Five aspects of learning a PL

1. **Syntax**: How do you write language constructs?
2. **Semantics**: What do programs mean? (Type checking, evaluation rules)
3. **Idioms**: What are typical patterns for using language features to express your computation?
4. **Libraries**: What facilities does the language (or a well-known project) provide "standard"? (E.g., file access, data structures)
5. **Tools**: What do language implementations provide to make your job easier? (E.g., top-level, debugger, GUI editor, …)

- All are essential for good programmers to understand
- Breaking a new PL down into these pieces makes it easier to learn

Expressions

Expressions (aka *terms*):

- primary building block of OCaml programs
- akin to *statements* or *commands* in imperative languages
- can get arbitrarily large since any expression can contain subexpressions, etc.

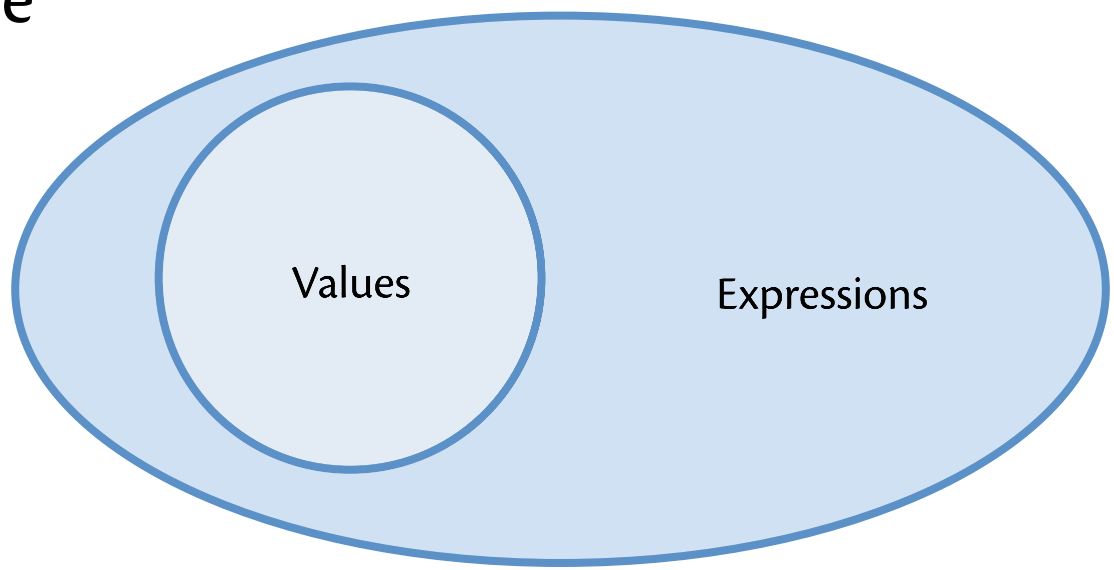
Every kind of expression has:

- **Syntax**
- **Semantics:**
  - **Type-checking rules:** produce a type or fail with an error message
  - **Evaluation rules:** produce a *value*
    - (or exception or infinite loop)
    - Used only on expressions that type-check

Values

A **value** is an expression that does not need any further evaluation

- **`34`** is a value of type **`int`**
- **`34+17`** is an expression of type **`int`** but is not a value



Let expressions

Syntax:

```
let x = e1 in e2
```

`x` is an *identifier*

`e1` and `e2` are *expressions*

`let x = e1 in e2` is itself an expression

`x = e1` is a *binding*

e.g.

```
let x = 2 in x+x
let inc x = x+1 in inc 10
let y = "zar" in (let z = "doz" in y^z)
```

Let expressions

```
let x = e1 in e2
```

Evaluation:

- Evaluate `e1` to a value `v1`
- Substitute `v1` for `x` in `e2`, yielding a new expression `e2'`
- Evaluate `e2'` to `v`
- Result of evaluation is `v`

Let expressions

```
   let x = 1+4 in x*3
-->
```
Evaluate **`e1`** to a value **`v1`**
```
   let x = 5 in x*3
-->
```
Substitute **`v1`** for **`x`** in **`e2`**, yielding a new expression **`e2'`**
```
   5*3
-->
```
Evaluate **`e2'`** to **`v`**
```
   15
```
Result of evaluation is **`v`**

Let expressions in REPL

Syntax:

```
let x = e
```

Implicitly, "`in` *rest of what you type*"

E.g., you type:

```
let a="zar";;
let b="doz";;
let c=a^b;;
```

OCaml understands as

```
let a="zar" in
  let b="doz" in
    let c=a^b in...
```

Scope

Bindings are in effect only in the *scope* (the “block”) in which they occur.

```
let x=42 in
  (* y is not in scope here *)
  x + (let y="3110" in
         (* y is in scope here *)
         int_of_string y)
```

Exactly what you’re used to from (e.g.) Java

Overlapping scope

Overlapping bindings of the same name is usually bad **idiom** (and darn confusing)

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

To what value does the above expression evaluate?

- 10
- 11
- 12
- None of the above

Substitution

```
  let x = 5 in ((let x = 6 in x) + x)
-->
  ???
```

Not a choice:

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

Two choices:

```
A.  ((let x = 6 in x) + 5)
B.  ((let x = 6 in 5) + 5)
```

Substitution

```
  let x = 5 in ((let x = 6 in x) + x)
-->
  ???
```

Not a choice:

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

Two choices:

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

B. ~~((let x = 6 in 5) + 5)~~

Why?

Principle of Name Irrelevance

The name of a variable should not matter.

In math, these are the same functions:

$f(x) = x^2$

$f(y) = y^2$

So in programming, these should be the same functions:

```
let f x = x*x
let f y = y*y
```

This principle is also called *alpha equivalence*

Principle of Name Irrelevance

Likewise, these should be the same expressions:

```
(let x = 6 in x)
(let y = 6 in y)
```

So these should also be the same:

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

But if we substitute inside inner **`let`** expression, they will not be the same:

```
(let x = 6 in 5) + 5   ----> 10
(let y = 6 in y) + 5   ----> 11
```

Back to substitution

```
  let x = 5 in ((let x = 6 in x) + x)
-->
  ???
```

Not a choice:

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

Two choices:

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

B. ~~((let x = 6 in 5) + 5)~~

That's why!

Shadowing

A new binding *shadows* an older binding of the same name

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

Shadowing is not assignment

```
let x = 5 in ((let x = 6 in x) + x)
----> 11

let x = 5 in (x + (let x = 6 in x))
----> 11
```

Types

Write colon to indicate type of expression

As does the top-level:

```
# let x = 42;;
val x : int = 42
```

Type-checking of let expression:

If `e1:t1`,

and if `e2:t2` (assuming that `x:t1`),

then `(let x = e1 in e2) : t2`

Let expressions (summary)

- **Syntax:**
  `let x = e1 in e2`
- **Type-checking:**
  If `e1:t1`, and if `e2:t2` under the assumption that `x:t1`, then `let x = e1 in e2 : t2`
- **Evaluation:**
  - Evaluate `e1` to `v1`
  - Substitute `v1` for `x` in `e2` yielding new expression `e2'`
  - Evaluate `e2'` to `v`
  - Result of evaluation is `v`

Function declaration

Functions:

- Like Java methods, have arguments and result
- Unlike Java, no classes, **`this`**, **`return`**, etc.

Example *function declaration*:

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow x y =
  if y=0 then 1
  else x * pow x (y-1)
```

Note: "**`rec`**" is required because the body includes a recursive *function call*:
`pow(x,y-1)`

Function declaration

- **Syntax:**
  `let f x1 x2 ... xn = e`
- **Evaluation:**
  - No evaluation!
  - Just declaring the function
  - Will be evaluated when applied to arguments
- **Type-checking:**
  - Conclude that `f : t1 -> ... -> tn -> t` if `e : t` under assumptions:
    - `x1:t1, ..., xn:tn` (arguments with their types)
    - `f: t1 -> ... -> tn -> t` (for recursion)

Writing argument types

Though types can be inferred, you can write them too:

```
let rec pow (x : int) (y : int) : int =
  if y=0 then 1
  else x * pow x (y-1)

let rec pow x y =
  if y=0 then 1
  else x * pow x (y-1)

let cube x = pow x 3
let cube (x : int) : int = pow x 3
```

Function application

Syntax: `e0 e1 ... en`

- Parentheses not strictly required around argument(s)
- If there is exactly one argument and you do use parentheses and you leave out the space, syntax looks like C function call: `e0(e1)`

Function application

Type-checking

if `e0 : t1 -> ... -> tn -> t`

and `e1 : t1, ..., en : tn`

then `e0 e1 ... en : t`

e.g., `pow 2 3 : int`

Function application

Evaluation of `e0 e1 ... en`

1. Evaluate **`e0`** to a function
   `let f x1 ... xn = e`
2. Evaluate arguments **`e1...en`** to values **`v1...vn`**
3. Substitute **`vi`** for **`xi`** in **`e`** yielding new expression **`e'`**
4. Evaluate **`e'`** to a value **`v`**, which is result

Anonymous functions

Something that is *anonymous* has no name.

- `42` is an anonymous `int`
- and we can bind it to a name:
  ```
  let x = 42
  ```

- `(fun x -> x+1)` is an anonymous function
- and we can bind it to a name:
  ```
  let inc = fun x -> x+1
  ```

Anonymous functions

Syntax: `(fun x1 ... xn -> e)`

Evaluation:

- A function is already a value: no further computation to do
- In particular, body `e` is not evaluated until function is applied

Type checking:

`(fun x1 ... xn -> e) : t1->...->tn->t`

if `e:t` under assumptions `x1:t1, ..., xn:tn`

Anonymous functions

These two declarations are **syntactically different** but **semantically equivalent**:

```
let inc = fun x -> x+1
let inc x = x+1
```

Anonymous functions

These two expressions are **syntactically different** but **semantically equivalent**:

```
let x = 7 in x+1
(fun x -> x+1) 7
```

Functions are values

- Can use them **anywhere** we use values
- Functions can **take** functions as arguments
- Functions can **return** functions as results

  ...so functions are *higher-order*
- This is not a new language feature; just a consequence of "functions are values"
- But it is a feature with massive consequences

"A language that doesn't affect the way you think about programming is not worth knowing." *--*Alan Perlis

Alan Jay Perlis

1922-1990

First Winner of Turing Award (1966)

for his influence in the area of advanced programming techniques and compiler construction

Google "perlisisms" for great quotes about programming

Higher-order functions

```
(* some base function *)
let double x = 2*x
let square x = x*x

(* apply those functions twice *)
let quad x = double (double x)
let fourth x = square (square x)
```

Higher-order functions

```
(* higher order function that
 * applies f twice to x *)
let twice f x = f (f x)
```

`val twice : ('a -> 'a) -> 'a -> 'a`

`'a` is a *type variable*: could be any type

Higher-order functions

```
(* higher-order function that
 * applies f twice to x *)
let twice f x = f (f x)

(* define functions using twice *)
let quad x = twice double x
let fourth x = twice square x

(* even better definitions *)
let quad = twice double
let fourth = twice square
```