CMSC 330: Organization of Programming Languages

Lets, Tuples, Records

CMSC 330 - Spring 2021

- 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

- Syntax
 - let x = e1 in e2

- Evaluation
 - Evaluate e1 to v1
 - Substitute v1 for x in e2
 - yielding new expression e2'
 - Evaluate e2' to v2, the final result

Example x e1 e2 let z = 3+4 in 3*z
(evaluate e^{1}) > let z = 7 in 3*z
(substitute for var z in <u>e</u> 2) ➤ 3 *7
(compute the final result) ▶ 21

- Syntax
 - let x = e1 in e2
- Type checking

• So the type of let z = 3+4 in 3*z is int

- Syntax
 - let x = e1 in e2
- Type checking
 - If e1 : t1 and
 - If assuming x : t1 implies e2 : t
 - Then (let **x** = **e1** in **e2**): **t**

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 *)

Top-level expressions

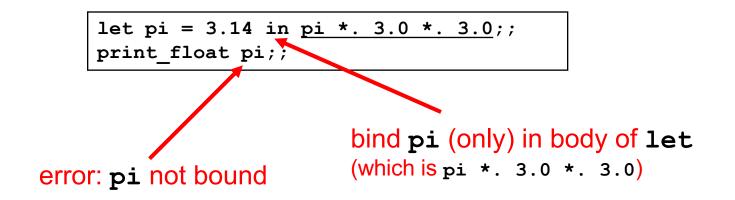
- We can write any expression at top-level, too
 - **e;;**
 - This says to evaluate e and then ignore the result
 - Equivalent to let _ = e;;
 - Useful when *e* has a side effect, such as reading/writing a file, printing to the screen, etc.

```
let x = 37;;
let y = x + 5;;
print_int y;;
print_string "\n";;
```

• When run, outputs 42 to the screen

Let Expressions: Scope

• In let x = e1 in e2, var x is not visible outside of e2



• 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;; (* pi unbound! *)
```

• Compare to similar usage in Java/C

```
{
  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 *)

Examples – Scope of Let bindings

Х;;

– (* Unbound value x *)

Nested Let Expressions

- Uses of let can be nested (last example on prev. slide)
 - Nested bound variables (pi and r) not visible outside
- Similar scoping possibilities C and Java

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

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

Nested Let Style: Generally Avoid

- Sometimes a nested binding can be rewritten in a more linear style
 – Fasier to understand
- Can go too far: namespace pollution
 - Avoiding adding unnecessary variable bindings to top-level

	let res =				
	(let area =				
	(let pi = 3.14 in)				
	let $r = 3.0$ in				
	pi *. r *. r) in				
	area /. 2.0);;				
let <mark>r</mark>	ces =	7			
let pi = 3.14 in					
let $\mathbf{r} = 3.0$ in					
let area = pi *. r *. r in					
area /. 2.0;;					

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

Let Expressions in Functions

• You can use let inside of functions for local vars

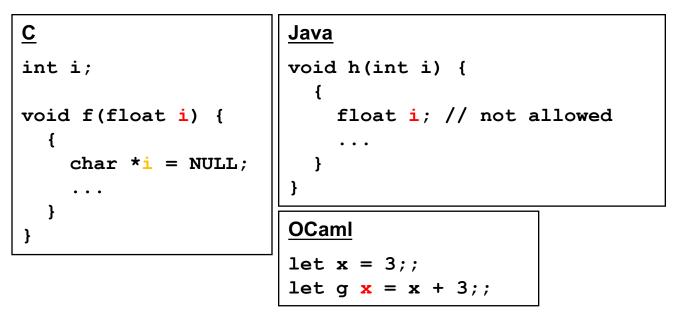
```
let area r =
   let pi = 3.14 in
   pi *. r *. r
```

- And you can use many lets in sequence

```
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



Shadowing, by the Semantics

- What if **e**2 is also a **let** for **x**?
 - Substitution will **stop** at the **e**2 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
```

Shadowing (of Locals) Discouraged

• You can use shadowing to simulate update (mutation)

```
let rec f x n =
    if x = 0 then 1
    else
        let x = x - 1 in (* shadowed *)
        n * (f x n)
```

- But avoiding shadowing is clearer
 - With no shadowing, if you see a variable x, you know it hasn't been "changed," no matter where it appears
 - If you want to "update" n, use a new name n1, n', etc.

Quiz 1

Which of these is **not** an expression that evaluates to 3?

A. let x=2 in x+1
B. let x=3 in x
C. let x=3
D. 3

Quiz 1

Which of these is **not** an expression that evaluates to 3?

A. let x=2 in x+1
B. let x=3 in x
C. let x=3 ---> not an expression
D. 3

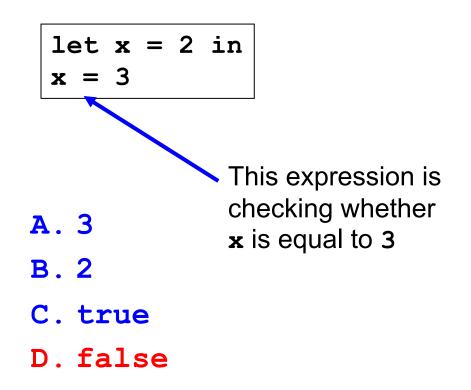
Quiz 2: What does this evaluate to?

let
$$x = 2$$
 in $x = 3$

A. 3

- **B.** 2
- C. true
- D. false

Quiz 2: What does this evaluate to?



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

let Specializes match

More general form of let allows patterns:

• let p = e1 in e2

 where *p* is a pattern. If *e1* fails to match that pattern then an exception is thrown

This pattern form of let is equivalent to

• match e1 with p -> e2

Examples

- let [x] = [[1]] in 1::x (* evals to [1;1] *)
- let h::_ = [1;2;3] in h (* evaluates to 1 *)
- let () = print_int 5 in 3 (* evaluates to 3 *)

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) :
 - (1, "string", 3.5) :
 - (1, ["a"; "b"], 'c') :
 - [(1,2)] :
 - [(1, 2); (3, 4)] :
 - [(1,2); (1,2,3)] :

```
int * int
```

```
int * string * float
```

```
int * string list * char
```

```
(int * int) list
```

```
(int * int) list
```

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 plusThree t =
   match t with
   (x, y, z) -> x + y + z;;
plusThree : int*int*int -> int = <fun>
```

```
# let plusThree' (x, y, z) = x + y + z;;
plusThree' : int*int*int -> int = <fun>
```

```
# let addOne (x, y, z) = (x+1, y+1, z+1);;
addOne : int*int*int -> int*int*int = <fun>
```

```
# plusThree (addOne (3, 4, 5));;
```

```
-: int = 15
```

Remember, **semicolon** for lists, **comma** for tuples

- [1, 2] = [(1, 2)] which is a list of size one
- (1; 2) Warning: This expression should have type unit

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
 - Patterns in the same **match** must have the same type

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. 3B. type errorC. 2D. 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

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

```
print_string today.month;; (* prints feb *)
(* patterns *)
let { month=_; day=d } = today in
let { year } = today in
let _ = print_int d in (* prints 16 *)
print_int year;; (* prints 2017 *)
```

- Notes:
 - In record patterns, you can skip or reorder fields
 - You can use the field name as the bound variable

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