Lecture 2: Syntax

v3, 17th January 2024

Programming Languages (H)

Simon Fowler & Michele Sevegnani

Semester 2, 2024/2025

sli.do 4793883

University of Glasgow

sli.do 4793883

Today

- This lecture: what does a programming language look like, and how do we specify this formally?
- **Today's topics:**
  - Concrete and abstract syntax
  - Informal vs. formal specification
  - Regular expressions, Backus-Naur Form grammars
  - Ambiguity
- I want to stress from the outset: syntax is an important part of a PL design, but there is **far more** to PLs than syntax and parsing.

sli.do 4793883

Let's start with a small language…

- Let's start with a small programming language for arithmetic expressions, which we'll call L_{arith}
  - This is a very minimal language! We'll extend it through the course.

- In this language, we can have:
  - Integers
  - Two integers added together
  - Two integers subtracted from each other
  - Two integers multiplied together
  - Two integers divided by each other

sli.do 4793883

Yes!

```
1 + 3        4 * 5        5
```

But…

```
Add(1 , 2)                  3 times five
Subtract 1 from 2            1 + 2 + 3
```

The description is simultaneously:

- **Too restrictive** (it doesn't allow nested expressions)
- **Too permissive** (it doesn't specify the exact way of stating numbers or operations)

sli.do 4793883

Fine, how about this?

- In this language, an expression can be:
  - Integer literals
  - Two expressions added together using the ‘+’ symbol
  - Two expressions subtracted from each other using the ‘-’ symbol
  - Two expressions multiplied together using the ‘*’ symbol
  - Two expressions divided by each other using the ‘/’ symbol
  - An expression can be enclosed in brackets

sli.do 4793883

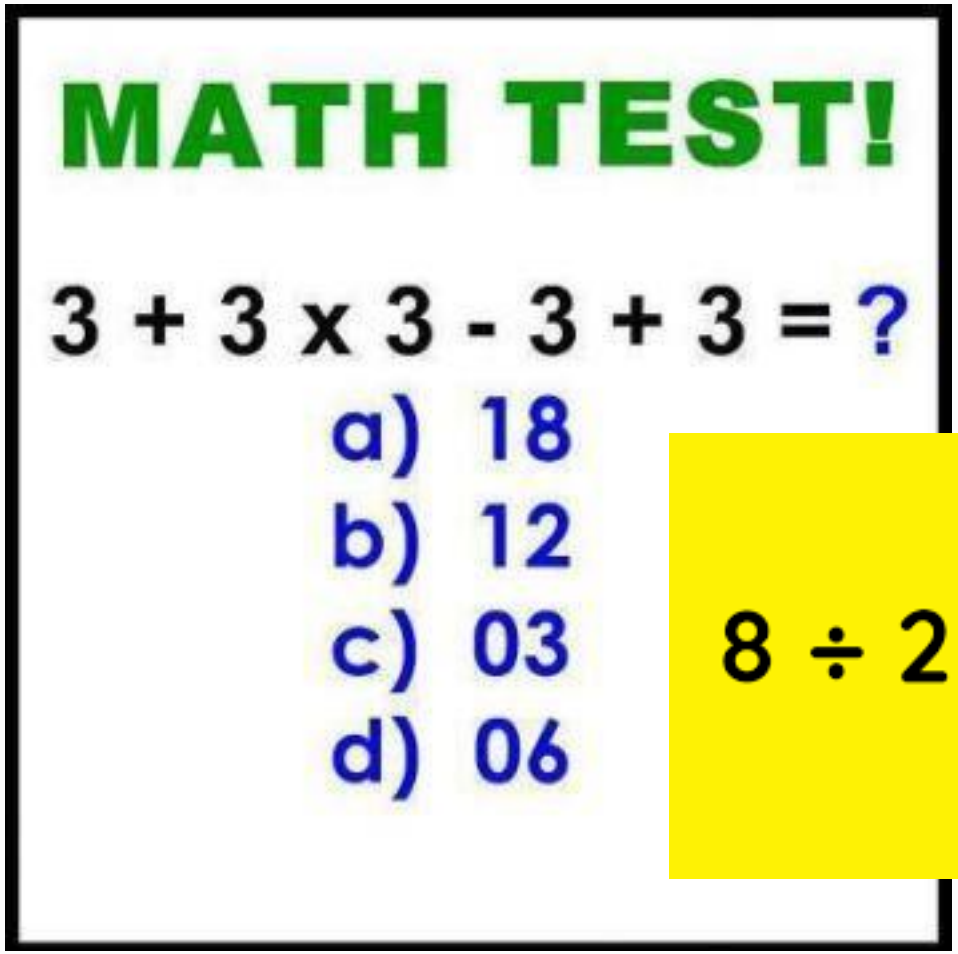
Yes!

```
1 + (2 * 3)   (1 + 2) * (3 + 4)
```

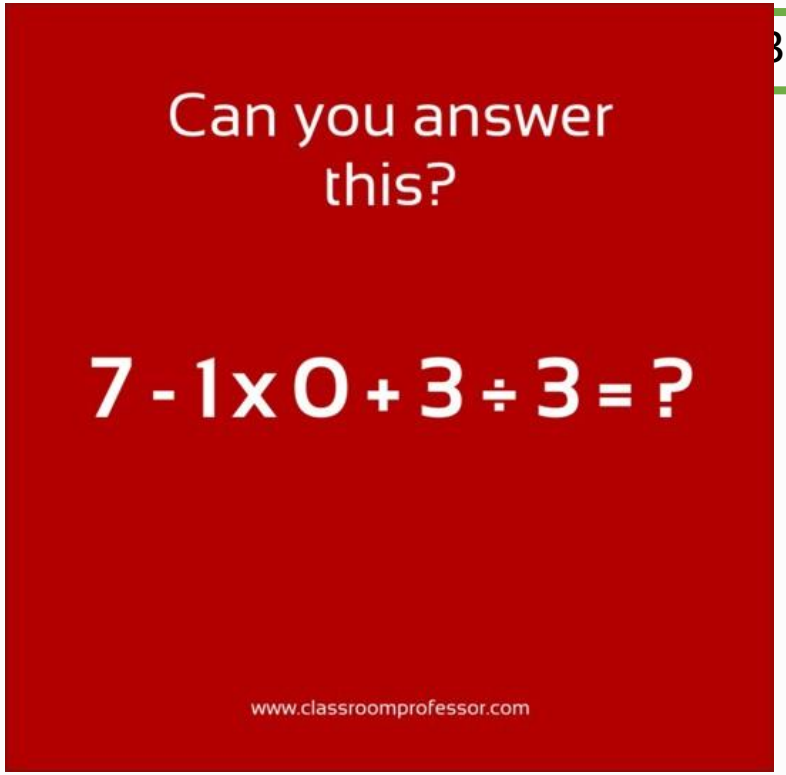
But…

```
1 + 3 * 4 + 5      7 + 8 / 9 + 10
```

Arithmetic operators have a **precedence** (remember BODMAS from school), yet this is not reflected in the description



$8 \div 2 (2 + 2) = ?$



These tedious things on Facebook / Instagram / whatever you young people use nowadays play on **grammar ambiguity**: how do we bracket the expressions?

sli.do 4793883

Fine, how about this?

- In this language, an expression can be:
  - An integer literal
  - Two expressions added together using the ‘+’ symbol
  - Two expressions subtracted from each other using the ‘-’ symbol
  - Two expressions multiplied together using the ‘*’ symbol
  - Two expressions divided by each other using the ‘/’ symbol
- ’*’ and ‘/” bind most tightly; then ‘+’ and ‘-’. Bracketed expressions take the highest priority.

sli.do 4793883

By now, I hope I've convinced you that we need to be a bit more precise.

sli.do 4793883

Regular Expressions

- A **regular expression** is a pattern that can match strings. Regular expressions have many uses:
  - As an argument to ‘grep’ at the command line to search a file system
  - Specifying a pattern to find and replace in a text file
  - E-mail filter rules
- Or, for what we’re most interested in: recognising **tokens** in program source

sli.do 4793883

Regular Expressions: Examples

`recogni(s|z)ing`	Matches the strings ‘recognising’ and ‘recognizing’
`[a-z][a-zA-Z0-9]*`	Matches an identifier that starts with a lowercase letter, followed by any letters or a number
`\d\d\d\d-\d\d-\d\d` `\d\d:\d\d:\d\d(+|-)\d\d`	Matches ISO-8601-like dates, e.g., 1992-04-24 07:00:00+00

sli.do 4793883

Regular Expressions: Definition

a	Matches character a	$R*$	Matches 0 or more copies of R
R_1R_2	Matches concatenation of patterns R1 and R2	(R)	Matches R (brackets used for grouping)
$R_1 \mid R_2$	Matches either R1 or R2	ε	Empty string

sli.do 4793883

Regular Expressions: Derived Forms

$R? \triangleq R|\varepsilon$

Optional occurrence of R

$R+ \triangleq RR*$

One or more occurrences of R

$[abc] \triangleq a|b|c$

Match any of a, b, or c

sli.do 4793883

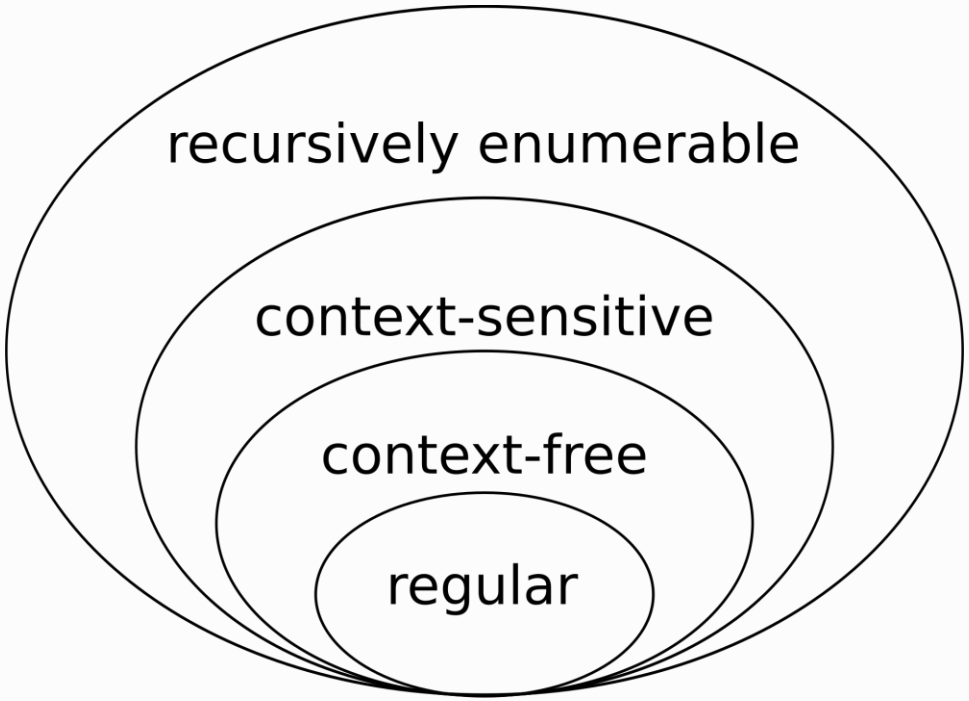
Regular Expressions: Expressive Power

- Regular expressions are useful for basic string matching, but can only match **regular languages**
- Many interesting languages are **not** regular
  - HTML / XML
  - Any programming language with nesting…
- Regular expressions **cannot handle nesting**, so we **cannot** use them to parse most programming languages
  - Nevertheless, they are very useful for **tokenisation**, which converts strings to streams of tokens; Michele will talk more about this later in the course

https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags

sli.do 4793883

Chomsky's Hierarchy of Language



- Regular expressions are **regular languages**
  - They can be recognised by a **finite state automaton**
- Context-free languages (the ones we will focus on) can be recognised by a **push-down automaton**
- Context-sensitive languages and recursively enumerable languages require more interesting recognisers (e.g. a Turing machine)

sli.do 4793883

Grammars

- Regular expressions help, but don't cut it for describing the syntax of a programming language.
- A **grammar** is a set of formal rules specifying how to construct a sentence in a language. It consists of:

A set of **terminal symbols**: symbols that occur in a sentence

A distinguished **sentence symbol** that stands for a complete sentence

A set of **nonterminal symbols**, which each 'stand for' part of a phrase

A set of **production rules** that show how phrases can be made up from terminals and sub-phrases

sli.do 4793883

Example: L_{Arith}

We can now give a formal grammar for the **concrete syntax** of our expression language

Strings occurring without quotes are **nonterminals**, standing for an occurrence of a member of the given symbol

This is a **production**, showing that an expr can either be a `prim` or a term of the form `expr op prim`

Strings in quotes are **terminal symbols**: character or string literals

```
expr ::= prim
       | expr op prim
op   ::= '+' | '-' | '*' | '/'
prim ::= int | '(' expr ')'
int  ::= digit | digit int
digit ::= '1' | '2' | '3' | '4' | '5'
        | '6' | '7' | '8' | '9' | '0'
```

sli.do 4793883

Backus-Naur Form

- Our grammar for L_{Arith} is an example of a grammar in **Backus-Naur Form (BNF)**
- A BNF grammar consists of a series of productions takes the form
  $S ::= \alpha \mid \beta \mid \gamma$
  where α, β, γ each stand for a sequence of terminal and nonterminal symbols
- **Extended** BNF (EBNF) allows us to use regular expression notation in productions (e.g. in `int`)

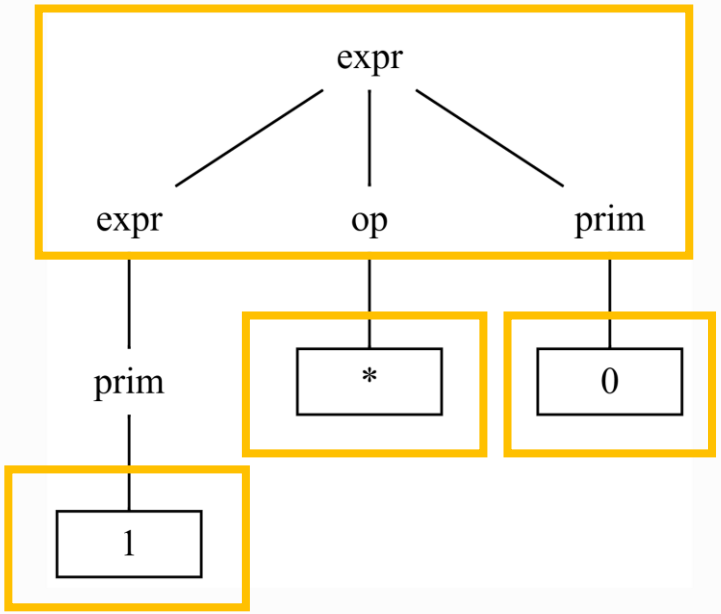
```
expr ::= prim
       | expr op prim
op   ::= '+' | '-' | '*' | '/'
prim ::= int | '(' expr ')'
int  ::= digit+
digit ::= '1' | '2' | '3'
        |  '4' | '5' | '6'
        | '7' | '8' | '9' | '0'
```

sli.do 4793883

```
expr ::= prim
       | expr op prim
op   ::= ‘+’ | ‘-’ | ‘*’ | ‘/’
prim ::= int | ‘(‘ expr ‘)’
int  ::= digit+
digit ::= ‘1’ | ‘2’ | ‘3’
        |  ‘4’ | ‘5’ | ‘6’
        | ‘7’ | ‘8’ | ‘9’ | ‘0’
```

Parse Trees

We can represent how a string corresponds to a grammar *G* using a **parse tree** (or **syntax tree**)



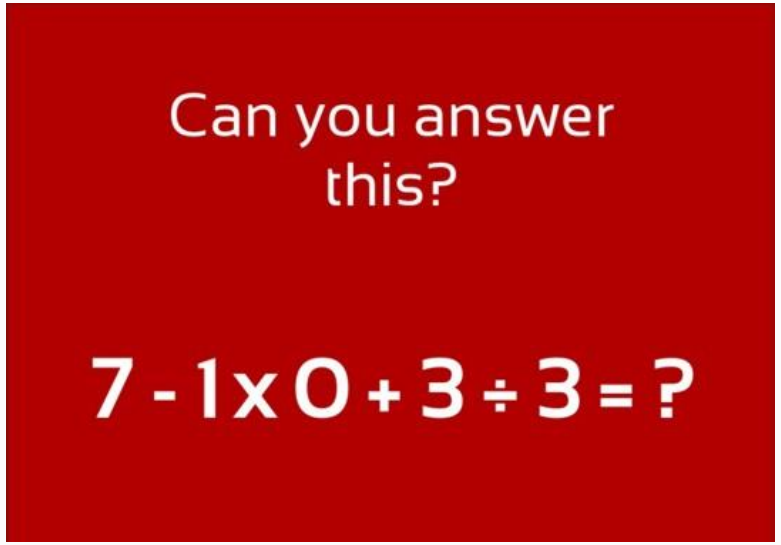
Each **terminal node** is labelled by a terminal symbol (here, digits and operators)

Each **nonterminal node** is labelled by a nonterminal symbol of G and can have children nodes X, Y, Z **only if G has a production rule N ::= … | X Y Z | …**

sli.do 4793883

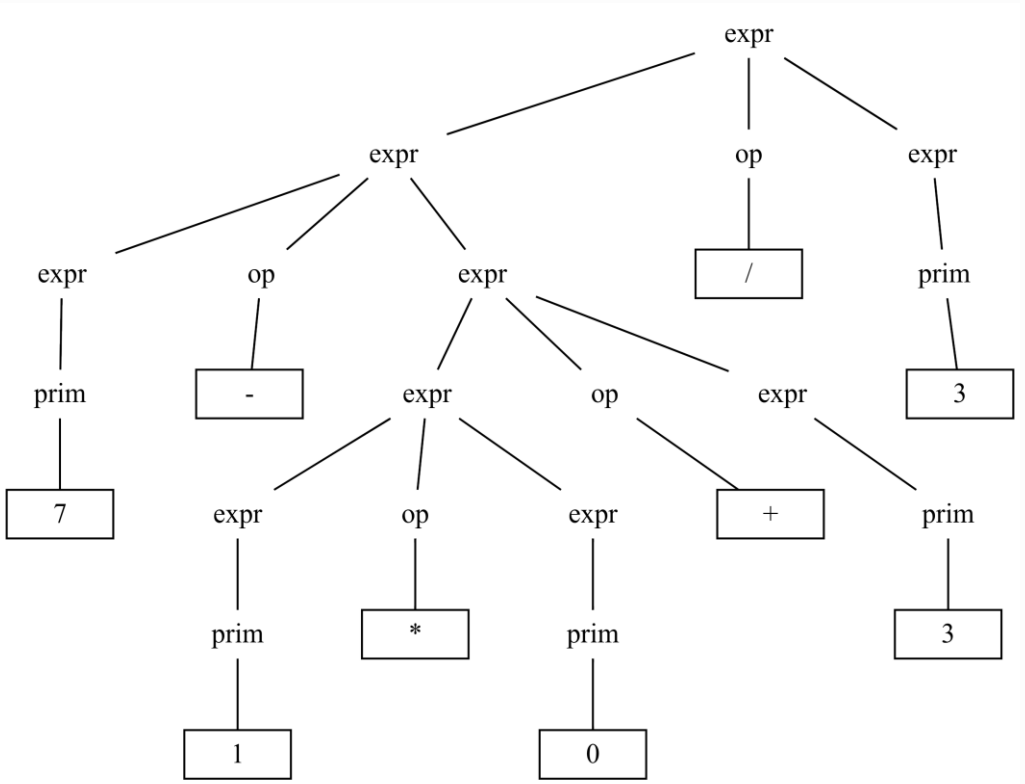
Ambiguity

- A grammar is **ambiguous** if we can derive two or more different parse trees for a string
- If we were to allow arbitrary expressions on either side of an operator, we can show how one of our tedious Facebook puzzles is ambiguous...

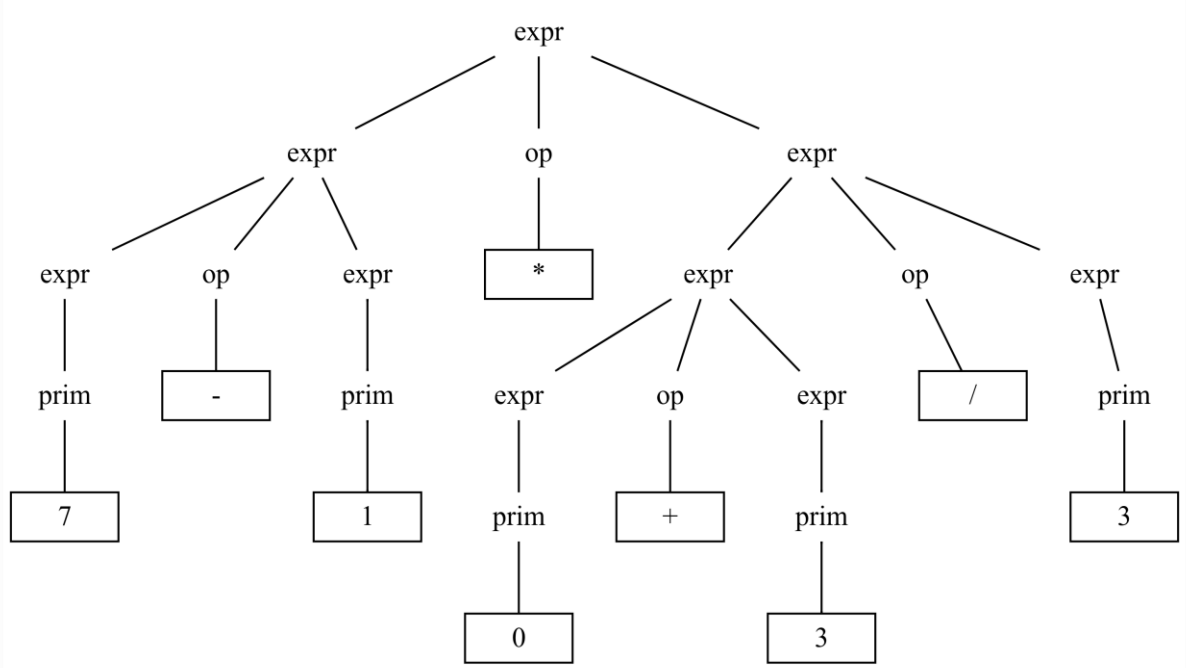


7-1x0+3÷3=?

sli.do 4793883



(7 - ((1 * 0) + 3)) / 3



(7 - 1) * ((0 + 3) / 3)

sli.do 4793883

Ambiguity in Natural Language

“Fruit flies like a banana”

Are bananas liked by fruit flies, or does fruit have wings and fly in the manner of a banana?

“He saw the dog with one eye”

Did the man see the dog using one eye, or did the dog have one eye?

“He ate the ham sandwich with the tomato”

Did the sandwich contain a tomato, or did the man use a tomato as cutlery?

sli.do 4793883

Ambiguity in PLs: The Dangling-If Problem

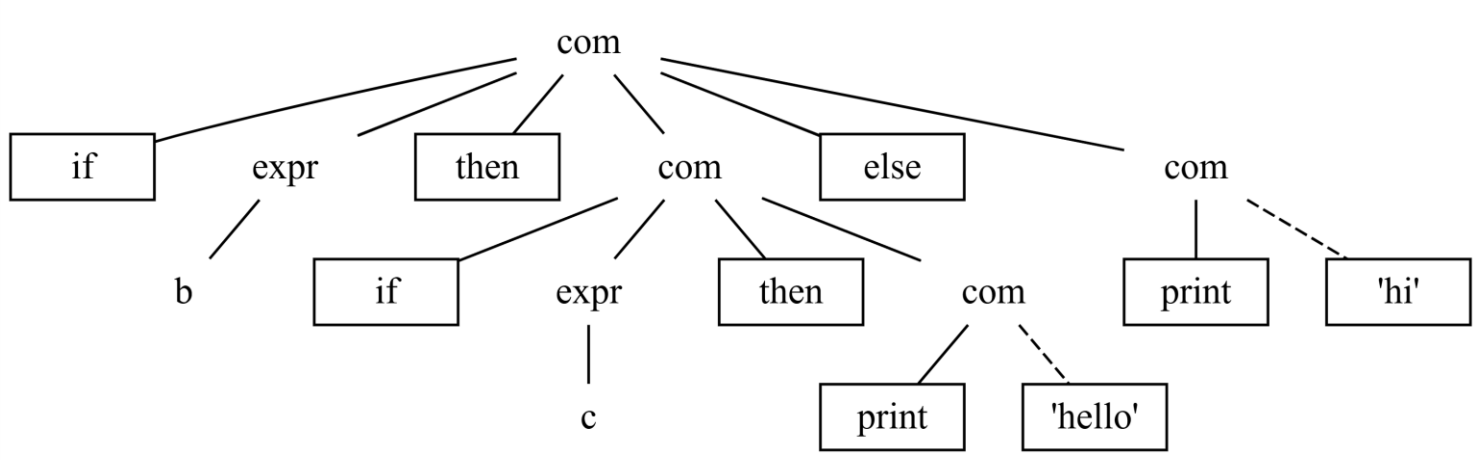
- Ambiguity is OK in natural language (although try and write unambiguously if you can!)
- The syntax of a programming language should be **unambiguous**, though, as otherwise different parses would change the meaning of a program.
- An infamous example is the 'dangling if' problem

```
stmt ::= 'print' expr
       | 'if' expr 'then' stmt
       | 'if' expr 'then' stmt 'else' stmt
```

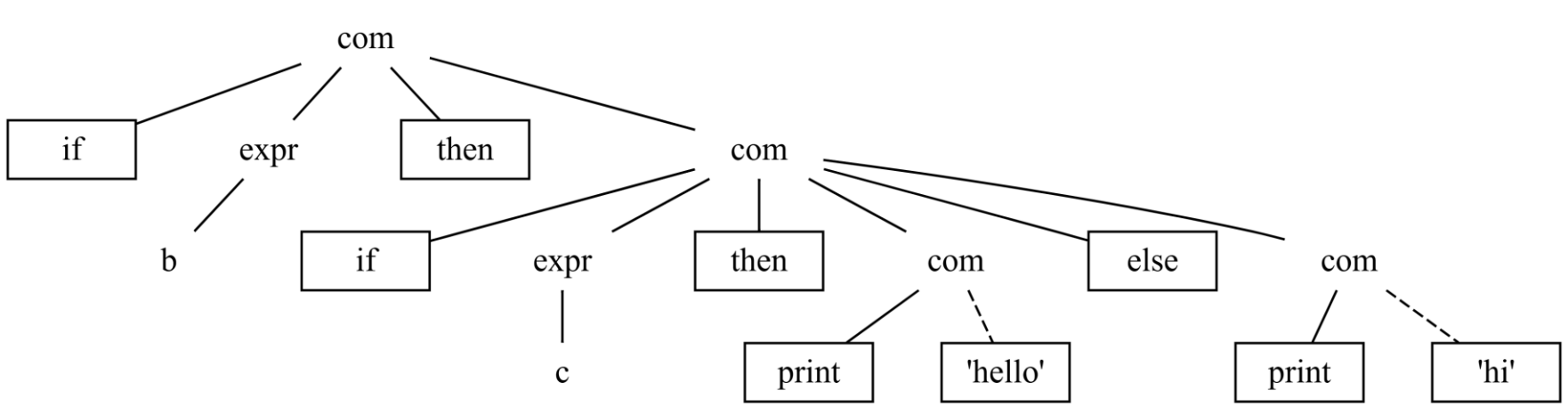
Which 'if' statement does this 'else' belong to?

```
if b then if c then print "hello" else print "hi"
```

```
if b then if c then print “hello” else print “hi”
```




```
if b then if c then print “hello” else print “hi”
```



sli.do 4793883

Concrete vs. Abstract Syntax

The grammar of our language contains **more syntax than is needed for computation**: in particular, brackets are only needed to show how to parse an expression.

```
(1 + 2 * 3) + (3 * 4)
```

Includes irrelevant syntactic information, has implicit information about operator precedence

```
Add(
  Add(Num(1),
    Mul(Num(2), Num(3))),
  Mul(Num(3), Num(4)))
```

Simplified representation of a parsed expression: unambiguous precedence, no irrelevant syntax

sli.do 4793883

Abstract Syntax

- Abstract syntax allows us to **abstract away irrelevant syntactic noise**: we can concentrate on the important parts of the language
  - (i.e., we assume parsing has been done already)
- Apart from the lecture on parsing later in the course, **all examples from now on will use abstract rather than concrete syntax**

```
expr ::= prim
       | expr op prim
op   ::= ‘+’ | ‘-’ | ‘*’ | ‘/’
prim ::= int | ‘(‘ expr ‘)’
int  ::= digit+
digit ::= ‘1’ | ‘2’ | ‘3’
        |  ‘4’ | ‘5’ | ‘6’
        | ‘7’ | ‘8’ | ‘9’ | ‘0’
```

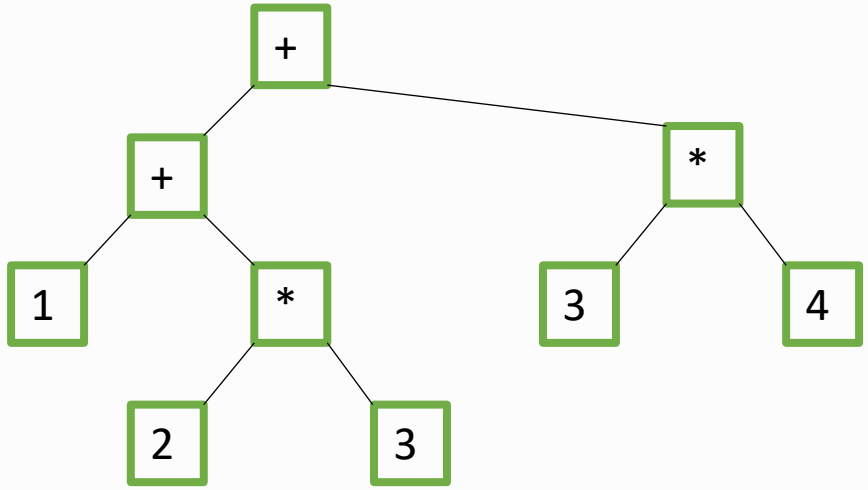
```
Integers n
Operators ⊙  ::= + | - | * | /
Terms L, M, N ::= n
                | L ⊙ M
```

sli.do 4793883

Abstract Syntax Trees (ASTs)

- We can also write abstract syntax as a tree (and indeed, this is how every parsed program is represented).
- Consider our example from earlier, the AST for:
  `(1 + 2 * 3) + (3 * 4)`
- All further operations (desugaring, typechecking, evaluation, compilation) are done on ASTs

```
Add(
  Add(Num(1),
    Mul(Num(2), Num(3))),
  Mul(Num(3), Num(4)))
```



sli.do 4793883

How do we parse text into an AST?

- Generally, we do the following:
  - **Tokenise** text into chunks using regular expressions (lexing)
  - Match **token streams** and convert these into AST nodes (parsing)

- In practice there are various ways of implementing this: for example, recursive-descent parsers, parser generators…
  - We will discuss these in more detail later in the course

sli.do 4793883

Summing Up

- **This lecture:**
  - Concrete and abstract syntax
  - Informal vs. formal specification
  - Regular expressions, Backus-Naur Form grammars
  - Ambiguity
- **Next time**
  - Variables and binding, functions, and substitution
  - …and after that, we'll look at operational semantics
- **Over to you**
  - Please fill in this week's reflection!

Sli.do Questions

sli.do 4793883

Sli.do Questions (1)

- “Why is Python not both Imperative and OO?”
  - Indeed it is multi-paradigm (the lists were non-exhaustive). Many languages are multi-paradigm, nowadays – for example Python takes some inspiration from functional programming (e.g. with list comprehensions, which originated in Haskell I believe)
- Is it possible to use a language like SQL for systems programming. Obviously very difficult and tedious but is it possible?
  - SQL has recursion, so can express looping behaviour. The other part of the puzzle is some way of doing system calls (e.g. through a foreign function interface). If there is (I expect you can do this with PostgreSQL plugins for example), then plausibly you could use it! It wouldn’t be nice though.

sli.do 4793883

Sli.do Questions (2)

- Do void function calls still count as expressions?
  - Great question. In Python, function calls are always expressions (they return None). In functional languages, similarly you have a dummy value called 'unit' (written `()`) that's returned from a side-effecting operation.
  - A void function call in Java, I believe, can't be evaluated in an expression position you can't have a value of type '`void`', nor can you expect a value of type `void` as an argument to a function, so it would have to be a statement.
  - This is different to having a `void*` pointer in C, which is perfectly sensible (and used for passing values of unspecified type). There's also the `Void` (note capital V) type in Java which is quite different; this is a type whose only inhabitant is `null`.