

CMSC 330: Organization of Programming Languages

Lets, Tuples, 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

- $e1 \Rightarrow v1$

- $e2\{v1/x\}$

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

```
21
```

Let Expressions

- Syntax

- `let  $x$  =  $e1$  in  $e2$`

- Type checking

- If $e1 : t1$ and

- If assuming $x : t1$ implies $e2 : t$

- Then $(\text{let } x = e1 \text{ in } e2) : t$

Example

What is the type of `let  $z$  =  $3+4$  in  $3*z$` ?

- $3+4 : \text{int}$
- Assuming $z : \text{int}$, we have $3*z : \text{int}$
- 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;;  
print_float pi;;
```

error: `pi` not bound



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



```
{  
  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 *)

Nested Let Expressions

```
let res =  
  (let area =  
    (let pi = 3.14 in  
      let r = 3.0 in  
        pi *. r *. r) in  
    area /. 2.0) ;;
```

Similar scoping possibilities C and Java

```
float res;  
{ float area;  
  { float pi = 3.14  
    float r = 3.0;  
    area = pi * r * r;  
  }  
  res = area / 2.0;  
}
```


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;  
        ...  
    }  
}
```

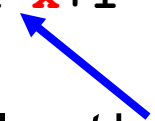
```
let x = 10 in  
  let z =  
    let x = 20 in  
      x*2 in  
x+z. (* 50 *)
```

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

Quiz 1: What does this evaluate to?

```
let x = 2 in  
let y = x + x in  
y * x
```

- A. 4
- B. 6
- C. 8
- D. Error

Quiz 1: What does this evaluate to?

```
let x = 2 in  
let y = x + x in  
y * x
```

A. 4

B. 6

C. 8

D. Error

Quiz 2: What does this evaluate to?

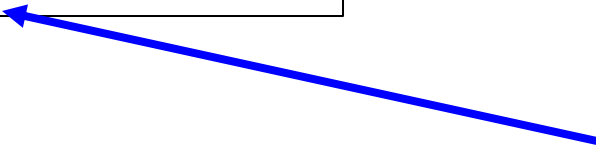
```
let x = 5 in  
x = 3
```

- A. 3
- B. 2
- C. true
- D. false

Quiz 2: What does this evaluate to?

```
let x = 2 in  
x = 3
```

- A. 3
- B. 2
- C. true
- D. false



This expression is checking whether **x** is equal to 3

Quiz 3: What does this evaluate to?

```
let y = 3 in  
let x = y+2 in  
let y = 6 in  
x+y
```

- A. 8
- B. 11
- C. 13
- D. 14

Quiz 3: What does this evaluate to?

```
let y = 3 in  
let x = y+2 in  
let y = 6 in  
x+y
```

A. 8

B. 11

C. 13

D. 14

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

- Tuple types use ***** to separate components
 - Type joins types of its components
- 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;;  
plusThree' : int*int*int -> int = <fun>
```

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`

Quiz 4: What does this evaluate to?

```
let get a b = (a+b,0) in  
get 1 2
```

- A. (3,0)
- B. (2,0)
- C. 3
- D. type error

Quiz 4: What does this evaluate to?

```
let get a b = (a+b,0) in  
get 1 2
```

A. (3,0)

B. (2,0)

C. 3

D. type error

Quiz 5: What does this evaluate to?

```
let get (a,b) y = a+y in  
get (2,1) 1
```

- A. 3
- B. type error
- C. 2
- D. 1

Quiz 5: What does this evaluate to?

```
let get (a,b) y = a+y in  
get (2,1) 1
```

A. 3

B. type error

C. 2

D. 1

CMSC 330

Organization of Programming Languages

OCaml
Higher Order Functions

Anonymous Functions

- Use `fun` to make a function with no name

Parameter  Body (in which parameter `x` is bound) 

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

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

= 8

Anonymous Functions

- Syntax

- **fun** *x1* ... *xn* $\rightarrow$ *e*

- Evaluation

- An anonymous function is an expression
 - In fact, *it is a value*.

- Type checking

- $(\text{fun } x1 \dots xn \rightarrow e) : (t1 \rightarrow \dots \rightarrow tn \rightarrow u)$
when $e : u$ under assumptions $x1 : t1, \dots, xn : tn$.
 - (Same rule as `let f x1 ... xn = e`)

Quiz 1: What does this evaluate to?

```
let y = (fun x -> x+1) 2 in  
(fun z -> z-1) y
```

- A. *Error*
- B. 2
- C. 1
- D. 0

Quiz 1: What does this evaluate to?

```
let y = (fun x -> x+1) 2 in  
(fun z -> z-1) y
```

A. *Error*

B. 2

C. 1

D. 0

Quiz 2: What is this expression's type ?

`(fun x y -> x) 2 3`

- A. *Type error*
- B. `int`
- C. `int -> int -> int`
- D. `'a -> 'b -> 'a`

Quiz 2: What is this expression's type ?

`(fun x y -> x) 2 3`

A. *Type error*

B. **int**

C. `int -> int -> int`

D. `'a -> 'b -> 'a`

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; ≠ 8
```

```
g 5
```

Example 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`

Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
let h = fun y -> g (y+1) in
h 1
```

- A. 0
- B. 1
- C. 2
- D. *Error*

Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
let h = fun y -> g (y+1)
h 1
```

A. 0

B. 1

C. 2

D. *Error*

Nested Functions

```
(* Filter the odd numbers from a list *)  
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] *)
```

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  
...
```

Quiz 6: What is the type of `shift`?

```
type point = {x:int; y:int}  
let shift { x = px } = [px]::[]
```

- A. `point -> int list`
- B. `int -> int list`
- C. `point -> point list`
- D. `point -> int list list`

Quiz 6: What is the type of `shift`?

```
type point = {x:int; y:int}  
let shift { x = px } = [px]::[]
```

- A. `point -> int list`
- B. `int -> int list`
- C. `point -> point list`
- D. `point -> int list list`