

CS 314: Principles of Programming Languages

OCaml Expressions and Functions

Lecture Presentation Style

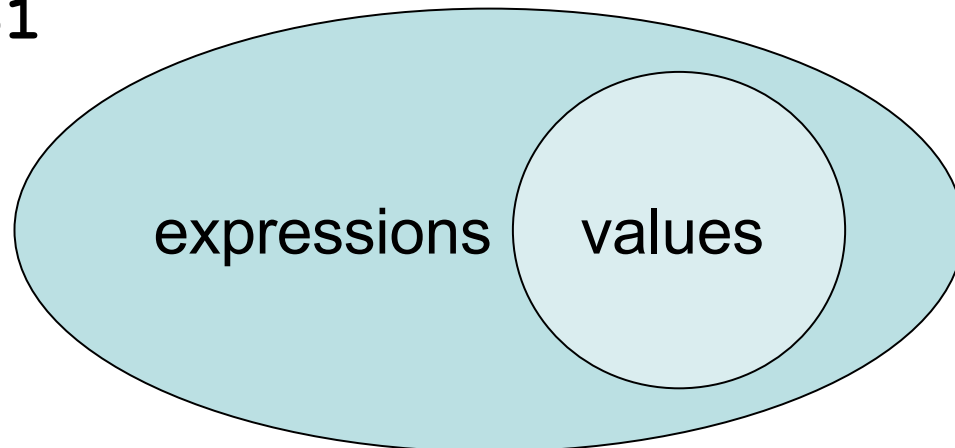
- Our focus: **semantics** and **idioms** for OCaml
 - *Semantics* is what the language does
 - *Idioms* are ways to use the language well
- We will also cover some useful **libraries**
- **Syntax** is what you type, not what you mean
 - In one lang: Different syntax for similar concepts
 - Across langs: Same syntax for different concepts
 - Syntax can be a source of fierce disagreement among language designers!

Expressions

- **Expressions** are our primary building block
 - Akin to *statements* in imperative languages
- Every kind of expression has
 - **Syntax**
 - We use metavariable ***e*** to designate an arbitrary expression
 - **Semantics**
 - **Type checking** rules (static semantics): produce a type or fail with an error message
 - **Evaluation** rules (dynamic semantics): produce a value
 - (or an exception or infinite loop)
 - Used *only* on expressions that type-check

Values

- A **value** is an expression that is final
 - **Evaluating** an expression means running it until it becomes a value
 - We use metavariable **v** to designate an arbitrary value
- **34** is a value, **true** is a value
- **34+17** is an *expression*, but *not* a value
 - It *evaluates* to **51**



Types

- **Types** classify expressions
 - The set of values an expression could evaluate to
 - We use metavariable t to designate an arbitrary type
 - Examples include `int`, `bool`, `string`, and more.
- Expression e has type t if e will (always) evaluate to a value of type t
 - $\{ \dots, -1, 0, 1, \dots \}$ are values of type `int`
 - `34+17` is an expression of type `int`, since it evaluates to `51`, which has type `int`
 - Write $e : t$ to say e has type t
 - Determining that e has type t is called **type checking** (or simply, **typing**)

If Expressions

- Syntax

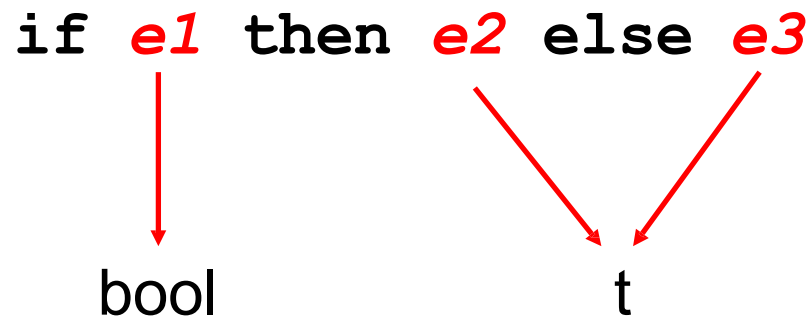
- **if** *e1* **then** *e2* **else** *e3*

- Evaluation

- If *e1* evaluates to **true**, and if *e2* evaluates to *v*, then **if** *e1* **then** *e2* **else** *e3* evaluates to *v*
 - If *e1* evaluates to **false**, and if *e3* evaluates to *v*, then **if** *e1* **then** *e2* **else** *e3* evaluates to *v*

If Expressions

- Type checking
 - If $e1$ has type `bool` and $e2$ has type t and $e3$ has type t then `if $e1$ then $e2$ else $e3$` has type t
 - If $e1 : \text{bool}$ and $e2 : t$ and $e3 : t$ then $(\text{if } e1 \text{ then } e2 \text{ else } e3) : t$



If Expressions: Examples

```
# if 7 > 42 then "hello" else "goodbye";;
```

```
- : string = "goodbye"
```

```
# if true then 3 else 4;;
```

```
- : int = 3
```

```
# if false then 3 else 3.0;;
```

```
Error: This expression has type float but  
an expression was expected of type  
int
```


Quiz 1

To what value does this expression evaluate?

```
if 22<0 then 2 else 1
```

A. 2

B. 1

C. 0

D. none of the above

Quiz 1

To what value does this expression evaluate?

```
if 22<0 then 2 else 1
```

A. 2

B. 1

C. 0

D. none of the above

Quiz 2

To what value does this expression evaluate?

```
if 22<0 then "bear" else 2
```

- A. 2
- B. 1
- C. 0
- D. none of the above

Quiz 2

To what value does this expression evaluate?

```
if 22<0 then "bear" else 2
```

A. 2

B. 1

C. 0

D. none of the above: doesn't type check so never gets a chance to be evaluated

Function Definitions

- OCaml functions are like mathematical functions
 - Compute a result from provided arguments

```
(* requires n>=0 *)  
(* returns: n! *)  
let rec fact n =  
  if n = 0 then  
    1  
  else  
    n * fact (n-1)
```

function body

Use (* *) for comments
(may nest)

Parameter
(type inferred)

rec needed for recursion
(else fact not in scope)

Structural equality

Line breaks, spacing ignored
(like C, C++, Java, not like Ruby)

Type Inference

- As we just saw, a declared variable need not be annotated with its type
 - The type can be **inferred**

```
(* requires n>=0 *)
(* returns: n! *)
let rec fact n =
  if n = 0 then
    1
  else
    n * fact (n-1)
```

n's type is **int**. Why?

= is an infix function that takes two **ints** and returns a **bool**; so n must be an **int** for n = 0 to type check

- **Type inference** happens as a *part of type checking*
 - Determines a type that satisfies code's constraints

Function Types

- In OCaml, `->` is the function type constructor
 - Type `t1 -> t` is a function with argument or *domain* type `t1` and return or *range* type `t`
 - Type `t1 -> t2 -> t` is a function that takes *two* inputs, of types `t1` and `t2`, and returns a value of type `t`. Etc.
- Examples
 - `let next x = x + 1 (* type int -> int *)`
 - `let fn x = (int_of_float x) * 3`
`(* type float -> int *)`
 - `fact`
`(* type int -> int *)`

Function Types

Considering inference

- `+` has type `int -> int -> int`.
 - Therefore, `x + 1` forces `x` to be an `int`.
- `int_of_float` has type `float -> int`.
 - Therefore `(int_of_float x)` forces `x` to be a `float`

• Examples

- `let next x = x + 1 (* type int -> int *)`
- `let fn x = (int_of_float x) * 3`
`(* type float -> int *)`
- `fact` `(* type int -> int *)`

Type Checking Functions

- Syntax `let rec f x1 ... xn = e`
- Type checking
 - Conclude that $f : t1 \rightarrow \dots \rightarrow tn \rightarrow u$ if $e : u$ under the following assumptions:
 - $x1 : t1, \dots, xn : tn$ (arguments with their types)
 - $f : t1 \rightarrow \dots \rightarrow tn \rightarrow u$ (for recursion)
- Example
 - Given $n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int}$
 - Does `if n = 0 then 1 ... : int` ?
 - It does!
 - Conclude `fact : int -> int`

```
let rec fact n =  
  if n = 0 then  
    1  
  else  
    n * fact (n-1)
```

Calling Functions

- Syntax $f\ e1\ \dots\ en$
 - Parentheses not required around argument(s)
 - No commas; use spaces instead
- Type checking
 - If $f : t1 \rightarrow \dots \rightarrow tn \rightarrow u$ and $e1 : t1, \dots, en : tn$ then $f\ e1\ \dots\ en : u$
- Example:
 - `fact 1 : int`
 - since `fact : int -> int` and `1 : int`
- Function call *aka* function application

Calling Functions

- Syntax $f\ e_1 \dots e_n$
- Evaluation
 - Evaluate arguments $e_1 \dots e_n$ to values $v_1 \dots v_n$
 - Order is actually right to left, not left to right
 - But this doesn't matter if $e_1 \dots e_n$ don't have side effects
 - Find the definition of f
 - `let rec f x1 ... xn = e`
 - Substitute v_i for x_i in e , yielding new expression e'
 - Evaluate e' to value v , which is the final result

Calling Functions

Example evaluation

- `fact 2`
 - `if 2=0 then 1 else 2*fact(2-1)`
 - `2 * fact 1`
 - `2 * (if 1=0 then 1 else 1*fact(1-1))`
 - `2 * 1 * fact 0`
 - `2 * 1 * (if 0=0 then 1 else 0*fact(0-1))`
 - `2 * 1 * 1`
 - `2`

```
let rec fact n =  
  if n = 0 then  
    1  
  else  
    n * fact (n-1)
```

Type Annotations

- The syntax `(e : t)` asserts that “`e` has type `t`”
 - This can be added (almost) anywhere you like

```
let (x : int) = 3
let z = (x : int) + 5
```
- Define functions’ parameter and return types

```
let fn (x:int):float =
    (float_of_int x) *. 3.14
```

 - Note special position for return type
 - Thus `let g x:int = ...` means `g` returns `int`
 - *Not* that `x` has type `int`
- Checked by compiler: Very useful for debugging

Quiz 3: What is the type of `foo 4 2`

```
let rec foo n m =  
  if n >= 9 || n < 0 then  
    m  
  else  
    n + m + 1
```

- a) Type Error
- b) `int`
- c) `float`
- d) `int -> int -> int`

Quiz 3: What is the type of `foo 4 2`

```
let rec foo n m =  
  if n >= 9 || n < 0 then  
    m  
  else  
    n + m + 1
```

- a) Type Error
- b) int**
- c) float
- d) `int -> int -> int`

Quiz 4: What is the value of **bar 4**

```
let rec bar(n:int):int =  
  if n = 0 || n = 1 then 1  
  else  
    bar (n-1) + bar (n-2)
```

- a) Syntax Error
- b) 4
- c) 5
- d) 8

Quiz 4: What is the value of **bar** 4

```
let rec bar(n:int):int =  
  if n = 0 || n = 1 then 1  
  else  
    bar (n-1) + bar (n-2)
```

- a) Syntax Error
- b) 4
- c) 5**
- d) 8