INF3110 – Programming languages
Syntax and Semantics

Eyvind W. Axelsen

eyvinda@ifi.uio.no | @eyvindwa

http://eyvinda.at.ifi.uio.no

Slides adapted from previous years' slides made by Birger Møller-Pedersen

birger@ifi.uio.no

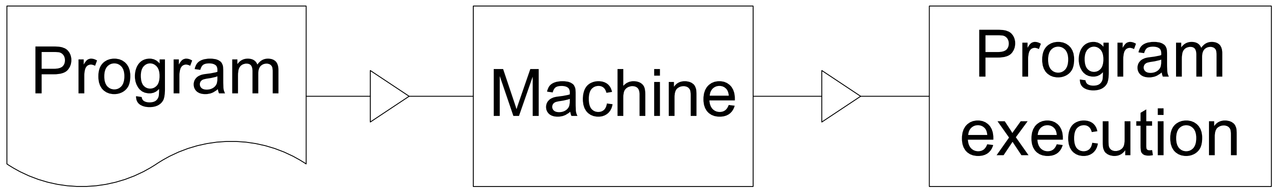
Plan

- Today
  - Syntax and semantics
  - Jumpstart Types and OO? (time permitting)
- Next week (August 31st)
  - Types and OO proper
  - Mandatory exercise 1 posted
- September 7th
  - Daniel Schnetzer Fava, SML and functional programming

Outline: Syntax and semantics

- Program != program execution
- Compiler/interpreter
  - This is not a compiler course…
  - …but some basic knowledge of language constructs is needed
    - Will be provided!
- Syntax
  - Grammars
  - Syntax diagrams
  - Automata/State Machines
  - Scanning/Parsing
- Meta-models

Program != program execution



Syntax

Semantics

Important topics for this course:

- Understand design tradeoffs in PL design
- Understand how programs are executed
- 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?

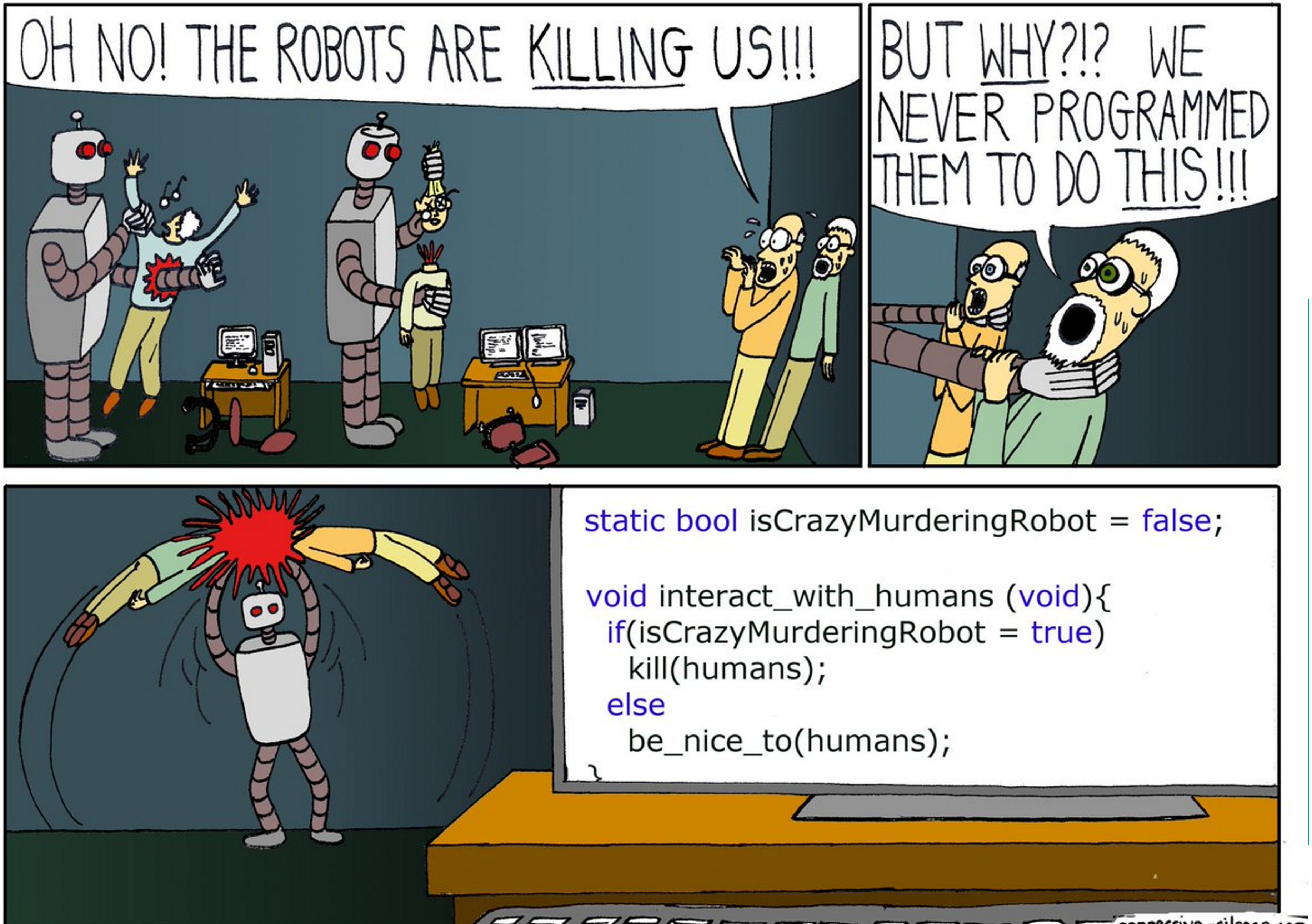
What will be printed?

Syntax != Semantics

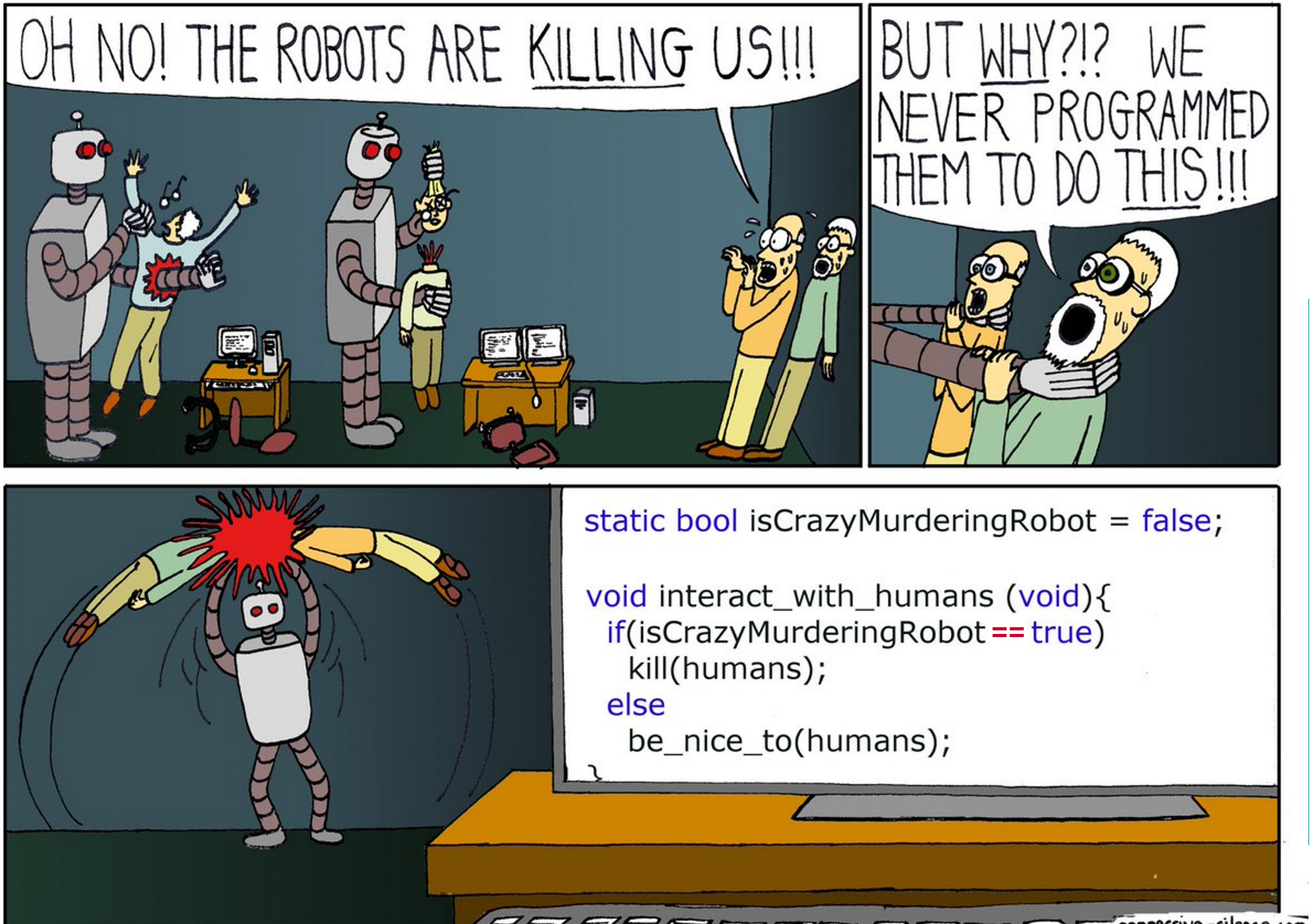
- A description of a programming language consists of two main components:
  - *Syntactic* rules
    - What *form* does a legal program have.
  - *Semantic* rules:
    - Which programs are meaningful?
    - What do the sentences (of meaningful programs) in the language mean?

  - *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 behaviour of an (idealised) abstract machine performing a program,
      - Or, mapping to something else (but well-known and well-defined) - *denotational* semantics.

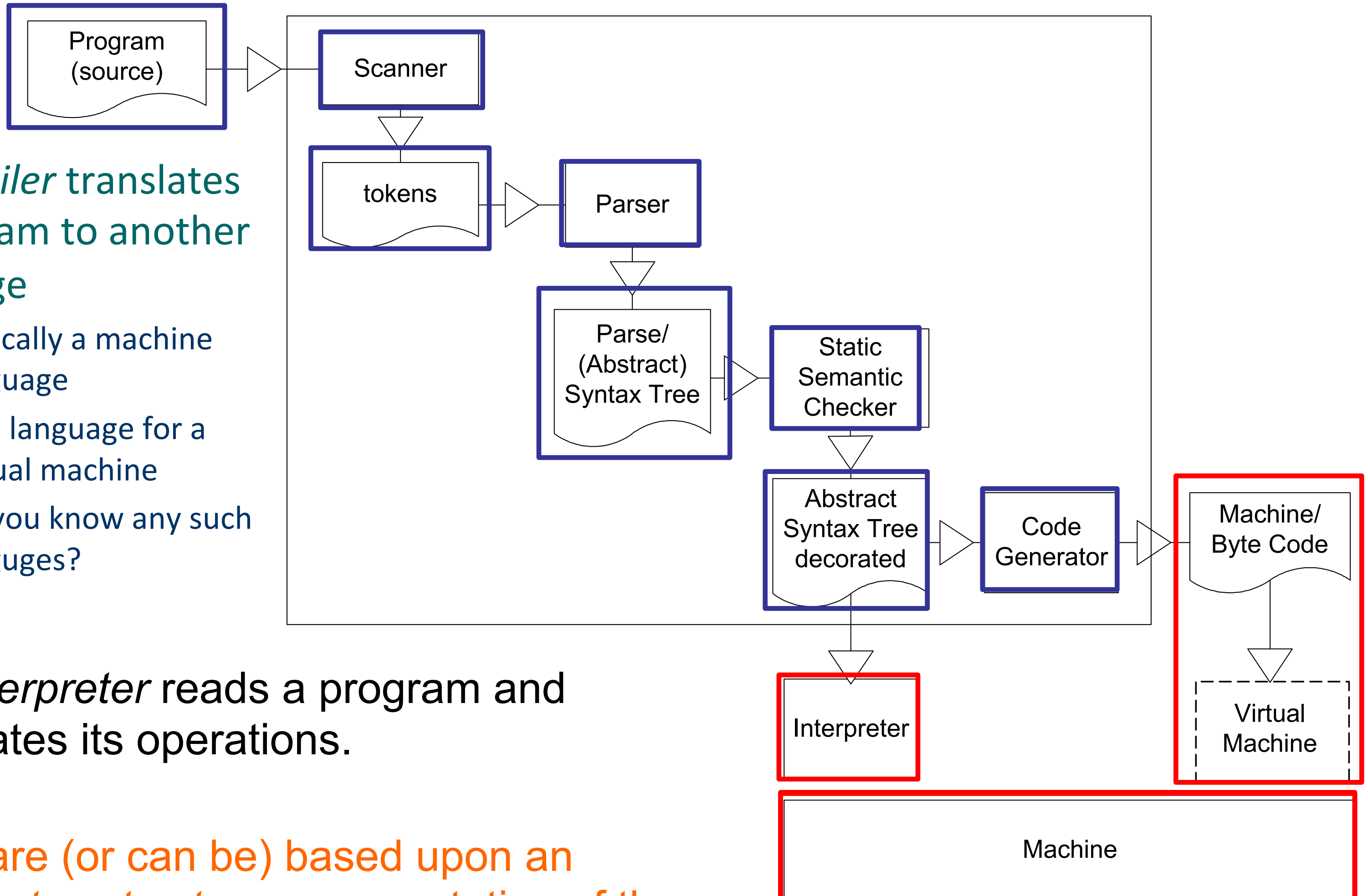
Syntax matters!



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 langugues?

- 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”

- 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: Described by BNF-grammars

```
e ::=  n
e ::=  e + e
e ::=  e - e
n ::=  d
n ::=  nd
d ::=  0
d ::=  1
...
d ::=  9
```

production, rule

nonterminal

terminal

terminal

metasymbol

meta-language

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
```

Aside: A question that some may have

- What is the difference between a Context-Free Grammar (CFG) and BNF?
  - A CFG is (informally) a grammar where all the rules are one-to-one, one-to-many or one-to-none.
  - The left hand side of a rule in a CFG contains one (and only one) non-terminal symbol, and no terminal symbols (thus, no *context* → *context-free)*
  - This is the way rules are expressed in BNF too!
- Thus, BNF is a *notation* for CFGs.
  - Other notations are possible
  - Notably «Van Wijngaarden form»

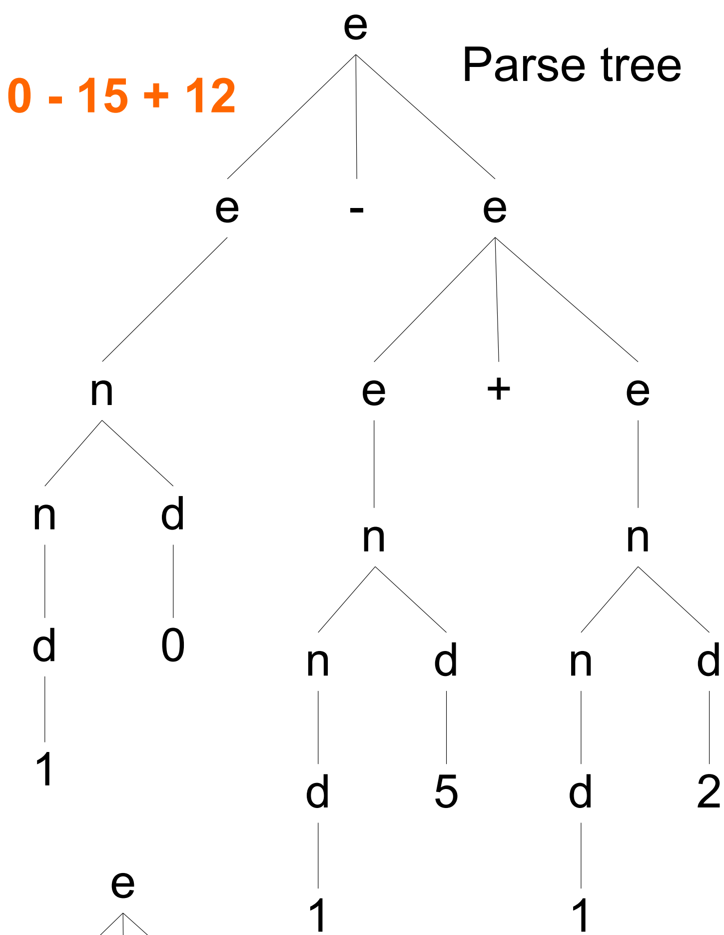
Derivation of sentences

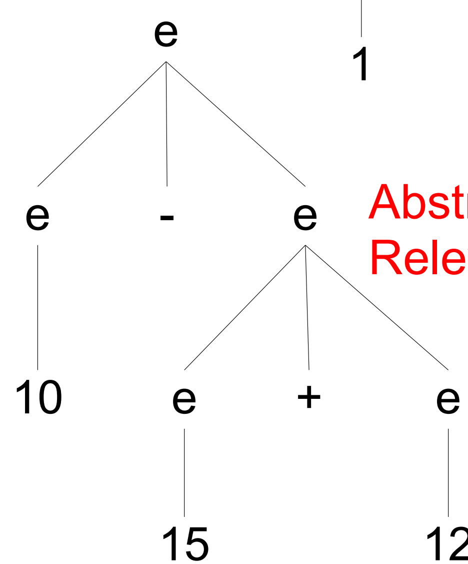
- The possible sentences in a language defined by a BNF-grammar are those that emerge by following this procedure:
    1. Start with the start symbol (e).
    2. For each nonterminal symbol (e, n, d) exchange this with one of the alternatives on the right hand side of the production defining this nonterminal.
    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
```

10 - 15 + 12

Parse tree



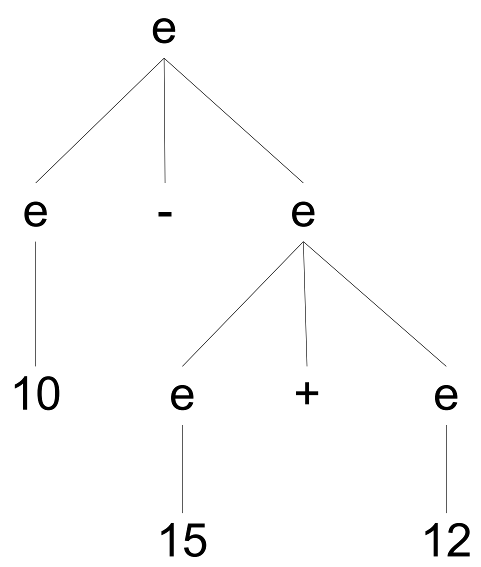


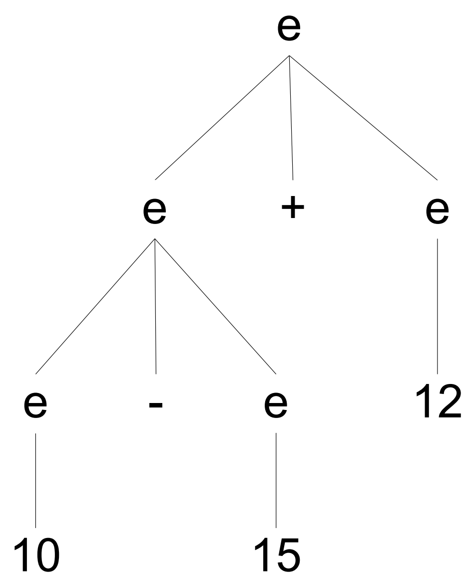
Abstract syntax tree
Relevant for oblig!

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

Only one possible production?

10 - 15 + 12 = ?





10 – (15 + 12) =
10 – 27 =
-17

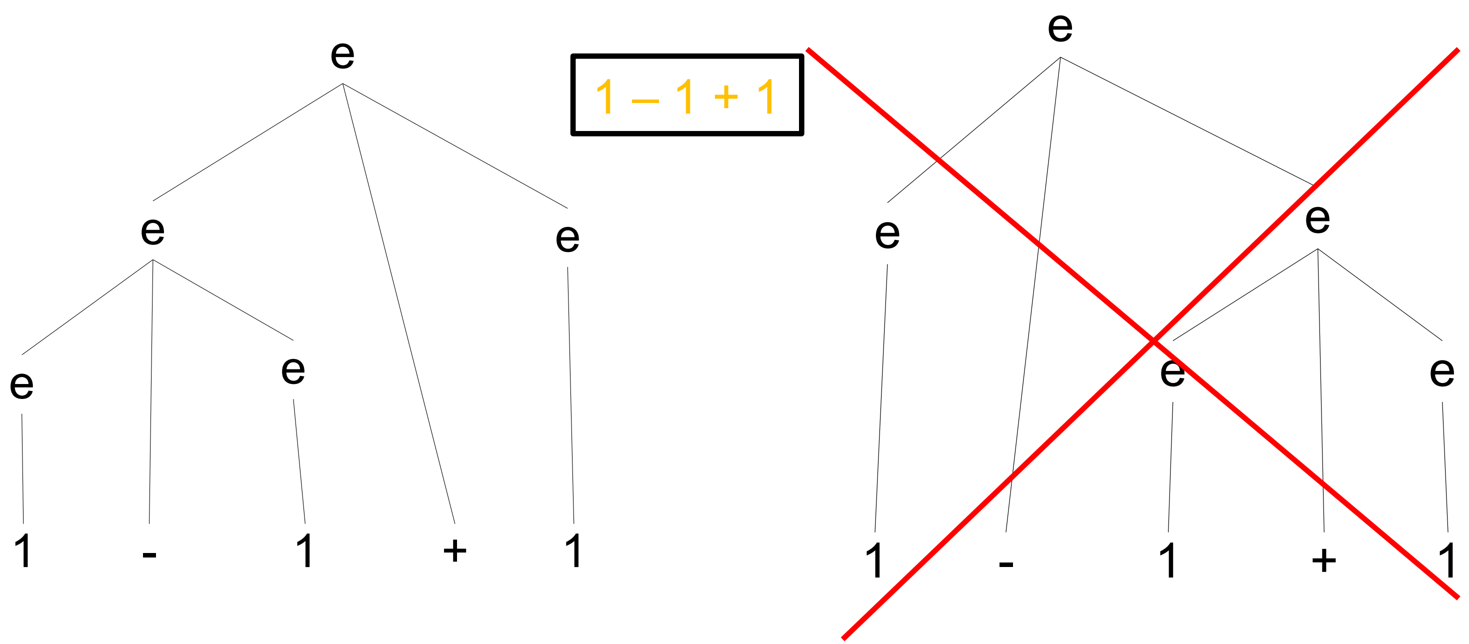
(10 – 15) + 12 =
-5 + 12 =
7

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**.

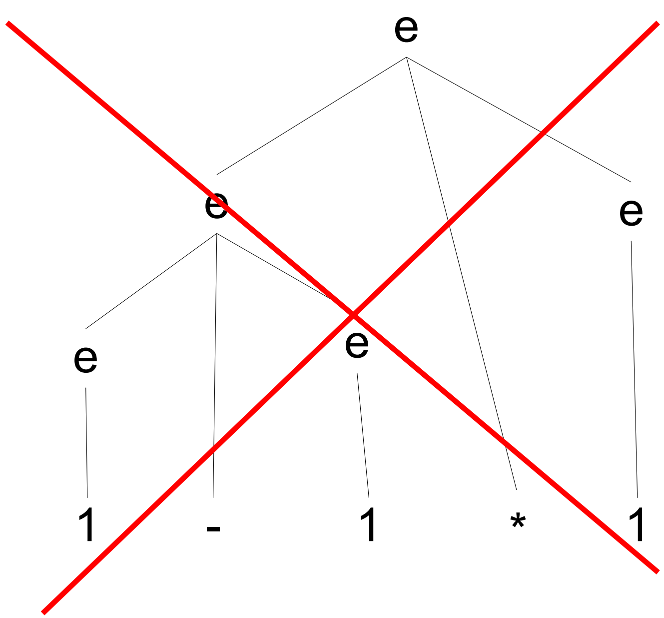
```
e ::= 0 | 1 | e + e | e - e | e * e
```

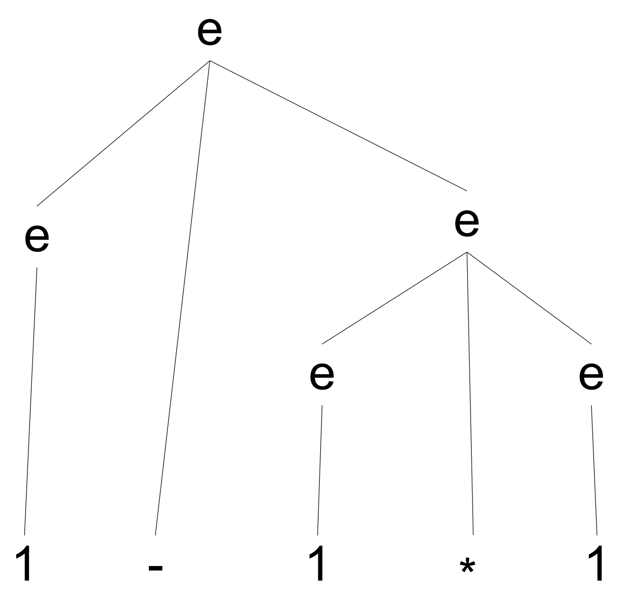
- Ambiguity handled by **associativity** and **precedence** rules



Which is «correct»?

1 – 1 * 1
Which is «correct»?

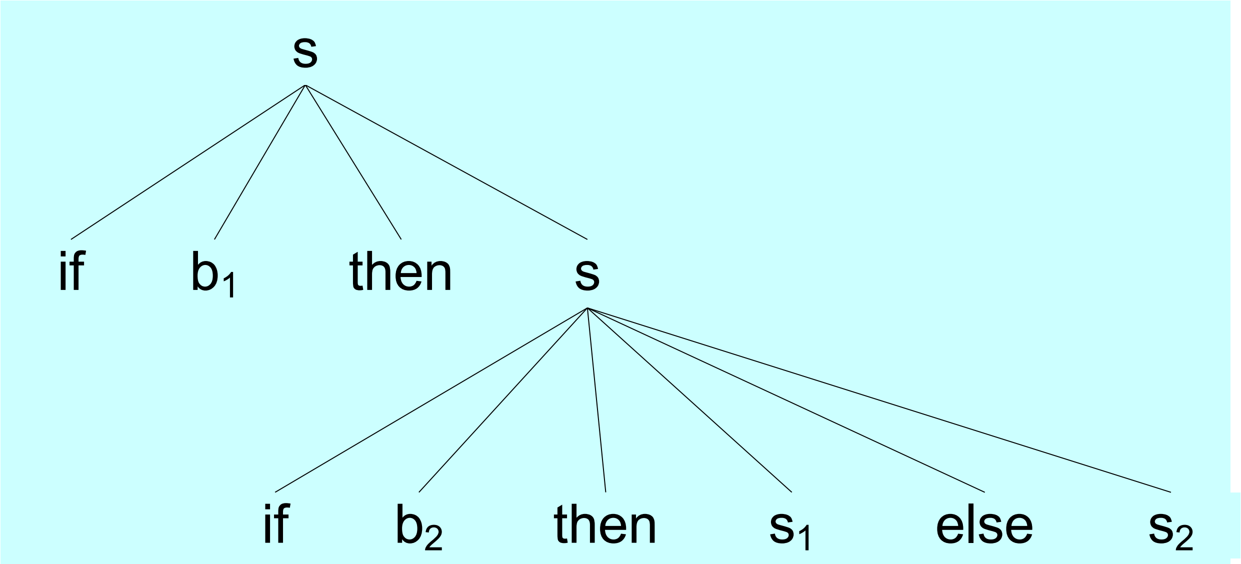


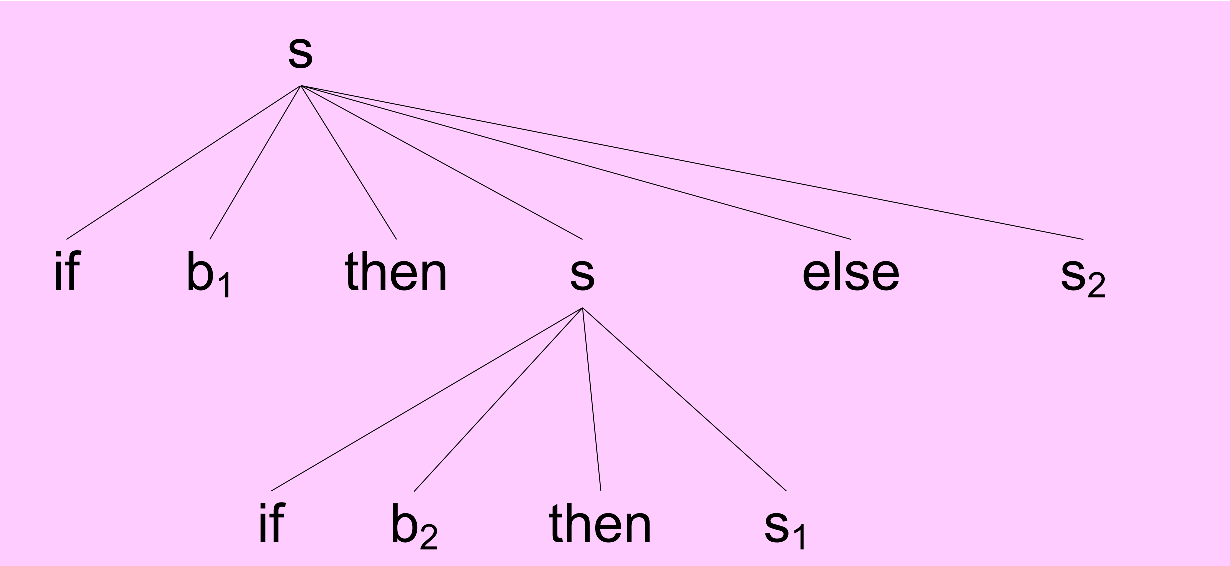


A somewhat more interesting language

$$
\begin{array}{ll}
s ::= & v := e \mid s\,;\,s \mid \boxed{\text{if } b \text{ then } s \mid \text{if } b \text{ then } s \text{ else } s} \\
v ::= & x \mid y \mid z \\
e ::= & v \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \\
b ::= & e = e
\end{array}
$$

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

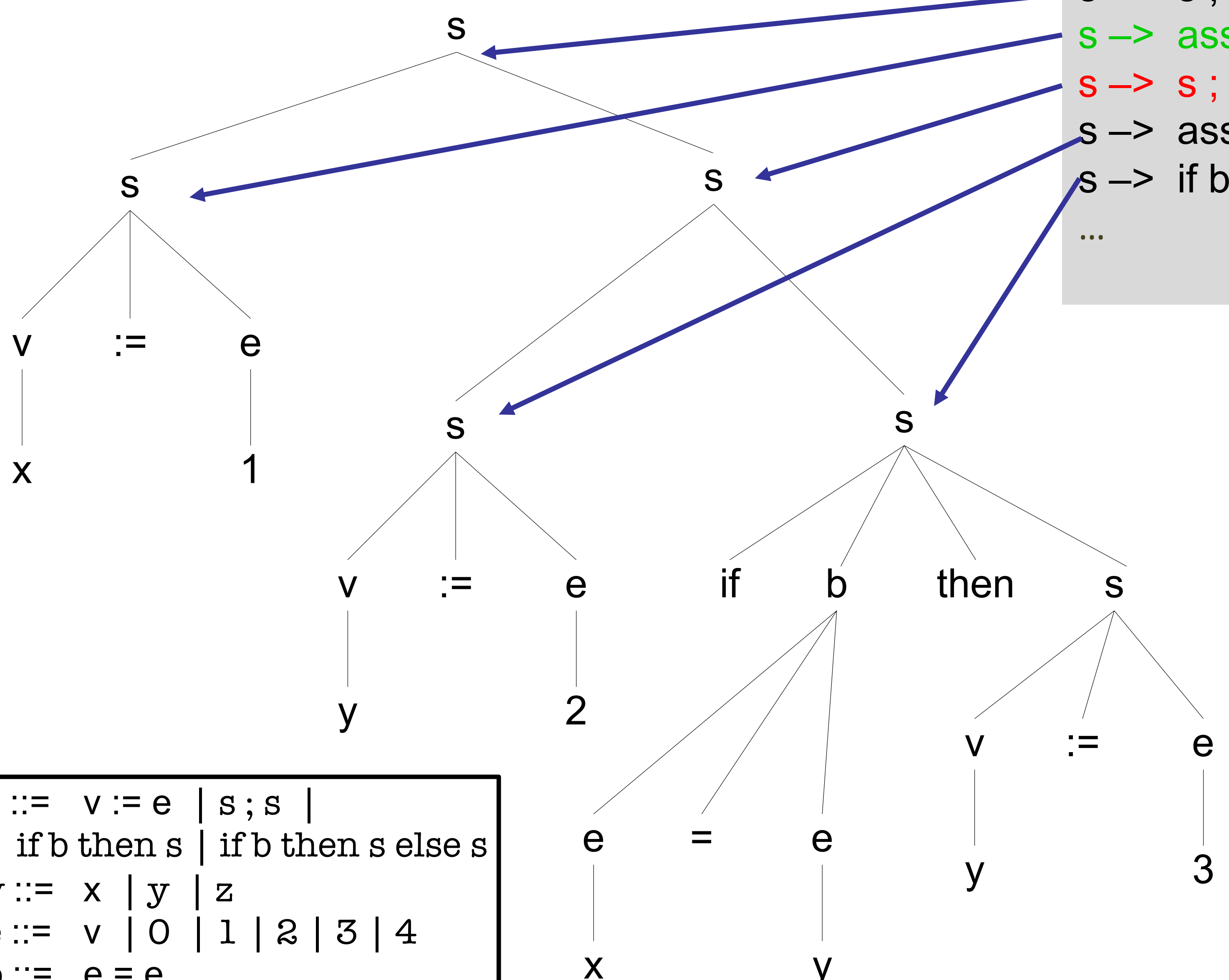
b_1 = false
b_2 = true

no

yes

Parsing matters!

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



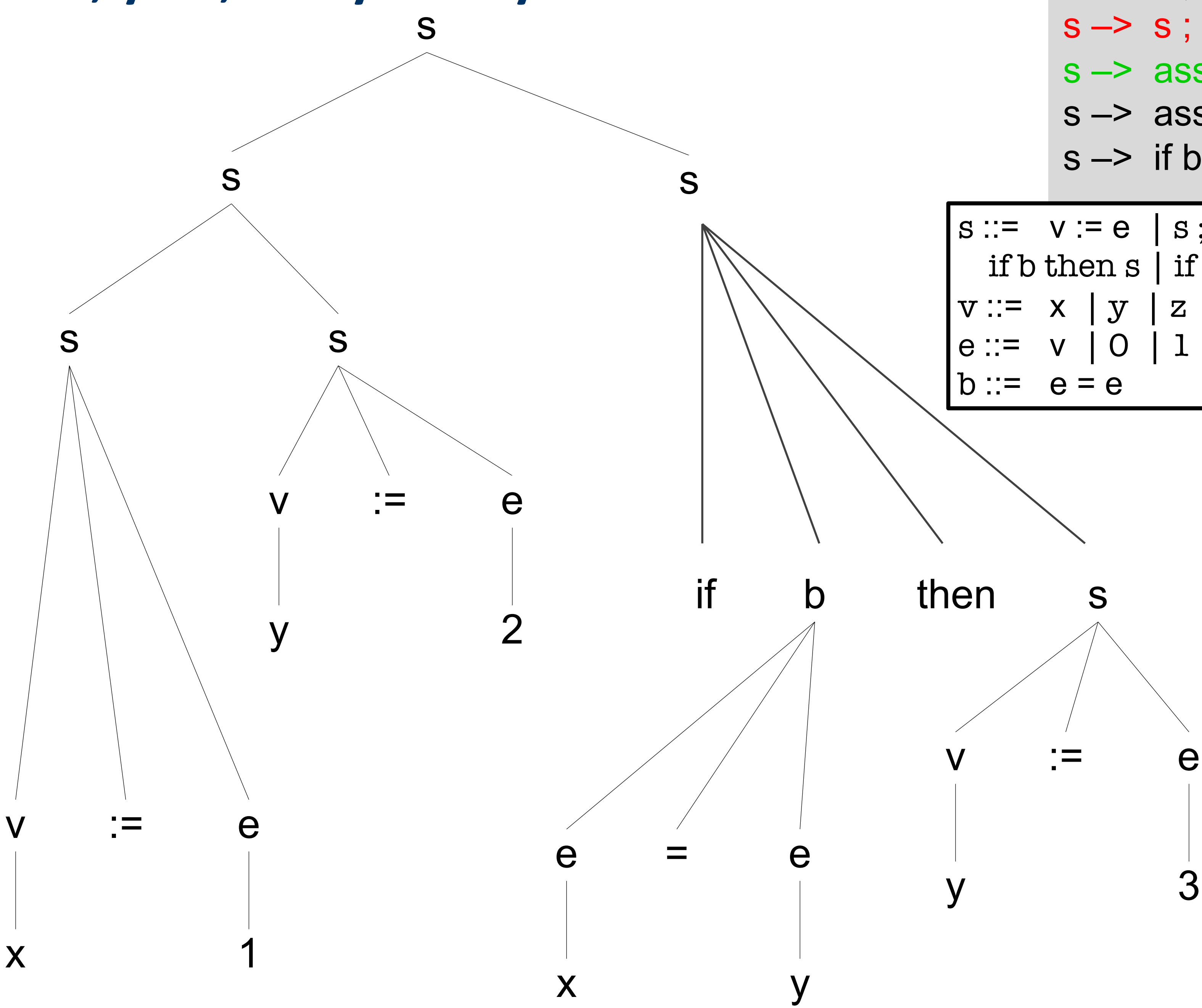
s –> s ; s
s –> assign (x := 1)
s –> s ; s
s –> assign (y := 2)
s –> if b then s
...

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

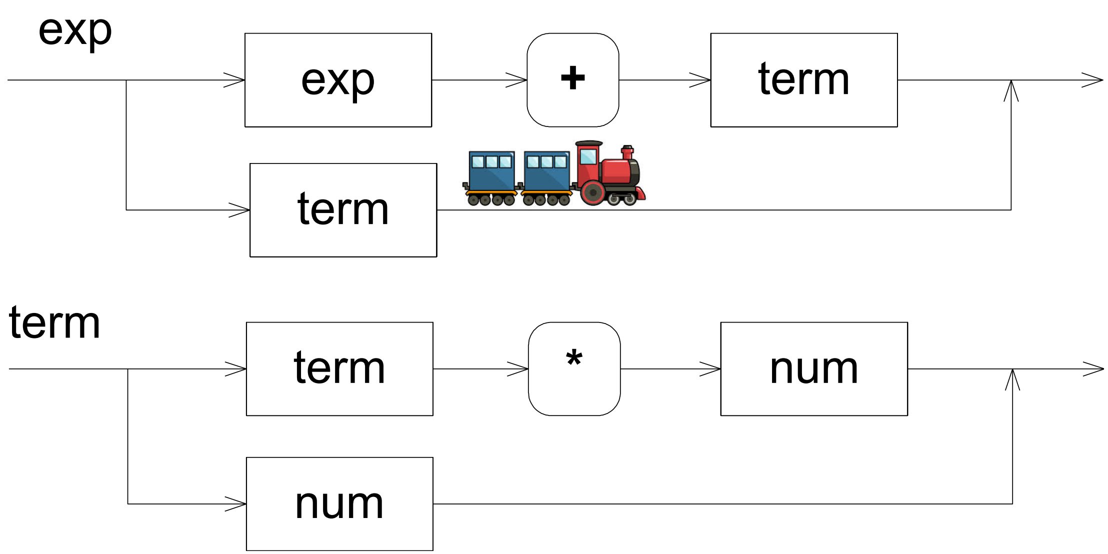


Alternatives to EBNF grammars

- Syntax diagrams
- Meta-models
- Automata/State Machines

Syntax diagram

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

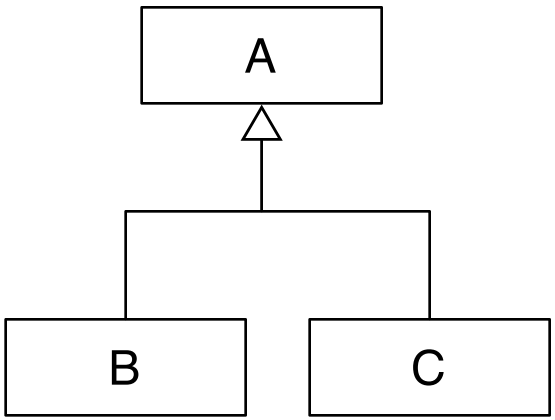


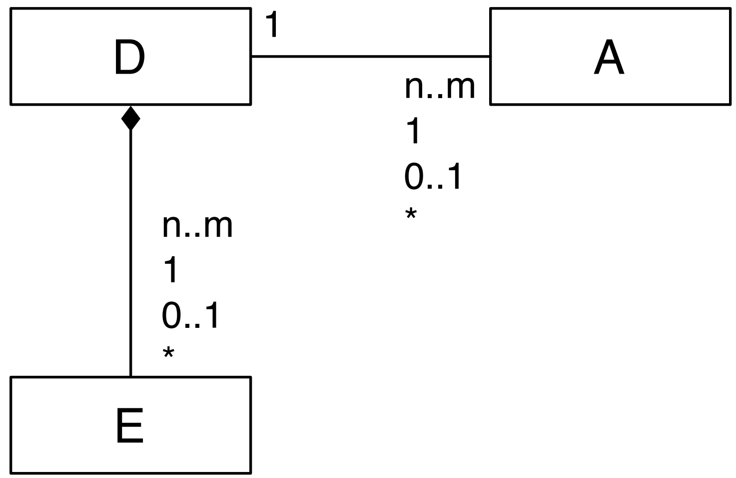
exp ::= exp + term | term

term ::= term * num | num

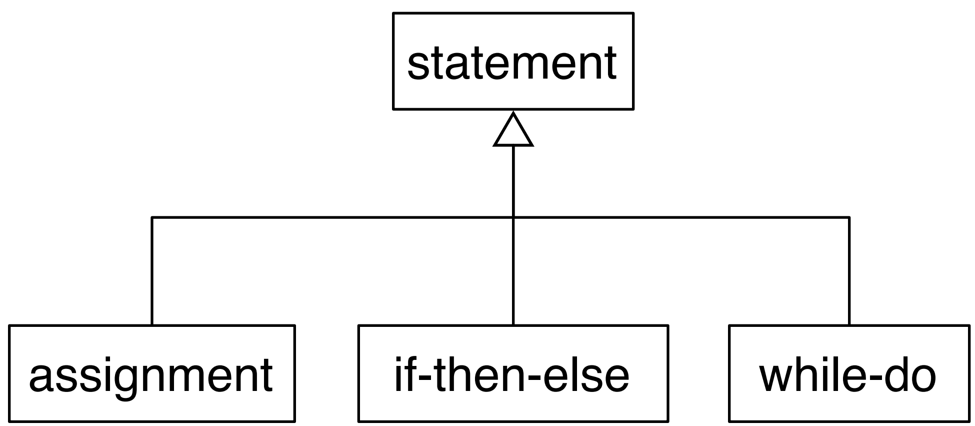
Meta-models

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





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



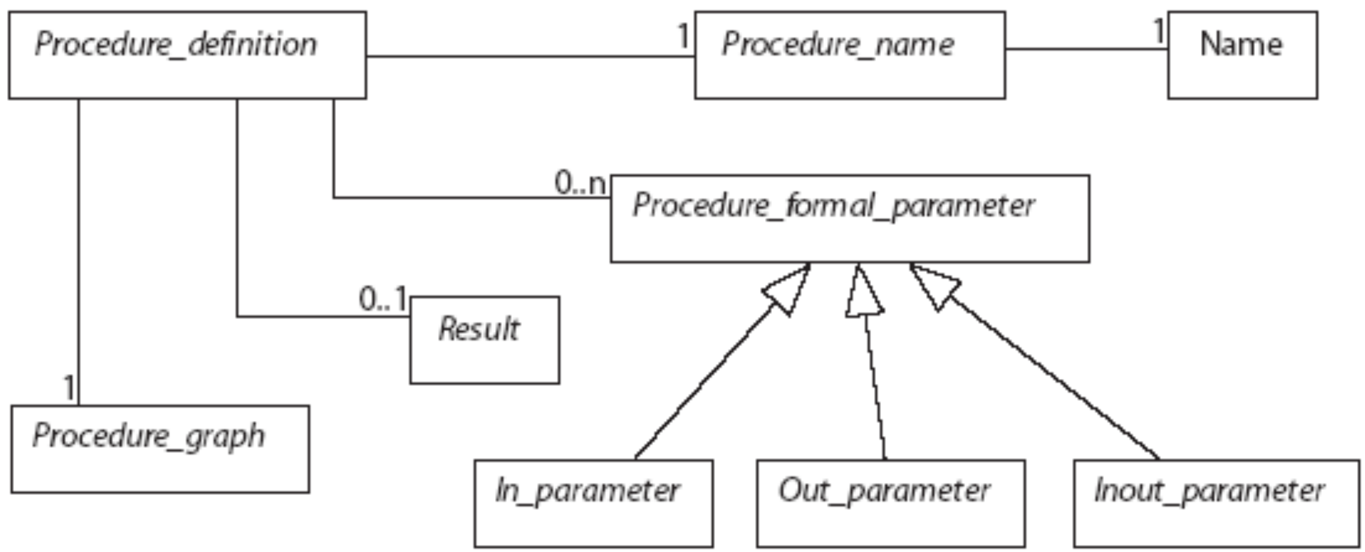
Why meta models?

- Inspired by abstract syntax trees in terms of object structures, interchange formats between tools
- Not all modeling/programming tools are parser-based (e.g. wizards)
- 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»

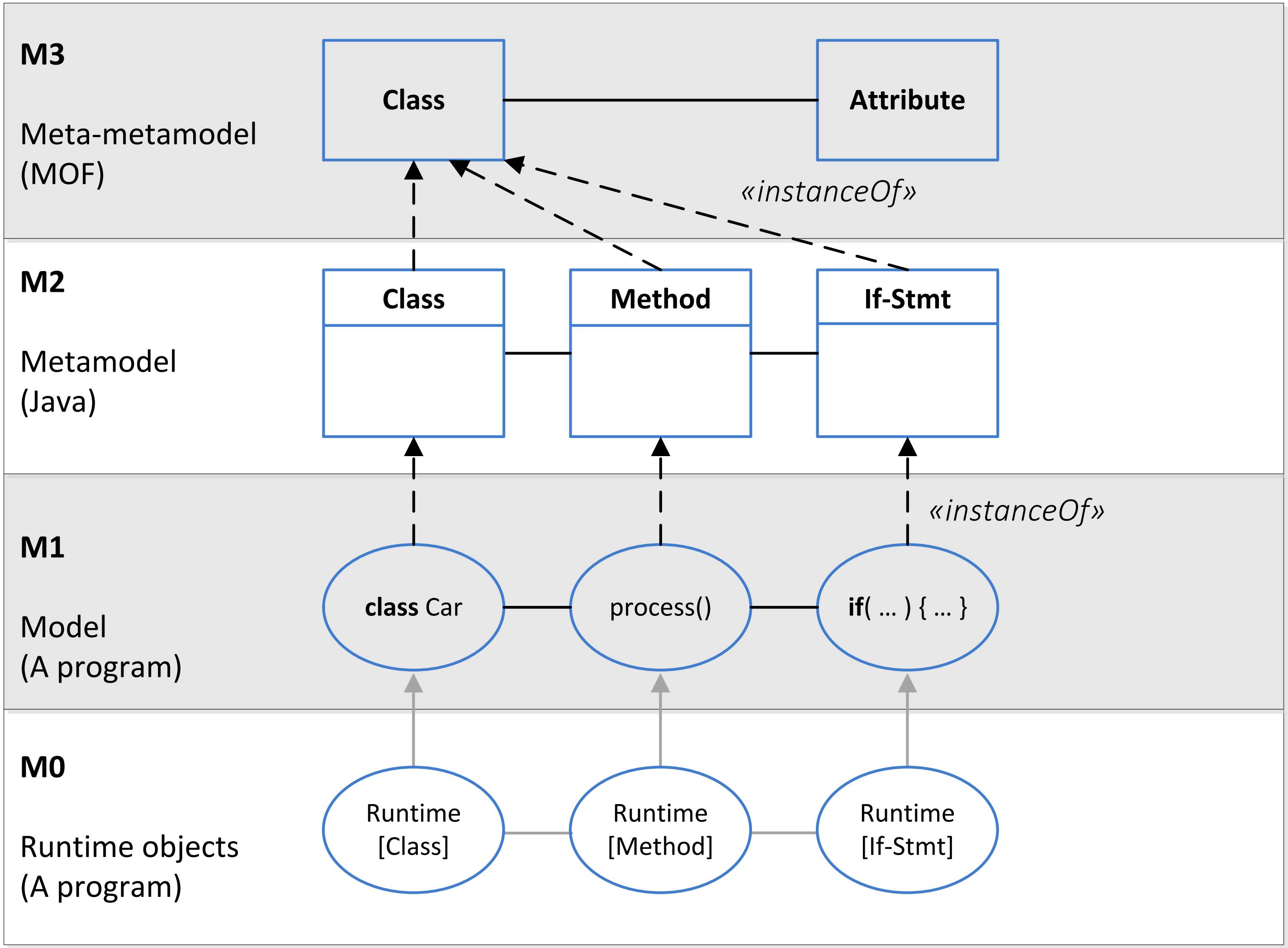
Example Metamodel

```
Procedure_definition :: Procedure_name
    Procedure_formal_parameter*
    [ Result ]
    Procedure_graph;

Procedure_name = Name;

Procedure_formal_parameter = In_parameter
    | Inout_parameter
    | Out_parameter;
```



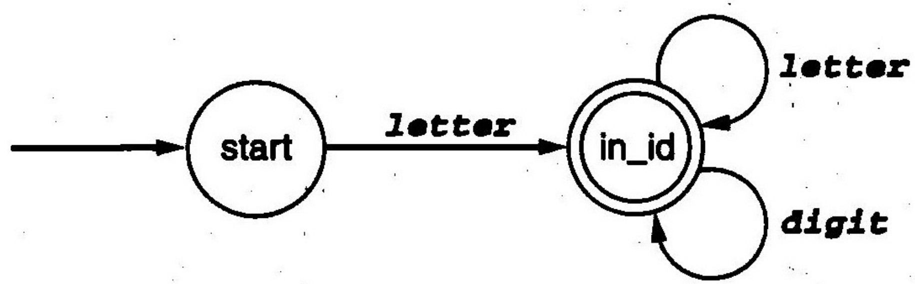
Metamodel levels
(details of this slide is not on the curriuculum, no need to worry ☺)

Model conformance: M2 → M3, M1 → M2

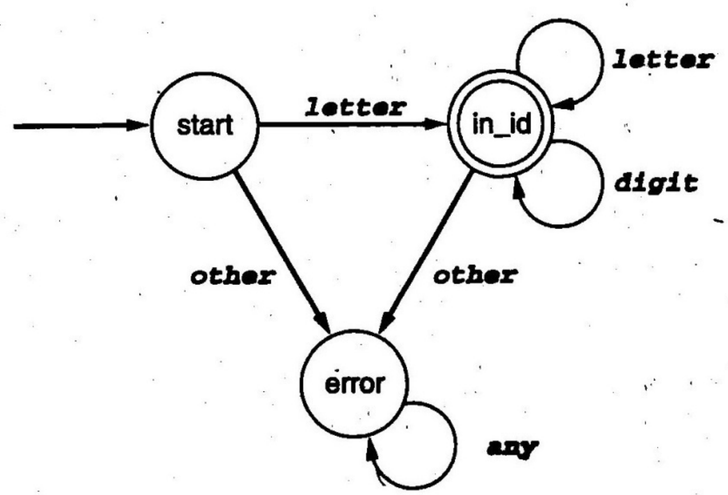


Automata/State Machines

- 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 }*

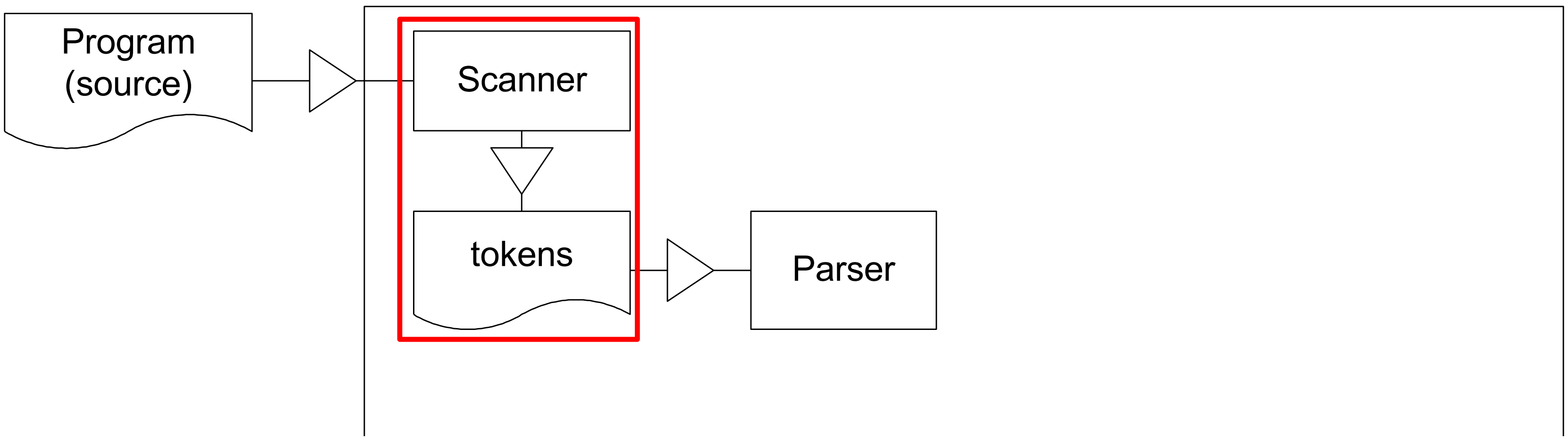


Disclaimer

- This is not a compiler course, but on the following slides, we'll briefly look at some central compiler concepts

Image from https://www.teeturtle.com/

Scanning



- A scanner groups relevant characters to symbols called *tokens*

```
begin
        OutText("Hello")
end
```

Token:	BEGIN	IDENT	LPAR	TEXT	RPAR	END
Value:	begin	OutText	(	”Hello”	)	end

- A scanner is normally constructed as an automata/state machine

Parsing

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

- Example grammar

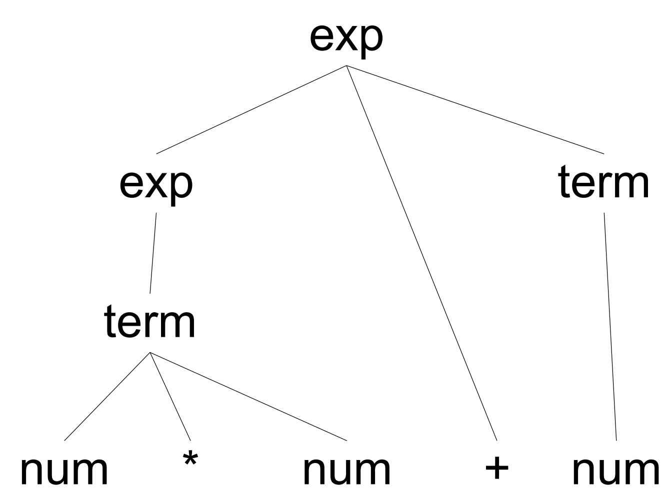
```
exp  ::= exp + term | term
term ::= term * num | num
```

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

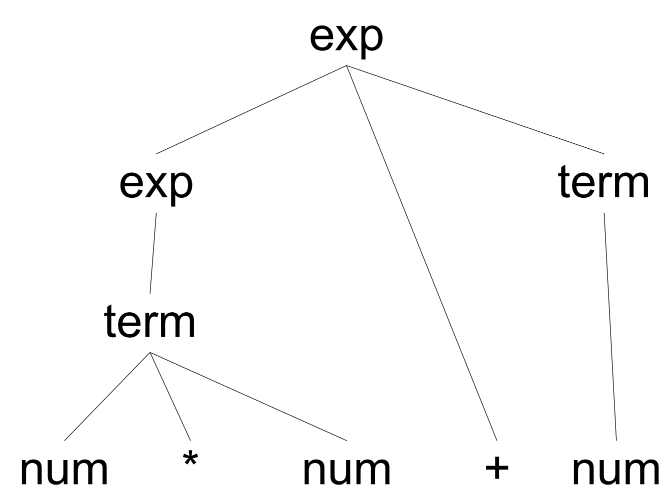


Bottom-up parsing

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

- LL(1)-parsing is a top-down strategy with a *left derivation* from the start symbol (the **l**eftmost symbol).
  - A common approach to parsing that is simple and efficient
- 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(); }
}
```

Exercises

1. Mandatory
   - Mandatory exercise will be out next week
   - Make an interpreter for the ROBOL language, a simple robot language that supports moving around on a grid
   - Shall be written in both Java?? (OO) and SML (functional programming)
2. Weekly exercises
   - On the lecture plan, will be explained in the group sessions.