

# OCAML, PART I

We are going to **learn** a new language

What does that mean?

## Five Aspects of Learning a PL

Syntax. Programs are phrases written in an artificial language

Languages are made of symbols that are combined according to **grammatical rules**

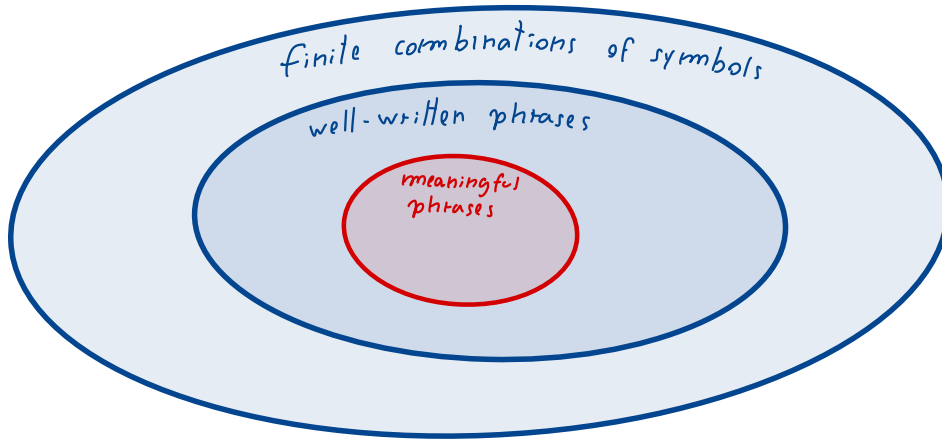
Syntax defines the **grammatically well-written** phrases of a language

Ex.

. let $x = \text{let}$ in $x$	not well-written
. let $x = 5$ in length (5)	well-written but meaningless
. let $x = 5$ in $x + x$	well-written and meaningful

## Semantics

- Among well-written phrases (= programs), what are the **meaningful** ones?



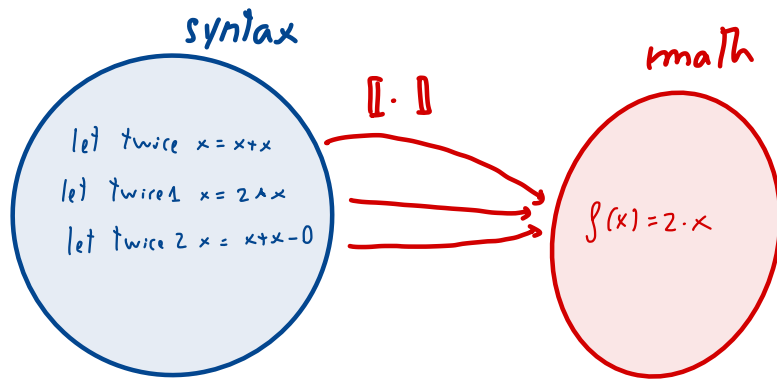
- What is the **meaning** of a program?

- Denotational semantics

Programs = syntax for mathematical objects

"  
denotations

$\llbracket \text{let plus } x = x + x \rrbracket = \text{double function in math}$



- OPERATIONAL SEMANTICS

Programs = syntax + computational content  
 " meaning (dynamic)

Op. semantics explains how computers understand programs

Dynamic Semantics: how programs are evaluated

$$e \rightarrow e' \quad \langle \sigma, c \rangle \rightarrow \langle \sigma', c' \rangle$$

Static Semantics: syntax also carries a meaning

let foo x = x + x

↳ foo is a function that has an input with a notion of addition

→ static semantics usually given via type systems

In This course: ① Syntax

ex.  $e_1, e_2, e_3$  expressions, then  
if-then-else ( $e_1, e_2, e_3$ ) expression

② Statics

$e_1 : \text{bool} \quad e_2 : \tau \quad e_3 : \tau$   
if-then-else ( $e_1, e_2, e_3$ ) :  $\tau$

③ Dynamics

if-then-else (true,  $e_2, e_3$ ) →  $e_2$   
if-then-else (false,  $e_2, e_3$ ) →  $e_3$

$b \rightarrow b'$

if-then-else ( $b, e_2, e_3$ ) → if-then-else ( $b', e_2, e_3$ )

→ All of that tells us what needed to **implement** a language

**syntax** → 
$$\left. \begin{array}{c} \text{if-then-else} \\ \swarrow \quad | \quad \searrow \\ e_1 \quad e_2 \quad e_3 \end{array} \right\} \begin{array}{l} \text{syntax-tree (abstract syntax)} \\ \text{lexer, parser, ...} \end{array}$$

**static semantics** → type-checking  
type-inference

$$\begin{aligned} & \text{ty-infer} (\text{if-then-else} (e_1, e_2, e_3)) \\ &= \text{let } \text{ty}_2 = \text{ty-infer} (e_1) \\ & \quad \text{ty}_2 = \text{ty-infer} (e_2) \\ & \quad \text{ty}_3 = \text{ty-infer} (e_3) \\ & \text{in} \\ & \quad \text{if } \text{ty}_2 == \text{bool} \ \& \\ & \quad \quad \text{ty}_2 == \text{ty}_3 \\ & \quad \text{then } \text{ty}_3 \end{aligned}$$

**dynamic semantics** → interpreter

$$\begin{aligned} & \text{eval} (\text{if-then-else} (e_1, e_2, e_3)) = \\ & \quad \text{let } b = \text{eval} (e_1) \\ & \quad \text{in } \text{if } b == \text{true} \ \text{then } \text{eval} (e_2) \ \text{else } \text{eval} (e_3) \end{aligned}$$

## Idioms

What are the patterns that people fluent in the language use to solve problems?

Ex. Use Java-style expressions in OCaml does not work well, although you can do that

→ learning by doing  $\begin{cases} \text{read good code} \\ \text{look for beautiful code} \\ \text{experience} \end{cases}$

LIBRARIES

TOOLS

} → not covered in this course

Building blocks

Expressions



2. What is a program in FP? (cf. paradigm)

In imperative programming, programs are built out of

instructions  $\left\{ \begin{array}{l} \text{statements} \\ \text{commands} \\ \text{expressions} \\ \vdots \end{array} \right\} \rightarrow \text{many syntactic categories}$

while ( $3+5 == 0$ )  $\rightarrow$  expression  $\rightarrow$  leaves state unchanged; just arithmetic  
 $x := x + 1$   $\rightarrow$  a command  $\rightarrow$  determines a change in the state of the machine

In FP, programs are expressions  $\rightarrow$  no command, no statement, ...  
 $\Rightarrow$  no state (!)

$\leadsto$  Programming as advanced algebra  
Computation as calculation

Any expression has a

syntax

$(e_1 + e_2, \text{if-then-else } (e_1, e_2, e_3), \text{let } x = e_1 \text{ in } e_2, \dots)$

semantics

→ static = meaning of an expression "at least", without when non-executed

→ dynamic = " " when executed

What does it mean to execute an expression?

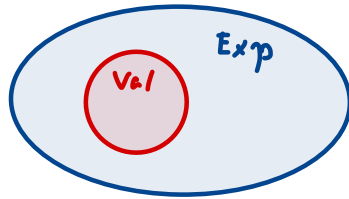
computation = calculation

Ex. High-school

Calculate  $(1+2)^2 =$  Simplify  $(1+2)^2$  to simplest expressions until the simplest is achieved

$(1+2)^2 \rightarrow 3^2$   
 $\rightarrow 3 \wedge 3$   
 $\rightarrow 9$  } computation

**Computation** = reduce an expression until a value is reached, if any  
**Value** = an expression that cannot be further reduced



$e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n \rightarrow v$   
evaluation

$$\text{eval}(e_0) = v$$

OCaml interpreter ( $U_{top}$ ) is a generalised calculator



# OCAML EXPRESSIONS

- integers  $\left\{ \begin{array}{l} \text{values: } 0, 1, 2, -1, -2, \dots \\ \text{exp: } e_1 + e_2, e_1 * e_2, \dots \\ \text{typing: } \text{int} \\ \text{evaluation: } \dots \end{array} \right.$

- booleans  $\left\{ \begin{array}{l} \text{value: } \text{true, false} \\ \text{exp: } \text{if } e_1 \text{ then } e_2 \text{ else } e_3, e_1 \ \&\& \ e_2, \dots \\ \text{typing: } \text{bool} \\ \text{evaluation: } \end{array} \right.$

---

$e_1 \Rightarrow \text{true} \quad e_2 \Rightarrow v$   
 $\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v$

- float  $\left\{ \begin{array}{l} \text{value: } 3.14, \dots \\ \text{exp: } e_1 * e_2, e_1 + e_2, \dots \\ \text{typing: } \text{float} \\ \text{evaluation: } \dots \end{array} \right.$

↳ to evaluate "if  $e_1$  then  $e_2$  else  $e_3$ "

- evaluate  $e_1$  to a value  $b$
- if  $b$  is true, then evaluate  $e_2$  and return the result
- otherwise, evaluate  $e_3$  and return the result

What does OCaml do when we give it an expression  $e$ ?

① Massage syntax of  $e$  :  $3+2 \rightsquigarrow$  

② Type-inference  $\rightsquigarrow$  infer the type of  $e$

• No need for the programmer to write the type (but better if you do that)

• Type inference is done by the **compiler**  
 $\rightsquigarrow$  before program execution ( $\rightsquigarrow$  **static**)

• Find lots of bugs without wasting resources

Java does the same;  
Python infer types  
at dynamic time  $\leftarrow$

Relation between **statics** and **dynamics**

"Well-typed programs do not go wrong" (R. Milner)

$$\frac{\vdash e : \tau \quad e \rightarrow e'}{\vdash e' : \tau}$$

$\vdash e : \tau \implies$  either  $e$  is a **value**  
or  
 $\exists e'. e \rightarrow e'$   
computation can progress

} no error during computation

# DEFINITIONS

Among expressions, we have definitions

let  $x = 42$   
  ↑  
  variable

evaluating  $\text{let } x = 42$  gives

$\text{val } x : \text{int} = 42$

"we have the value 42, of type int, which is bound to the name x"

If we know evaluate  $x$ , we just get 42

Syntax :  $let\ x = e$   
          ↓      ↗ expression  
          identifier  
          (variable)

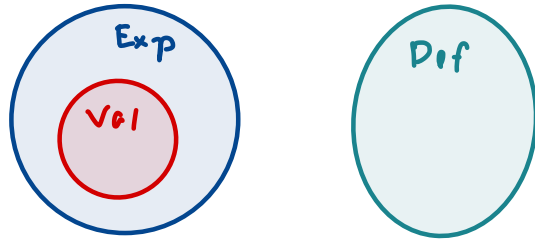
- Dynamics.
- We need first to introduce the notion of an **environment**  
 $\eta$  = stores of variables/identifiers with associated values  
e.g.  $\eta : [x \mapsto 3, y \mapsto 2]$
  - We evaluate expressions within environments

$$\eta \Vdash e \Rightarrow v$$

- To evaluate  $let\ x = e$  in environment  $\eta$  do:
  - evaluate  $e$  in environment  $\eta$ , obtaining value  $v$
  - **bind**  $v$  to  $x$  in  $\eta$ , i.e. build  $\eta[x \mapsto v]$

## Comments

1. Are definitions expressions? Not really
- Definitions do not evaluate to a value  
→ just update the environment
  - Definitions do not have static semantics
  - We cannot use definitions as expressions  $(let\ x = 42) + 3$



3. Are definitions expressions? Yes, actually

Definitions are syntactic sugar for let-in expressions

$let\ x = 42\ in\ \underbrace{3 + x}_{\text{continuation}}$



A definition is a let-in exp. "without" continuation (i.e. with trivial continuation).

Syntax.  $e ::= \dots \mid \text{let } x = e \text{ in } e$

Statics. 
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad \left. \vphantom{\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}} \right\} \text{more of that later}$$

Dynamics 
$$\frac{\eta \Vdash e_1 \Rightarrow v_1 \quad \eta [x \mapsto v_1] \Vdash e_2 \Rightarrow v_2}{\eta \Vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}$$

# VARIABLES AND IMMUTABILITY

Variables in FP are **immutable**: They are name/place holders for values, as in math

let  $x = 42$  ;;  
let  $x = 1$  ;;  $\rightarrow$  error:  $x$  already defined

Immutability makes code much safer

$\rightarrow$  **REFERENTIAL TRANSPARENCY**: an expression and the value it computes are equivalent  
(requires absence of side effects)

$e[e] \cong e[v]$  whenever  $e \Rightarrow v$

counterexample:  $e[e] = e/x$   
 $e = x := 0; 42$

That  $e[e] \Rightarrow$  error

$e[42] \Rightarrow 42/42$

- **EQUATIONAL REASONING**: reason about code using systems of equations
- $\left( \begin{array}{l} \text{let } x = e_1 \\ \text{in } e_2 \end{array} \right) \cong e_2$  if  $x$  not a variable in  $e_1$   
 dead-code optimisation  
 not valid if variables are mutable
  - $e_1 + e_2 \cong e_2 + e_1$   
 not valid if variables are mutable  
 (suppose we start with  $x \mapsto 1$ .  
 $(x := 0; 3) + 1/x \not\cong 1/x + (x := 0; 3)$ )
  - Easy to parallelise code

Immutability sometimes difficult to digest.

in FP "objects" do not change

sort  $([1, 3, 2])$  creates a new list  $[1, 2, 3]$

Suppose we have a structure "person"

Pippo = { age: 99, name: "Pippo" }

want to update the age of Pippo to 100

next\_year(Pippo) **does not change Pippo.**

A new structure is created, exactly like Pippo, but with age 100