CMSC 330: Organization of Programming Languages

Let Expressions, Tuples, Records

Spring 2026

Announcements 02/10/2026

- Quiz 1 is on Friday (02/13)
  - Coding and debugging
  - Review exercise one
  - More Ocaml exercises (https://cmsc330.github.io/exercises.html)
  - Coding and debugging questions in old exams (class resources page)
- Today
  - Let expressions
  - Tuples
  - Anonymous Functions
  - Records

Let Expressions

- Syntax
  - `let x = e1 in e2`
  - `x` is a *bound variable*
  - `e1` is the *binding expression*
  - `e2` is the *body expression*

- `let` expressions bind *local* variables
  - Different from `let` *definitions*, which are at the top-level

Let Expressions

- Syntax
  - `let x = e1 in e2`

- Evaluation
  - e1 ⇒ v1
  - e2{v1/x}

$$\frac{e_1 \Rightarrow v_1 \qquad e_2[v1/x] \Rightarrow v_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow v_2}$$

```
let z = 3+4 in 3*z

21
```

Let Expressions

- Syntax
  - `let x = e1 in e2`

- Type checking

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2} \quad \text{(T-Let)}$$

  - If `e1` : `t1` and
  - If assuming `x` : `t1` implies `e2` : `t`
  - Then (`let x = e1 in e2`) : `t`

Let Expressions

- Syntax
  - `let x = e1 in e2`
- Example: What is the type of `let z = 3+4 in 3*z`?
  - `3+4` : `int`
  - Assuming `z` : `int`, we have `3*z` : `int`
  - So the type of `let z = 3+4 in 3*z` is `int`

Let Definitions vs. Let Expressions

- At the top-level, we write
  - `let x = e;;` (* no `in e2` part *)
  - This is called a let *definition*, not a let *expression*
    - Because it doesn't, itself, evaluate to anything

- Omitting in means "from now on":

```
# let pi = 3.14;;
```

(* pi is now *bound* in the rest of the top-level scope *)

Let Expressions: Scope

- In `let x = e1 in e2`, var `x` is *not* visible outside of `e2`

```
let pi = 3.14 in pi *. 3.0 *. 3.0;;
```

(bind `pi` (only) in body of `let` (which is `pi *. 3.0 *. 3.0`) *)

```
print_float pi;; (* error: pi not bound *)
```

```
{
  float pi = 3.14;

  pi * 3.0 * 3.0;
}
pi; /* pi unbound! */
```

Examples – Scope of Let bindings

- `x;;` (* Unbound value x *)
- `let x = 1 in x + 1;;` (* 2 *)
- `let x = x in x + 1;;` (* Unbound value x *)
- `(let x = 1 in x + 1);;    x;;` (* Unbound value x *)
- `let x = 4 in (let x = x + 1 in x) ;;` (* 5 *)

Let Expressions in Functions

- You can use **`let`** inside of functions for local vars

```
let area d =
  let pi = 3.14 in
  let r = d /. 2.0 in
  pi *. r *. r
```

Shadowing Names

- Shadowing is rebinding a name in an inner scope to have a different meaning
  - May or may not be allowed by the language

C

```
int i;

void f(float i) {
   {
      char *i = NULL;
      ...
   }
}
```

```
let x = 10 in
    let z =
       let x = 20 in
       x*2 in
    x+z. (* 50 *)
```

Shadowing, by the Semantics

- What if ***e2*** is also a **`let`** for ***x***?
  - Substitution will **stop** at the ***e2*** of a shadowing ***x***

Example

```
    let x = 3+4 in let x = 3*x in x+1
    - let x = 7 in let x = 3*x in x+1
    - let x = 3*7 in x+1
    - let x = 21 in x+1
    - 21+1
    - 22
```

Will *not* be substituted, since it is shadowed by the inner let

Nested Functions

```
let filter lst =
  let rec aux l =
     match l with
      |[] -> []
      |h::t-> if h mod 2 <> 0 then h::aux t
        else aux t
    in aux lst

filter [1;2;3;4;5;6] (* int list = [1; 3; 5] *)
```

Tuples

- Constructed using `(e1, …, en)`
- Deconstructed using pattern matching
  - Patterns involve parens and commas, e.g., `(p1, p2, …)`

- Tuples are similar to C structs
  - But without field labels
  - Allocated on the heap
- Tuples can be heterogenous
  - Unlike lists, which must be homogenous
  - `(1, ["string1";"string2"])` is a valid tuple

Tuple Types

- Tuple types use `*` to separate components
  - Type joins types of its components
- Examples
  - `(1, 2) :`
  - `(1, "string", 3.5) :`
  - `(1, ["a"; "b"], 'c') :`
  - `[(1,2)] :`
  - `[(1, 2); (3, 4)] :`
  - `[(1,2); (1,2,3)] :`

Tuple Types

- Examples

• `(1, 2) :`	`int * int`
• `(1, "string", 3.5) :`	`int * string * float`
• `(1, ["a"; "b"], 'c') :`	`int * string list * char`
• `[(1,2)] :`	`(int * int) list`
• `[(1, 2); (3, 4)] :`	`(int * int) list`
• `[(1,2); (1,2,3)] :`	*`error`*

Because the first list element has type int * int, but the second has type int * int * int – list elements must all be of the same type

Pattern Matching Tuples

```
let plus3 t =
  match t with
   (x, y, z) -> x + y + z;;
```

```
plus3 : int*int*int -> int = <fun>
```

```
let plus3' (x, y, z) = x + y + z;;
```

Tuples Are A Fixed Size

- This OCaml definition

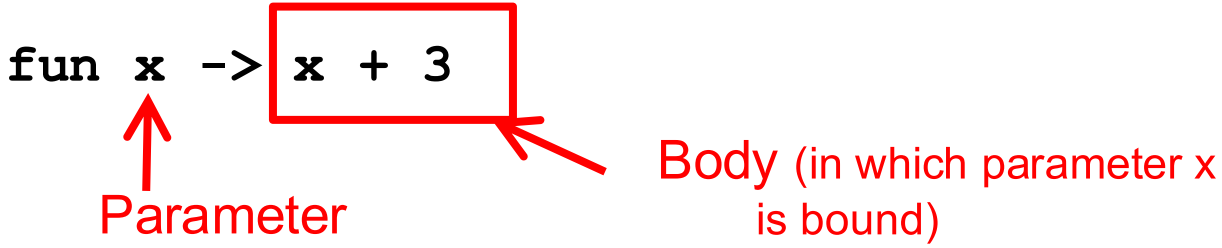
```
let foo x =
  match x with
     (a, b) -> a + b
  |  (a, b, c) -> a + b + c
```

has a type error. Why?

- Tuples of different size have different types
  - `(a, b) has type: 'a * 'b`
  - `(a, b, c) has type: 'a * 'b * 'c`

Anonymous Functions

- Use fun to make a function with no name



```
fun x -> x + 3
```

Parameter

Body (in which parameter x is bound)

```
(fun x -> x + 3) 5   = 8
```

- *anonymous functions* and *named functions* follow the same evaluation and typing rules. The only difference is whether the function is bound to a name.

Functions and Binding

- Functions are first-class, so you can bind them to other names as you like

```
let f x = x + 3;;
let g = f;;
g 5
= 8
```

let Shorthands

- let for functions is a syntactic shorthand

  `let f x = body` is semantically equivalent to

  `let f = fun x -> body`

- `let next x = x + 1`
  - Short for `let next = fun x -> x + 1`

- `let plus x y = x + y`
  - Short for `let plus = fun x y -> x + y`

Passing Functions as Arguments

You can pass functions as arguments

```
let plus3 x = x + 3 (* int -> int *)

let twice f z = f (f z)
(* ('a->'a) -> 'a -> 'a *)


 twice plus3 5 = 11
```

Records

- Records: identify elements by name
  - Elements of a tuple are identified by position

- Define a record type before defining record values

```
type date = { month: string; day: int; year: int }
```

- Define a record value

```
# let today = { day=16; year=2017; month=“f”^“eb” };;
today : date = { day=16; year=2017; month=“feb” };;
```

Destructing Records

```
type date = { month: string; day: int; year: int }
let today = { day=16; year=2017; month=“feb” };;
```

- Access by field name or pattern matching

```
today.month;; (* feb *)

let { year } = today in (* binds year to 2017 *)
let { month=_; day=d } = today in
…
```