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

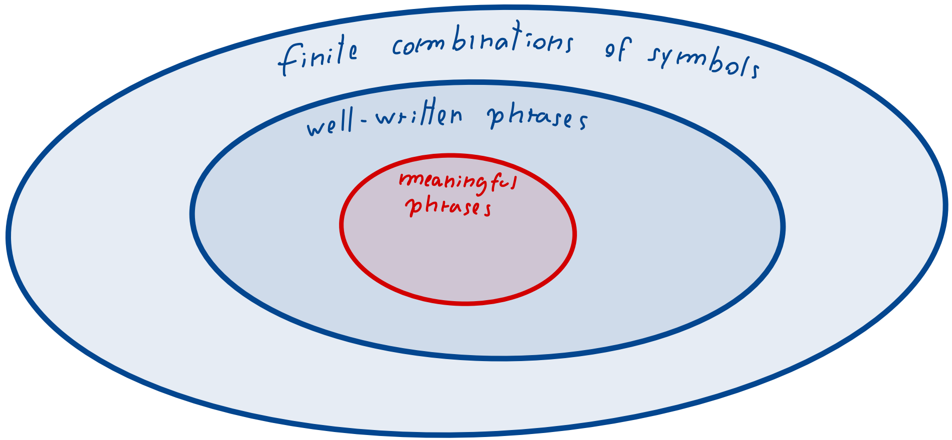
Syntax defines the grammatically well-written phrases of a language

Ex.

· let x = 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?

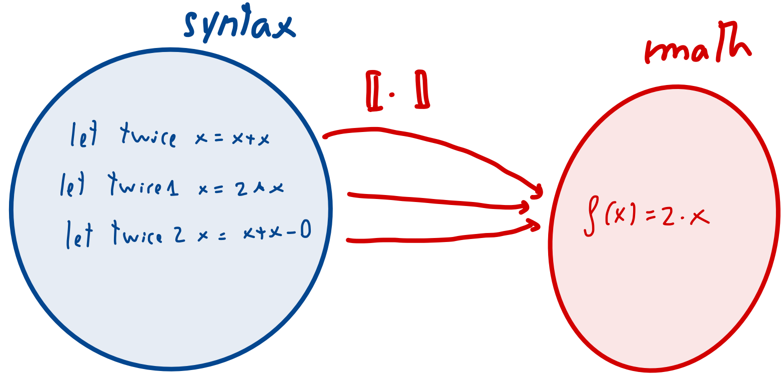


- What is the meaning of a program?
  - **Denotational semantics**

    Programs = syntax for mathematical objects

    = denotations

    $[\![\text{let plus } x = x + x]\!]$ = double function in math



- **OPERATIONAL SEMANTICS**

Programs = syntax + computational content

$\shortparallel$

meaning (dynamic)

Op. semantics explains how computers understand programs

Dynamic Semantics : how programs are evaluated

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

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

$$\frac{e_1 : \text{bool} \qquad e_2 : \mathbb{Z} \qquad e_3 : \mathbb{Z}}{\text{if-then-else}(e_1, e_2, e_3) : \mathbb{Z}}$$

③ Dynamics

$\text{if-then-else}(\text{true}, e_2, e_3) \to e_2$

$\text{if-then-else}(\text{false}, e_2, e_3) \to e_3$

$$\frac{b \to b'}{\text{if-then-else}(b, e_2, e_3) \to \text{if-then-else}(b', e_2, e_3)}$$

⇝ All of that tells us what needed to implement a language

syntax ⇝

$$\begin{array}{ccc} & \text{if-then-else} & \\ \diagup & | & \diagdown \\ e_1 & e_2 & e_3 \end{array}$$

} syntax-tree (abstract syntax)

lexer, parser, ...

static semantics ⇝ type-checking
type-inference

ty-infer (if-then-else (e_1, e_2, e_3))
= let ty_1 = ty-infer (e_1)
ty_2 = ty-infer (e_2)
ty_3 = ty-infer (e_3)
in
if ty_1 == bool &
ty_2 == ty_3
then ty_3

dynamic semantics ⇝ interpreter

eval (if-then-else (e_1, e_2, e_3)) =
let b = eval (e_1)
in if b == true then eval (e_2) else eval (e_3)

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
- read good code
- look for beautiful code
- experience

LIBRARIES

TOOLS

→ not covered in this course

Building blocks

Expressions

2. What is a <u>program</u> in FP? (cf. paradigm)

In imperative programming, programs are built out of

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

while $(\underbrace{3+5==0}_{})$ ↗ expression ⇝ leaves state unchanged; just arithmetic

$\underbrace{x := x+1}_{}$ ↗ a command ⇝ determines a change in the <u>state</u> of the machine

In FP, programs are expressions ⇝ no command, no statement, ...

⟹ no state (?!)

⇝ Programming as advanced <u>algebra</u>
Computation as <u>calculation</u>

Any expression has a

syntax ($e_1 + e_2$, if-then-else (e_1, e_2, e_3), let $x = e_1$ in e_2, ...)

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

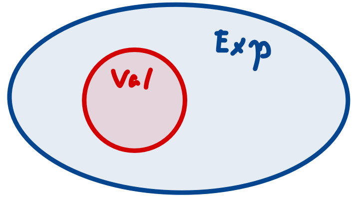
Ex. High-school

Calculate $\underbrace{(1+2)^2}_{\text{expression}}$ = Simplify $(1+2)^2$ to simpler expressions until the $\underbrace{\text{simplest}}_{\text{value}}$ is achieved

$$\left.\begin{aligned} (1+2)^2 &\longrightarrow 3^2 \\ &\longrightarrow 3 * 3 \\ &\longrightarrow 9 \end{aligned}\right\} \text{computation}$$

Computation = reduce an expression until a value is reached, if any

Value = an expression that cannot be further reduced



$$\underbrace{e_0 \to e_1 \to e_2 \to \cdots \to e_n \to v}_{\text{evaluation}}$$

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

OCaml interpreter (Utop) is a **generalised calculator**

$$\xrightarrow{\text{exp } e} \boxed{\text{OCAML}} \xrightarrow{\text{eval}(e)}$$

OCAML EXPRESSIONS

- integers
  - values: $0, 1, 2, -1, -2, \ldots$
  - exp: $e_1 + e_2$, $e_1 * e_2, \ldots$
  - typing: int
  - evaluation: ...

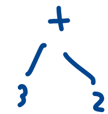
- booleans
  - value: true, false
  - exp: if e_1 then e_2 else e_3, e_1 && $e_2, \ldots$
  - typing: bool
  - evaluation:

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

↳ 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

- float
  - value: 3.14, ...
  - exp e_1 *. e_2, e_1 +. $e_2, \ldots$
  - typing: float
  - evaluation: ...

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

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

② Type-inference $\rightsquigarrow$ <u>infer the type of e</u>

- 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 waisting resources

Java does the same; Python infer types at dynamic time ←

Relation between statics and dynamics

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

$$\frac{\vdash e : \tau \qquad e \to e'}{\vdash e' : \tau}$$

$$\vdash e : \tau \implies \text{either } e \text{ is a value or } \underbrace{\exists e'.\ e \to e'}_{\text{computation can progress}}$$

no error during computation

DEFINITIONS

Among expressions, we have **definitions**

```
let x = 42
```

↑ variable

evaluating `let x = 42` gives

```
val x : 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 $x = e$

x: identifier (variable)

e: expression

Dynamics. · We need first to introduce the notion of an environment

η = stores of variables/identifiers with associated values

e.g. $\eta : [x \mapsto 3, y \mapsto 2]$

· We evaluate expressions within environments

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

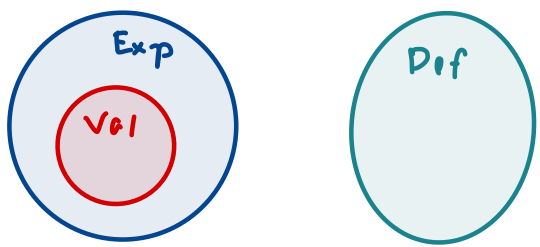
. To evaluate let $x = e$ in environment η do:

- evaluate e in environment η, obtaining value v
- bind v to x in η, i.e. build $\eta[x \mapsto v]$

Comments

1. Are definitions expressions? Not really

   - Definitions do <u>not</u> evaluate to a <u>value</u>
     ⇝ just update the environment
   - Definitions do <u>not</u> have static semantics
   - We cannot use definitions as expressions   (let x = 42) + 3



2. Are definitions expressions? Yes, actually

   Definitions are <u>syntactic sugar</u> for let-in expressions

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

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

<u>Syntax</u>. $e ::= \ldots \mid \mathit{let}\ x = e\ \mathit{in}\ e$

<u>Statics</u>.
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x:\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \tau_2}$$ } more of that later

<u>Dynamics</u>
$$\frac{\eta \Vdash e_1 \Longrightarrow v_1 \qquad \eta[x \mapsto v_1] \Vdash e_2 \Longrightarrow v_2}{\eta \Vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \Longrightarrow v_2}$$

Variables and Immutability

Variables in FP are immutable: They are name/placeholders for values, as in math

```
let x = 42;;
let x = 1;;
```
→ error: x already defined

Immutability makes code much safer

→ Referential Transparency: an expression and the value it computes are equivalent

(requires absence of side effects)

$\mathcal{C}[e] \cong \mathcal{C}[v]$ whenever $e \Rightarrow v$

Counterexample: $\mathcal{C}[e] = e/x$

$e = x := 0; 42$

Then $\mathcal{C}[e] \Rightarrow$ error

$\mathcal{C}[42] \Rightarrow 42/42$

→ **EQUATIONAL REASONING**: reason about code using systems of equations

- $\underbrace{\left(\begin{array}{l} \text{let } x = e_1 \\ \text{in } e_2 \end{array}\right) \cong e_2}_{\text{dead-code optimisation}}$ if x not a variable in e_1

  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 <u>parallelise code</u>

Immutability sometimes difficult to digest.

in FP "objects" do <u>not</u> change

sort ([1, 3, 2]) creates a <u>new</u> 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 <u>not</u> change Pippo.

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

Let Expressions

Sintassi: $e ::= \dots \mid \text{let } \underbrace{x}_{} = e \text{ in } e$

↳ variabile / identificatore

Statica

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

Se e_1 ha tipo τ_1 ed e_2 ha tipo τ_2 assumendo che abbia tipo τ_1, allora let $x = e_1$ in e_2 ha tipo τ_2.

Dinamica

$$\frac{e_1 \Rightarrow v_1 \qquad \overbrace{e_2[v_1/x]}^{\text{sostituzione}} \Rightarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}$$

per valutare let $x = e_1$ in e_2:

- valutare e_1 ad v_1
- sostituire v_1 per x in e_2, ottenendo nuova espressione e_2'
- valutare e_2' ad v_2
- restituire v_2

Ex

let $x = \underline{2+1}$ in $x + 39$

$\longrightarrow$ let $x = 3$ in $x + 39$

$\longrightarrow$ $(x+39)[3/x]$

$\equiv$ $\underline{3+39}$

$\longrightarrow$ 42

NB. let $x = e_1$ in e_2 è effettivamente una espressione:

(let $x = 3$ in $x+x$) + 2 ✓

(let $x = 3$ in $x+x$) + if (let $x =$ true in x) then 0 else 2 ✓

(let $x = 3$) + 2 ✗

Scope.

Lo scope di una variabile determina la porzione di codice dove la variabile /nome è meaningful

```
let x = 42 in
    x + (let y = 2 in
         x + y)
```

scope di y: `x + y`

scope di x: `x + (let y = 2 in x + y)`

Al di fuori dello scope, la variabile è "non dichiarata", e non può quindi essere valutata

Possiamo avere overlapping di scope

```
let x = 5 in
    ( (let x = 6 in x) + x )
```

VARIABLE RENAMING

In matematica, la scelta delle variabili usate nelle definizioni è irrilevante

$$\left.\begin{array}{l} f(x) = x+1 \\ f(y) = y+1 \end{array}\right\} \text{stessa funzione.}$$

Lo stesso accade in OCaml con le let-expression:

$$\left.\begin{array}{l} \text{let } x = e_1 \text{ in } e_2 \\ \text{let } y = e_1 \text{ in } \underbrace{e_2[y/x]}_{\text{renaming}} \end{array}\right\} \text{stessa espressione, purché } y \text{ non sia già usata}$$

Ex. let $x = 3$ in $x+x \cong$ let $y = 3$ in $y+y$

Diciamo che let $x = e_1$ in e_2 lega (bind) x in e_2, e quindi che x è legata in e_2. Diversamente diciamo che x è libera

$$\text{let } \underline{x = 5} \text{ in}$$
$$(\text{let } \underbrace{x = 6 \text{ in } x+x}_{\text{scopo}}) + x$$

La sostituzione $e[v/x]$ agisce solo sulle occorrenze libere di x

let $x=5$ in (let $x=6$ in $x+x$) $+ x$

$\to$ ((let $x=6$ in $x+x$) $+ x$) $[5/x]$

$=$ $\underbrace{(\text{let } x=6 \text{ in } x+x)[5/x]}_{}$ $+$ $\underbrace{x[5/x]}_{}$ → qui x è libera; possiamo sostituire

↳ qui x è legata; <u>non</u> avviene alcuna sostituzione

$$= (\text{let } x = 6 \text{ in } x + x) + 5$$

$$\to (x + x)[6/x] + 5$$

$$\to (6 + 6) + 5$$

$$\to 12 + 5$$

$$\to 17$$

Questo è consistente col principio di (α)-renaming

$$\text{let } x = e_1 \text{ in } e_2 \cong_\alpha \text{let } y = e_1 \text{ in } e_2[y/x]$$

(y fresco)

Infatti:

$$\text{let } x = 5 \text{ in } (\text{let } x = 6 \text{ in } x + x) + x \cong_\alpha \text{let } x = 5 \text{ in } (\text{let } y = 6 \text{ in } y + y) + x$$

Esercizio

Se scriviamo nell'interprete

```
let x = 1 ;;
let x = 2 ;;
```

$\xrightarrow{\text{valuta}}$ x = 2 : int

immutabilità ?!

zucchero sintattico

let x = 1 in → alloca spazio in memoria, chiamato x, in cui salva 1

let x = 2 in → alloca altro spazio in memoria, sempre col nome x, in cui salva 2

x

→ quale spazio di memoria vado a vedere, visto che ce ne sono 2 col nome x?

⇝ Quello definito dal binder sintatticamente più vicino

$\boxed{\text{scope statico}}$