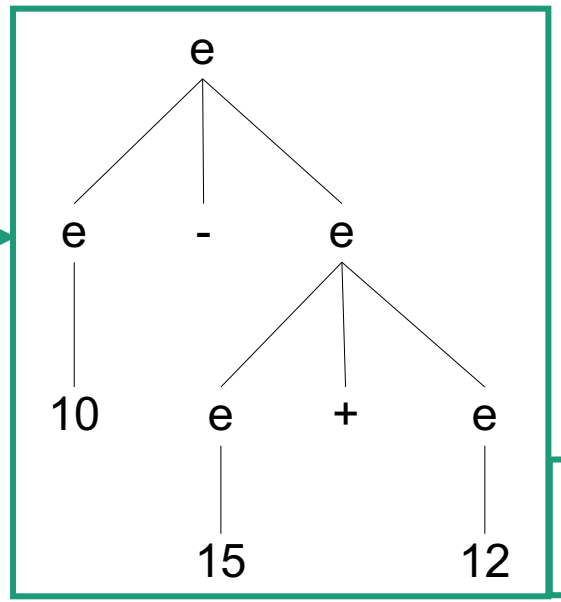
```
e ::=  n | e + e | e - e
n ::=  d | nd
d ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Derivation of sentences

- The possible sentences in a language defined by a BNF-grammar are those that emerge by following this procedure:
  1. Start with the start symbol (e).
  2. For each nonterminal symbol (e, n, d) exchange this with one of the alternatives on the right hand side of the production defining this nonterminal.
  3. Repeat §2 until only terminal symbols remain.
- This is called a *derivation* from the start symbol to a sentence, represented by a **parse tree / (concrete) syntax tree**
- Removing unnecessary derivations and nodes gives an **abstract syntax tree**

```
e ::= n | e + e | e - e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



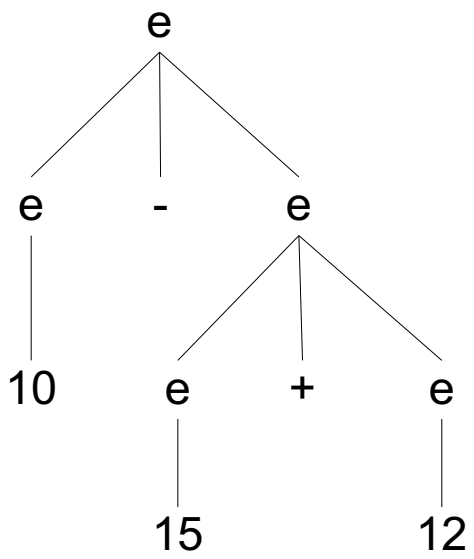


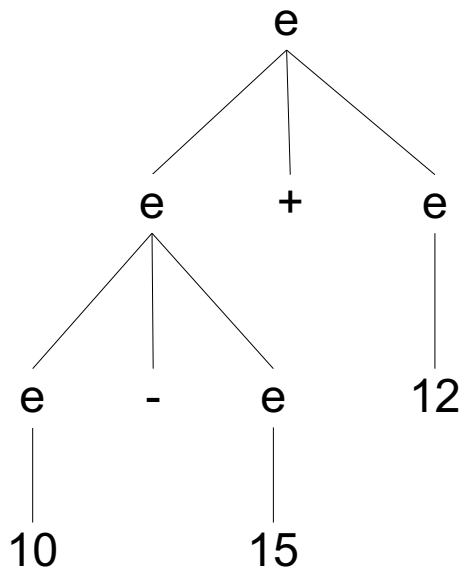
Abstract syntax tree
Relevant for oblig!

Only one possible production?

10 - 15 + 12 = ?

```
e ::= n | e + e | e - e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```





10 – (15 + 12) =
10 – 27 =
-17

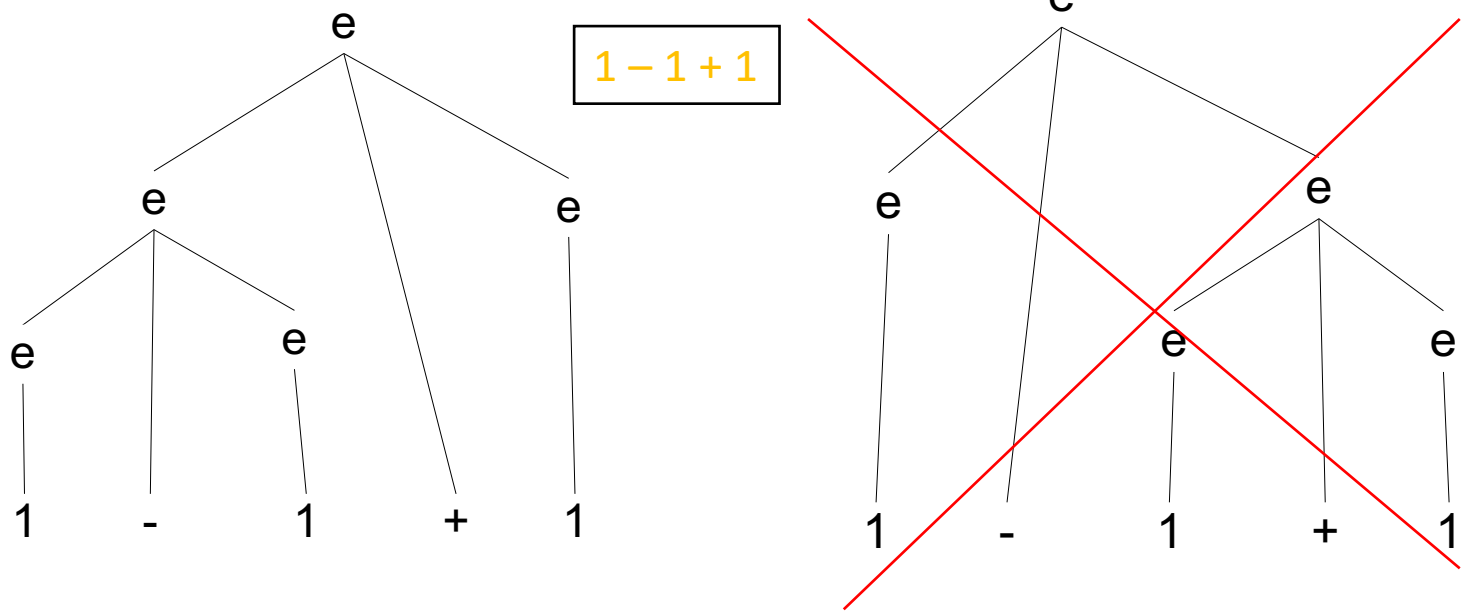
(10 – 15) + 12 =
-5 + 12 =
7

Unambiguous/ Ambiguous Grammars

- If every sentence in the language can be derived by one and only one parse tree, then the grammar is **unambiguous**, otherwise it is **ambiguous**.

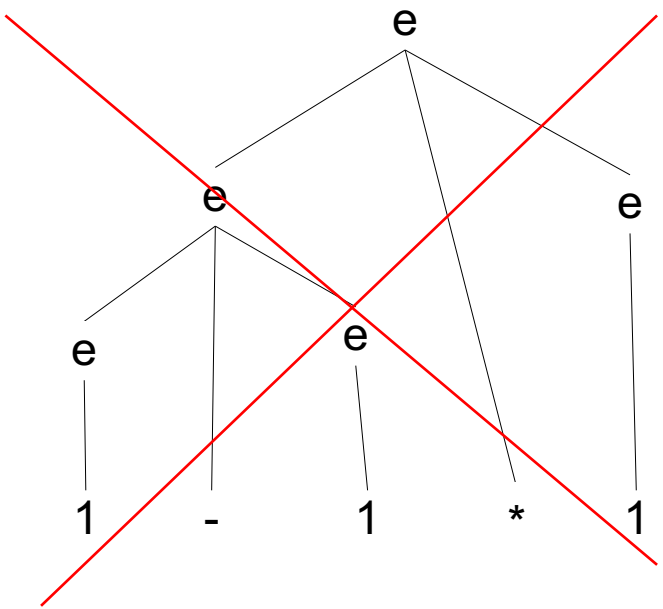
```
e ::= 0 | 1 | e + e | e - e | e * e
```

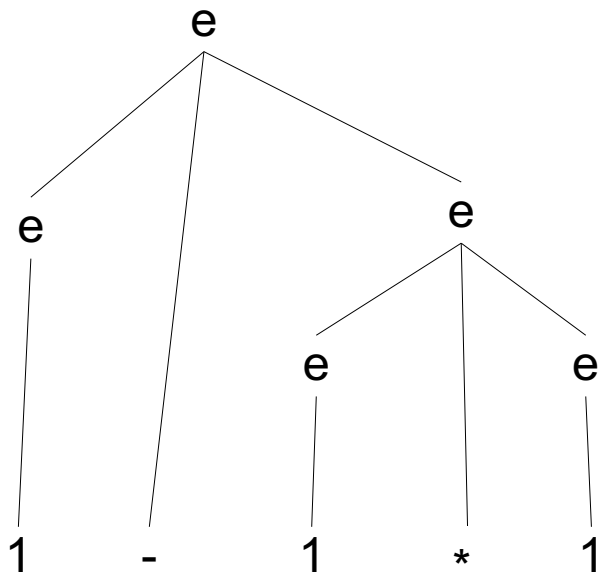
- Ambiguity handled by **associativity** and **precedence** rules



Which is «correct»?

1 – 1 * 1 Which is «correct»?

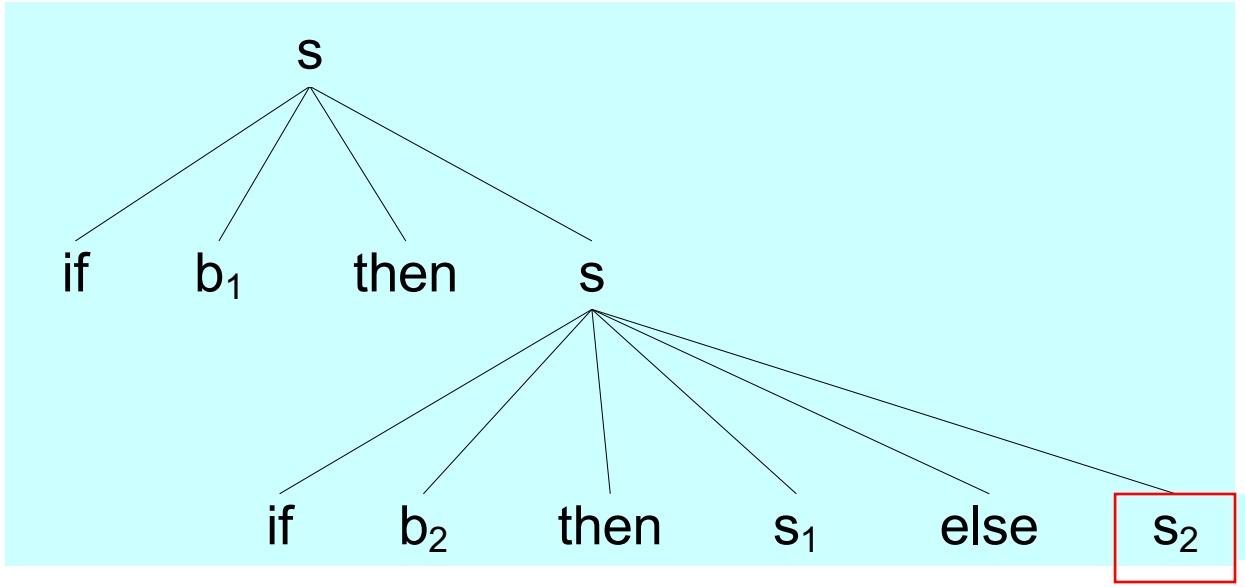


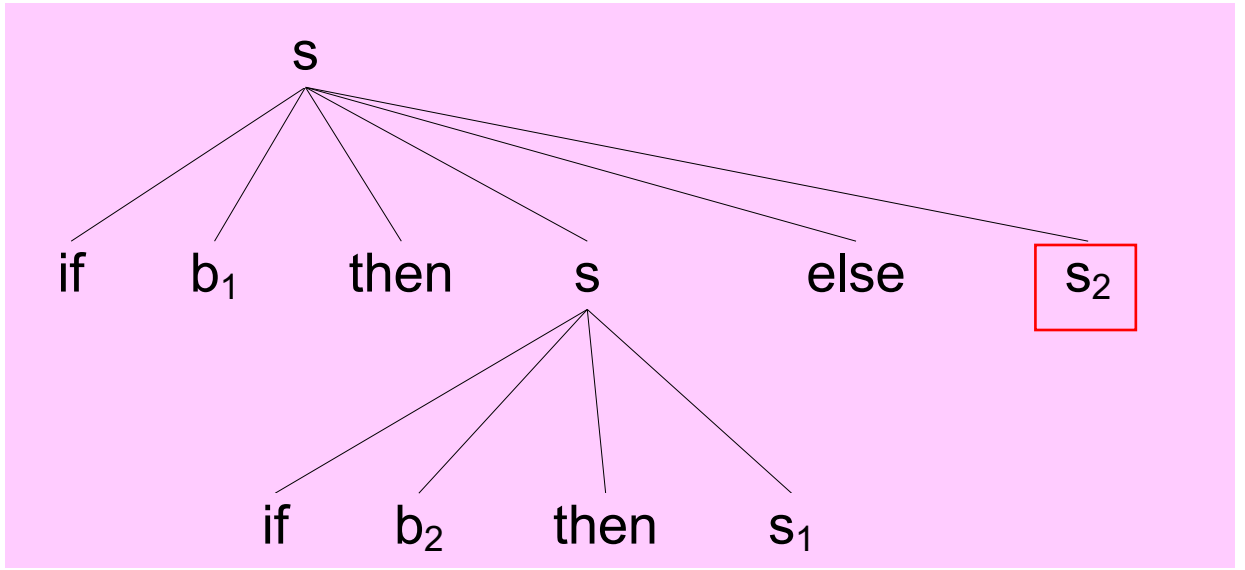


A somewhat more interesting language

```
s ::=   v := e  | s ; s  | if b then s | if b then s else s
v ::=   x  | y  | z
e ::=   v  | 0  | 1 | 2 | 3 | 4
b ::=   e = e
```

if b_1 then if b_2 then s_1 else s_2





is s_2 executed?

b_1 = true
b_2 = false

yes

no

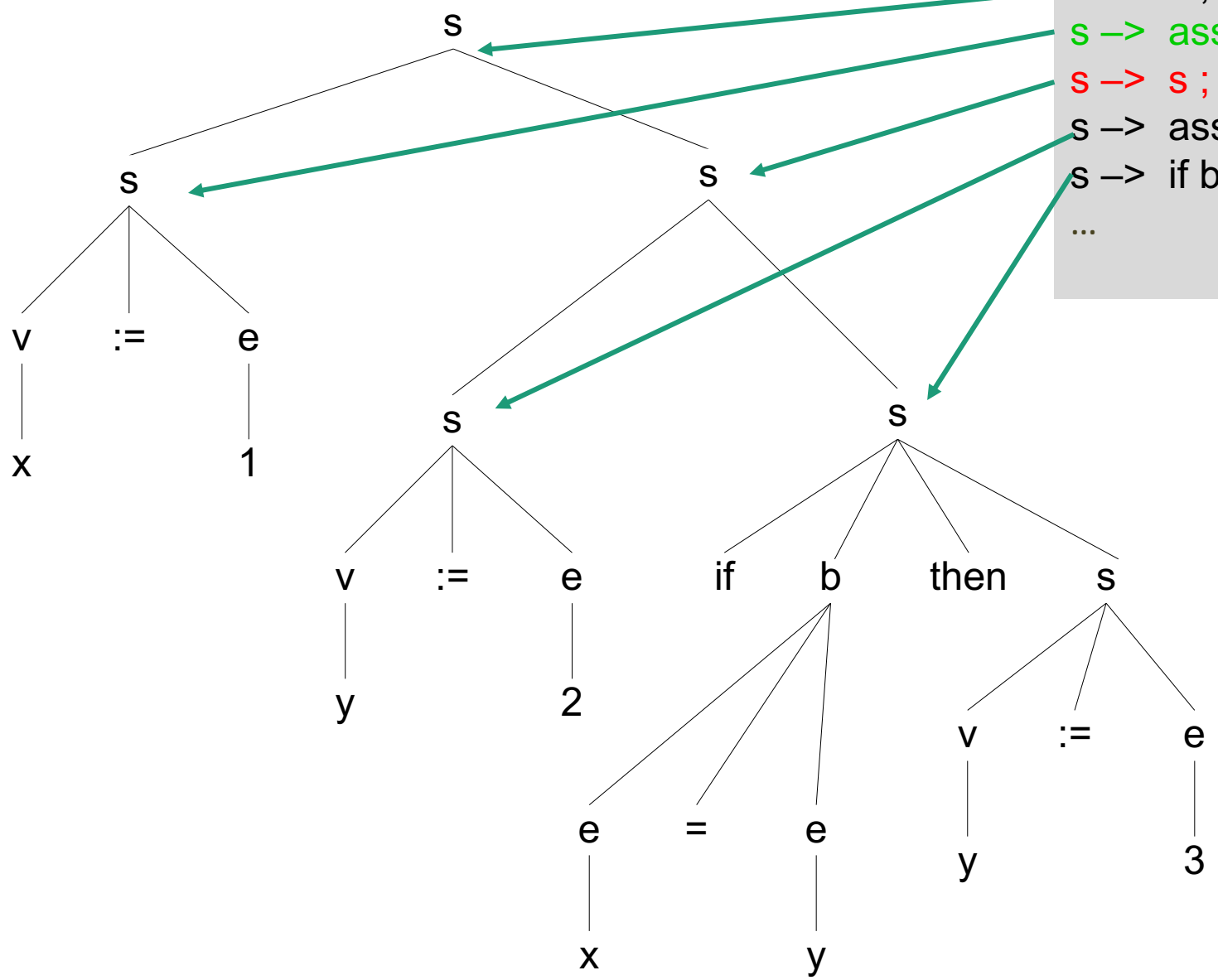
b_1 = false
b_2 = true

no

yes

Parsing matters!

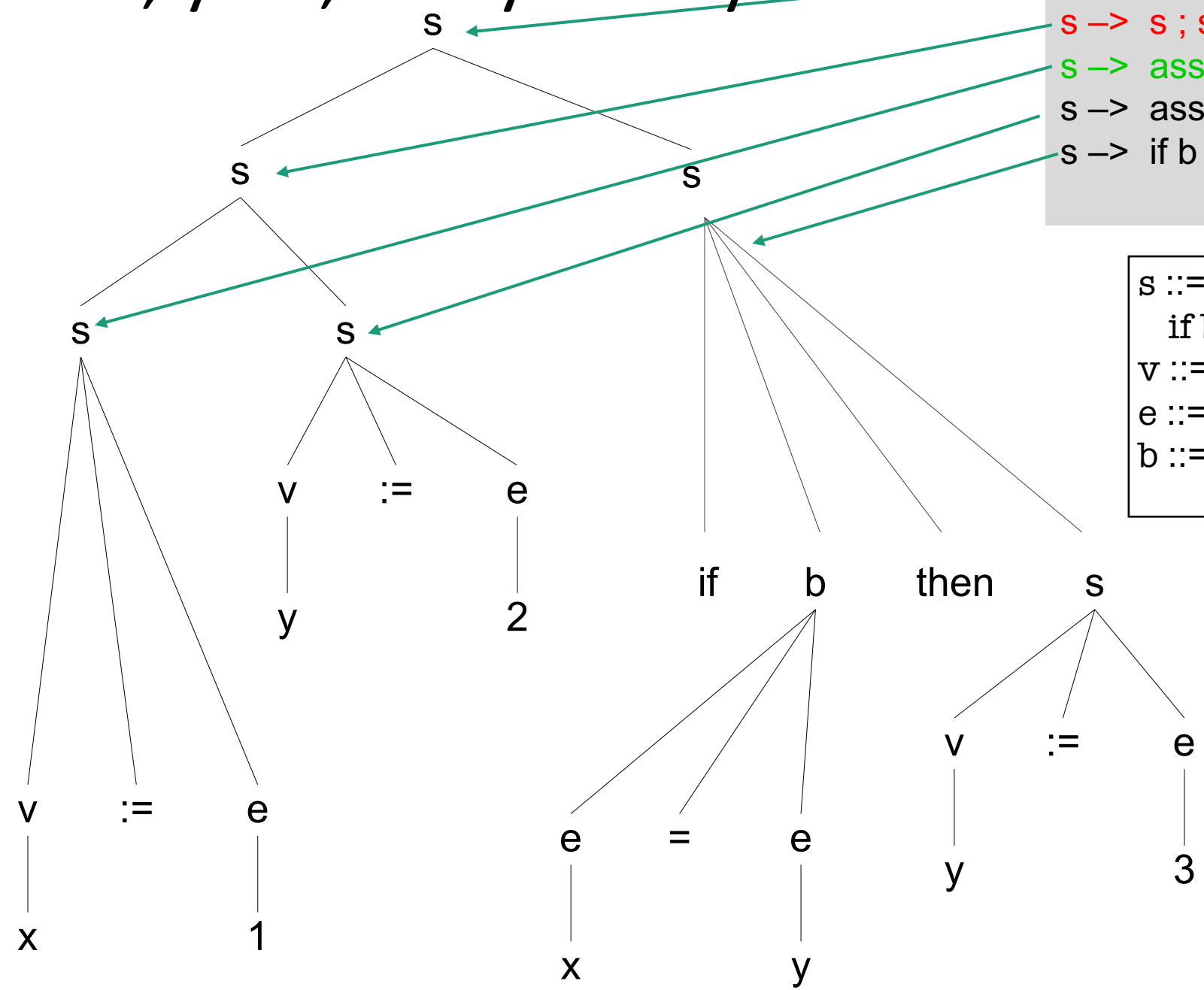
x:=1; y:=2; if x=y then y:=3



s –> s ; s
s –> assign (x := 1)
s –> s ; s
s –> assign (y := 2)
s –> if b then s
...

```
s ::=      v := e  | s ; s  |
  if b then s | if b then s else s
v ::= x  | y  | z
e ::= v  | 0  | 1 | 2 | 3 | 4
b ::= e = e
```

x:=1; y:=2; if x=y then y:=3



s –> s ; s
s –> s ; s
s –> assign (x := 1)
s –> assign (y := 2)
s –> if b then s

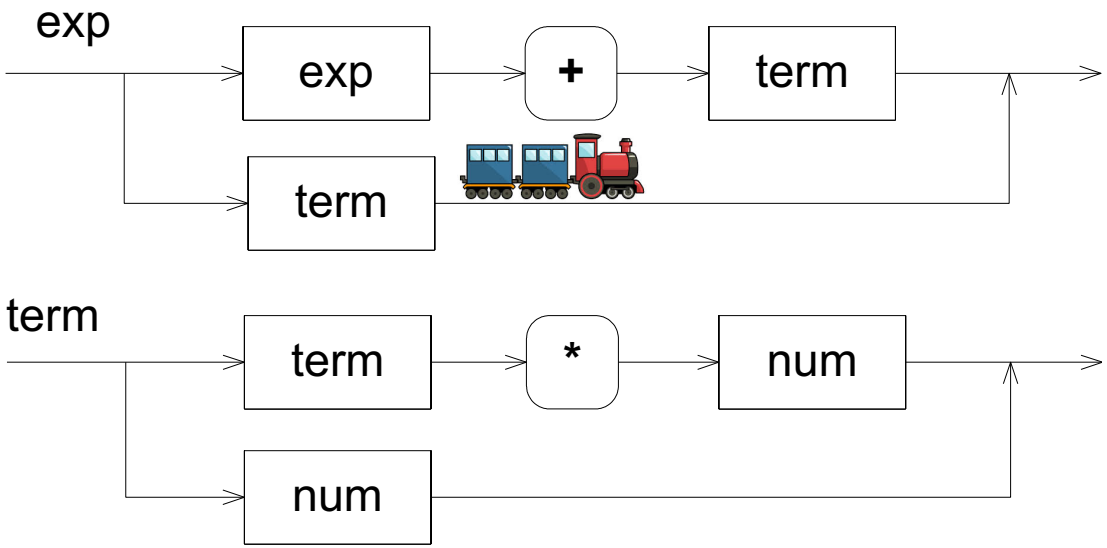
```
s ::=      v := e  | s ; s  |
  if b then s | if b then s else s
v ::= x  | y  | z
e ::= v  | 0  | 1 | 2 | 3 | 4
b ::= e = e
```

Alternatives to EBNF grammars for describing syntactic program structure

- Syntax diagrams
- Metamodels
- Automata/State Machines

Syntax diagram

- Older textbooks and reference manuals often had this kind of notation for syntax
- *«Jernbanediagram»*



Corresponding EBNF:

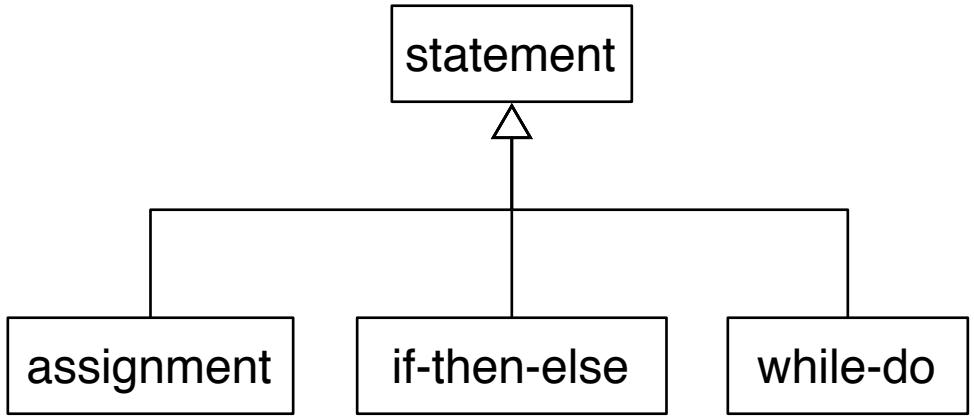
```
exp  ::= exp + term | term

term ::= term * num | num
```

Metamodels

Object model representing the program (*not* the execution)

statement ::= assignment | if-then-else | while-do

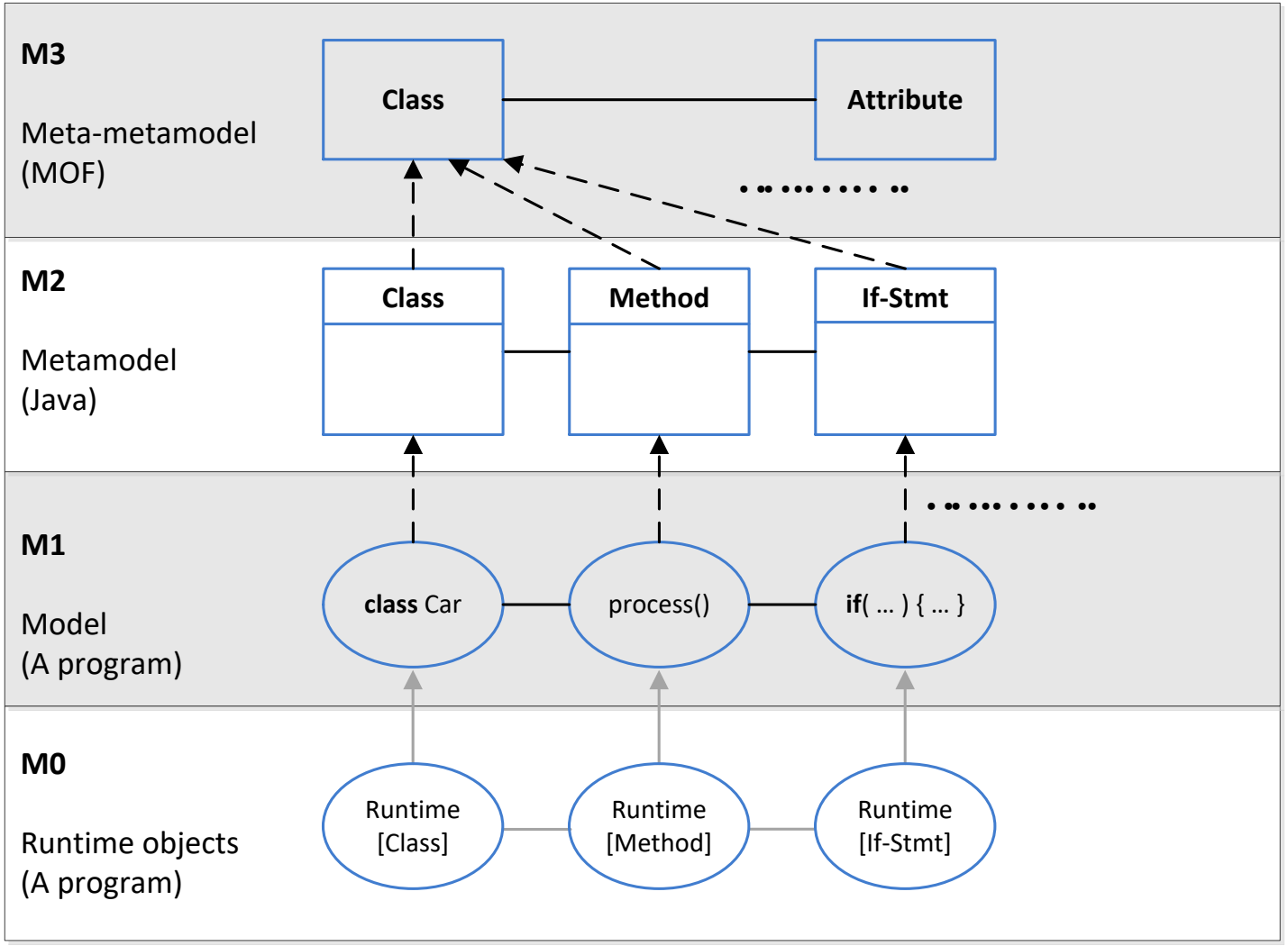


Why metamodels for language descriptions?

- Inspired by abstract syntax trees in terms of object structures, interchange formats between tools
- Growing interest in domain specific languages, often with a mixture of text and graphics
- Meta models often include name, binding and type information in addition to the pure abstract syntax tree
  - «annotated syntax tree»
  - Very useful when constructing type checkers, compilers etc

Metamodel levels
(details of this slide is not on the curriuculum, no need to worry ☺)

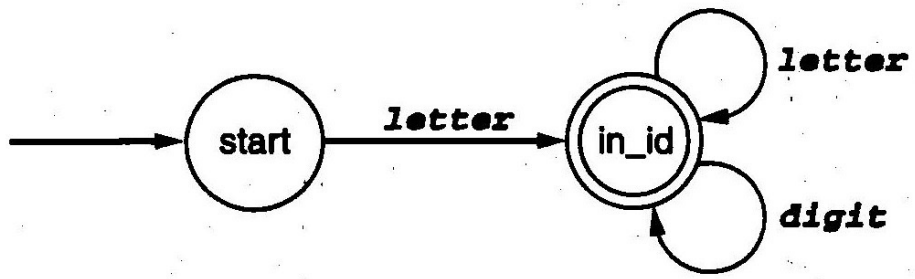
Model conformance: M2 → M3, M1 → M2

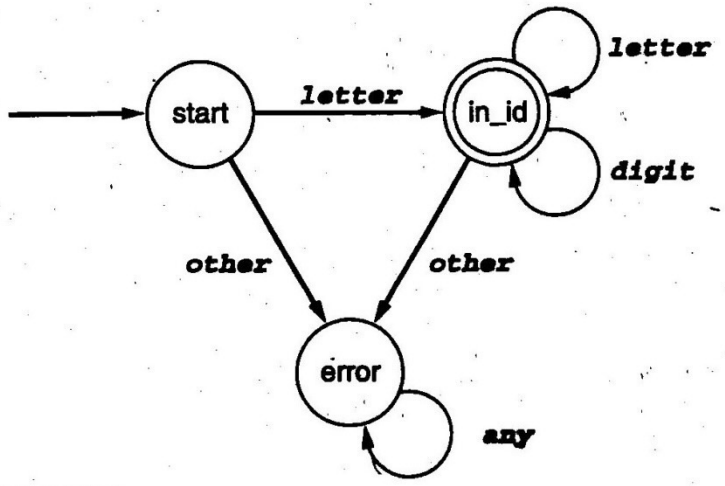


Automata/State Machines

- Transitions marked with *terminals*, one *start state* and a number of *stop states*
- Recognizes a string in the language if the terminals represent a valid sequence of transitions ending up in a stop state upon reading the last symbol
- Typically used for the part of the grammar that recognizes the smallest elements (tokens), called the *scanner*

identifier ::= letter { letter | digit }*





Parsing – using the grammar

- To check that a sentence (or a program) is syntactically correct, that is to construct the corresponding syntax tree.
  - Like we have seen some slides back
- In general we would like to construct the tree by reading the sentence once, from left to right.
- Example grammar

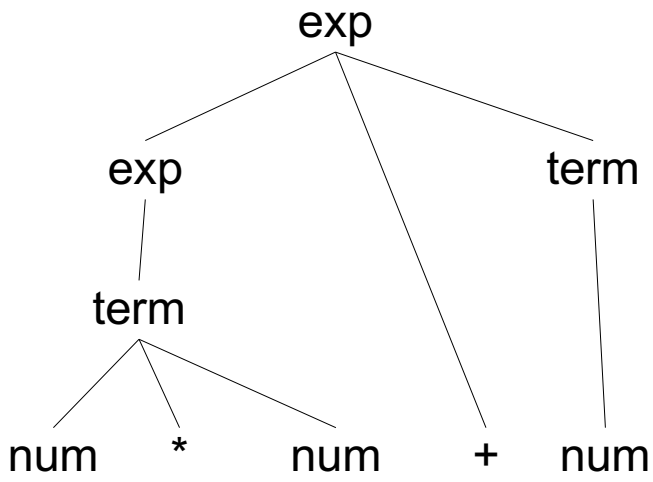
```
exp  ::= exp + term | term
term ::= term * num | num
```

Top-down parsing

The parse tree is constructed downwards, that is we start with the start symbol and try to derive the actual sentence by selecting appropriate rules:

```
exp  ::= exp + term | term

term ::= term * num | num
```

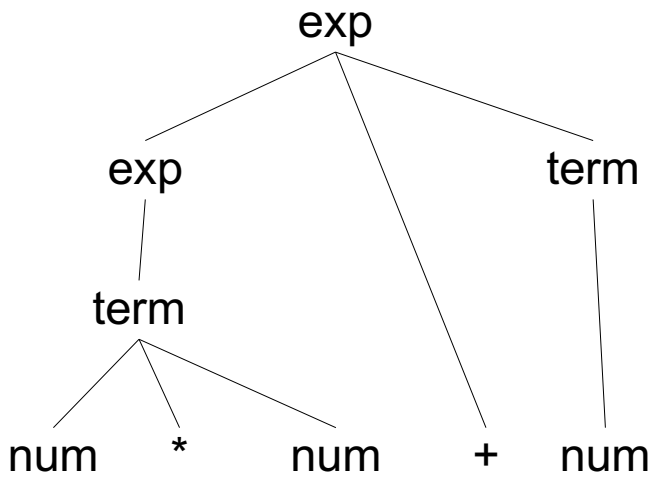


Bottom-up parsing

The tree is constructed upwards. Starts by finding part of the sentence that corresponds to the right hand side of a production and reduces this part of the sentence to the corresponding nonterminal. The goal is to reduce until the start symbol.

```
exp  ::= exp + term | term

term ::= term * num | num
```



LL(1)-parsing – a practical approach

- LL(1)-parsing is a top-down strategy with a ***left derivation*** from the start symbol (the **leftmost** symbol).
  - A common approach to parsing that is simple and efficient
  - The number 1 is for 1 single lookahead-token (from the scanner)

- Recursive descent – LL(k)
  - To each *nonterminal* there is a method.
  - The method takes care of the rule for for this nonterminal, and may call other methods.
    - For each *terminal* in the right hand side: Check that the next token (from the scanner) is this terminal.
    - For each *nonterminal* in the right hand side: Call the corresponding method.
  - When the method is called, the scanner shall have as its next token the first token of the corresponding rule.
  - When the method is finished, the scanner shall have as its next token the first token after the sentence.

Example – recursive descent parser

program ::= stmtList
stmtList ::= stmt +
stmt ::= input | output | assignment
input ::= ? variable
output ::= ! variable
assignment ::= **variable = variable operator operand**
operator ::= + | -
operand ::= variable | number
variable ::= v digit
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
number ::= digit +

```
void assignment() {
    variable();
    readToken('=');
    variable();
    operator();
    operand();
}
```

Example – recursive descent parser

program ::= stmtList
stmtList ::= stmt +
stmt ::= **input | output | assignment**
input ::= ? variable
output ::= ! variable
assignment ::= variable = variable operator operand
operator ::= + | -
operand ::= variable | number
variable ::= v digit
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
number ::= digit +

```
void stmt() {
   if (checkToken('v')) {
      assignment(); }
   else if (checkToken('?')) {
      input(); }
   else if (checkToken('!')) {
      output(); }
   else error();
}
```

Summary

- A programming language has syntactic and semantic rules
- The syntactic rules control the (textual) shape of legal programs
  - A parser recognizes program text, and can transform a program text into a parse tree, and an abstract syntax tree
  - A grammar is ambigous if there are more than one way to parse its sentences
- The semantic rules control which textually legal programs have meaning, and what this meaning is
  - What a program will do at runtime cannot be determined by its syntax alone
  - It depends on the semantics!
  - More on this later

- Next week: Logic programming with Volker

- Until then: Have a great weekend!

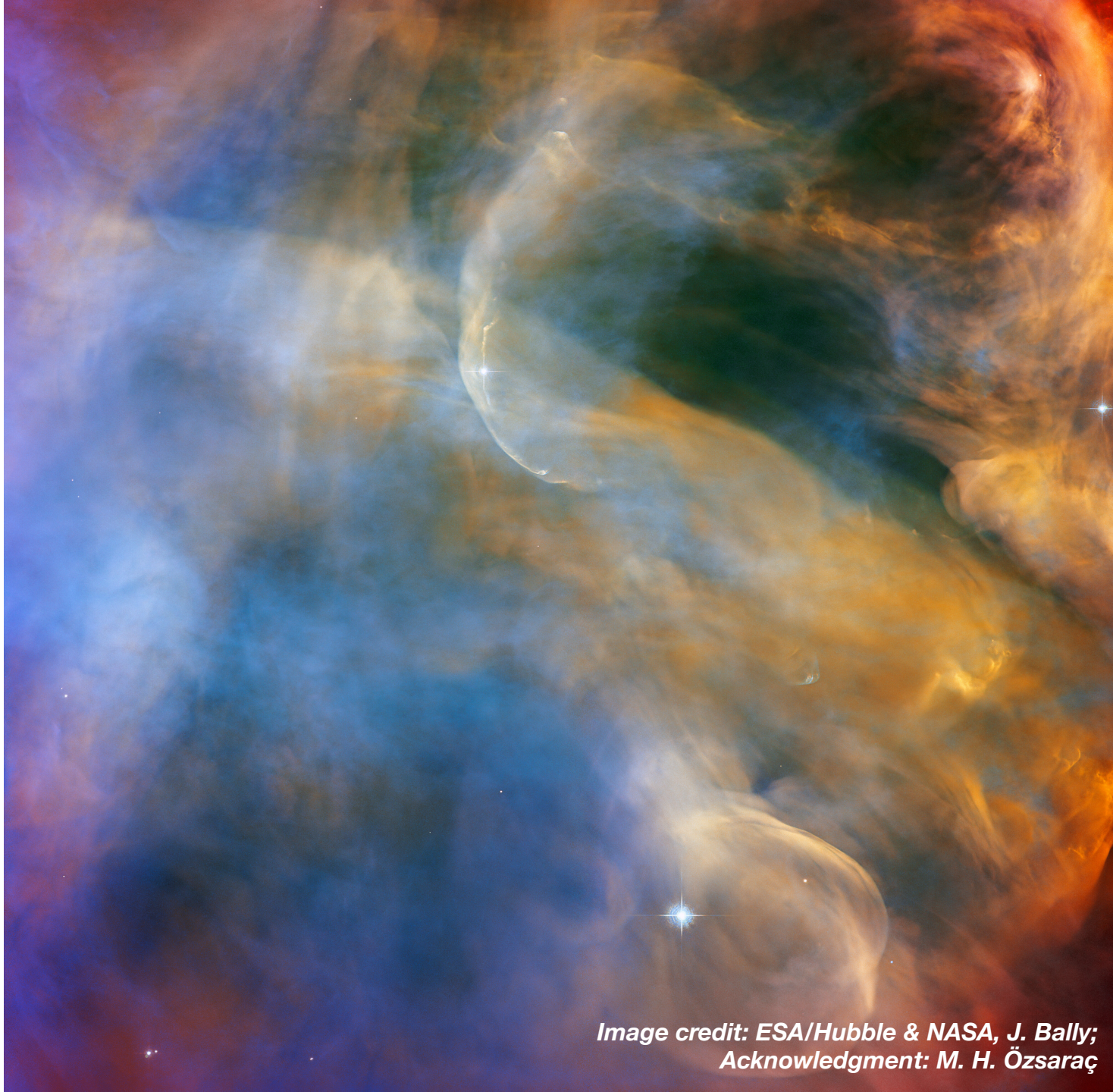


Image credit: ESA/Hubble & NASA, J. Bally; Acknowledgment: M. H. Özsaraç