Expressions

CS3100 Fall 2019

Recap

Last Time:

- Why functional programming matters?

Today:

- Expressions, Values, Definitions.

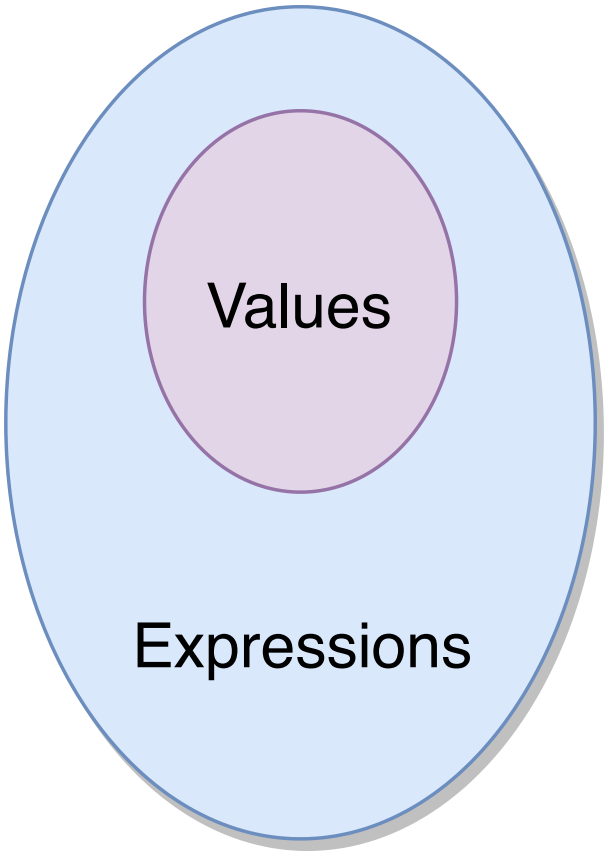
Expressions

Every kind of expression has:

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

Values

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



Values in OCaml

In [1]:

```
42
```

Out[1]:

```
- : int = 42
```

In [2]:

```
"Hello"
```

Out[2]:

```
- : string = "Hello"
```

In [3]:

```
3.1415
```

Out[3]:

```
- : float = 3.1415
```

- Observe that the values have
  - static semantics: types `int`, `string`, `float`.
  - dynamic semantics: the value itself.

Type Inference and annotation

- OCaml compiler **infers** types
  - Compilation fails with type error if it can't
  - Hard part of language design: guaranteeing compiler can infer types when program is correctly written
- You can manually annotate types anywhere – Replace `e` with `(e : t)`
  - Useful for resolving type errors

In [4]:

```
(42.4 : float)
```

Out[4]:

```
- : float = 42.4
```

More values

OCaml also support other values. See manual (https://caml.inria.fr/pub/docs/manual-ocaml/values.html).

In [5]:

```
()
```

Out[5]:

```
- : unit = ()
```

In [6]:

```
(1,"hello", true, 3.4)
```

Out[6]:

```
- : int * string * bool * float = (1, "hello", true, 3.4)
```

In [7]:

```
[1;2;3]
```

Out[7]:

```
- : int list = [1; 2; 3]
```

In [8]:

```
[|1;2;3|]
```

Out[8]:

```
- : int array = [|1; 2; 3|]
```

Static vs Dynamic distinction

Static typing helps catch lots errors at compile time.

Which of these is static error?

In [9]:

```
23 = 45.0
```

```
File "[9]", line 1, characters 5-9:
Error: This expression has type float but an expression was
expected of type
         int
   1: 23 = 45.0
```

In [10]:

```
23 = 45
```

Out[10]:

```
- : bool = false
```

If expression

```
if e1 then e2 else e3
```

- **Static Semantics:** If `e1` has type `bool` , and `e2` has type `t2` and `e3` has type `t2` then `if e1 then e2 else e3` has type `t2` .
- **Dynamic Semantics:** If `e1` evaluates to true, then evaluate `e2` , else evaluate `e3`

In [11]:

```
if 32 = 31 then "Hello" else "World"
```

Out[11]:

```
- : string = "World"
```

In [12]:

```
if true then 13 else 13.4
```

```
File "[12]", line 1, characters 21-25:
Error: This expression has type float but an expression was
expected of type
         int
   1: if true then 13 else 13.4
```

More Formally

Static Semantics of if expression

$$\frac{e1 : bool \quad e2 : t \quad e3 : t}{\text{if } e1 \text{ then } e2 \text{ else } e3 : t}$$

(omits some details which we will cover in later lectures)

to be read as

$$\frac{Premise_1 \quad Premise_2 \quad \ldots \quad Premise_N}{Conclusion}$$

Such rules are known as inference rules.

Dynamic semantics of if expression

For the case when the predicate evaluates to `true` :

$$\frac{e1 \rightarrow \mathit{true} \quad e2 \rightarrow v}{\text{if } e1 \text{ then } e2 \text{ else } e3 \rightarrow v}$$

For the case when the predicate evaluates to `false` :

$$\frac{e1 \rightarrow \mathit{false} \quad e3 \rightarrow v}{\text{if } e1 \text{ then } e2 \text{ else } e3 \rightarrow v}$$

Read $\rightarrow$ as *evaluates to*.

Let expression

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

- `x` is an identifier
- `e1` is the binding expression
- `e2` is the body expression
- `let x = e1 in e2` is itself an expression

In [13]:

```
let x = 5 in x + 5
```

Out[13]:

```
- : int = 10
```

In [14]:

```
let x = 5 in
let y = 10 in
x + y
```

Out[14]:

```
- : int = 15
```

In [15]:

```
let x = 5 in
let x = 10 in
x
```

Out[15]:

```
- : int = 10
```

Scopes & shadowing

```
let x = 5 in
let x = 10 in
x
```

is parsed as

```
let x = 5 in
(let x = 10 in
 x)
```

- Importantly, `x` is not mutated; there are two `x`s in different **scopes**.
- Inner definitions **shadow** the outer definitions.

In [16]:

```
let x = 5 in
let y =
  let x = 10 in
  x
in
x+y
```

```
File "[15]", line 1, characters 4-5:
Warning 26: unused variable x.
```

Out[16]:

```
- : int = 15
```

let at the top-level

```
let x = e
```

is implicitly, "**in** the rest of the program text"

In [17]:

```
let a = "Hello"
```

Out[17]:

```
val a : string = "Hello"
```

In [18]:

```
let b = "World"
```

Out[18]:

```
val b : string = "World"
```

In [19]:

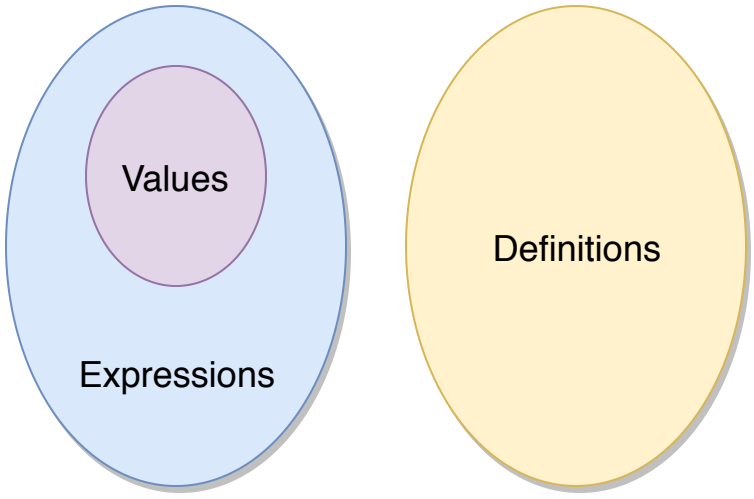
```
let c = a ^ b
```

Out[19]:

```
val c : string = "HelloWorld"
```

Definitions

- The top-level `let x = e` are known as **definitions**.
- Definitions give name to a value.
- Definitions are not expressions, or vice versa.
- But definitions syntactically contain expressions.



Let expression

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

Static semantics

$$\frac{x : t1 \quad e1 : t1 \quad e2 : t2}{\text{let } x = e1 \text{ in } e2 : t2}$$

(again omits some details)

Dynamic semantics

$$\frac{e1 \to v1 \quad \text{substituting } v1 \text{ for } x \text{ in } e2 \to v2}{\text{let } x = e1 \text{ in } e2 \to v2}$$

Exercise

In OCaml, we cannot use `+` for floating point addition, and instead have to use `+.` . Why do you think this is the case?

In [20]:

```
5.4 +. 6.0
```

Out[20]:

```
- : float = 11.4
```

Exercise

Write down the static semantics for `+` and `+.` .

Fin.