CMSC 330: Organization of Programming Languages

Lets, Tuples, Records
Let Expressions

- Enable binding variables in other expressions
  - These are different from the `let definitions` we’ve been using at the top-level
- They are expressions, so they have a value

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

• 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`
  – Result of evaluation is `v2`

Example
  
  
  `let x = 3+4 in 3*x`
  `let x = 7 in 3*x`
  `3*7`
  `21`
Let Expressions

• Syntax
  – let $x = e_1$ in $e_2$

• Type checking
  – If $e_1 : t_1$ and $e_2 : t$ (assuming $x : t_1$)
  – Then let $x = e_1$ in $e_2 : t$

• Example: let $x = 3+27$ in $x*3$
  – $3+27 : \text{int}$
  – $x*3 : \text{int}$ (assuming $x : \text{int}$)
  – so let $x = 3+27$ in $x*3 : \text{int}$
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 an 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`, variable `x` is not visible outside of `e2`

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

bind `pi` (only) in body of `let`
(error: `pi` not bound)

(which is `pi *. 3.0 *. 3.0`)

Binding in other languages

• Compare to similar usage in Java/C

let pi = 3.14 in
  pi *. 3.0 *. 3.0;;
pi;; (* pi unbound! *)

{
  float pi = 3.14;
  pi * 3.0 * 3.0;
}
pi; /* pi unbound! */
Examples – Let

• \texttt{x;;}
  – (* Unbound value x *)

• \texttt{let x = 1 in x + 1;;}
  – (* 2 *)

• \texttt{let x = x in x + 1;;}
  – (* Unbound value x *)
Examples – Let

• let \( x = 1 \) in \((x + 1 + x) \);;
  – (* 3 *)

• \((\text{let } x = 1 \text{ in } x + 1) \);; \(x\);
  – (* Unbound value \( x \) *)

• let \( x = 4 \) in \((\text{let } x = x + 1 \text{ in } x)\);;
  – (* 5 *)
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;
      ...
    }
}
```

```ocaml
let g = 3;;
let g x = x + 3;;
```

```java
void h(int i) {
    { float i; // not allowed
      ...
    }
}
```
Let Expressions in Functions

• You can use `let` inside of functions for local vars

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

– And you can use many `lets` in sequence

```plaintext
let area d =
    let pi = 3.14 in
    let r = d /.* 2.0 in
    pi *.* r *.* r
```
Nested Let Expressions

- Uses of `let` can be nested in OCaml
  - Nested bound variables (`pi` and `r`) invisible outside

- Similar scoping possibilities C and Java

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

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

```c
float res;
{ float area;
  { float pi = 3.14
    float r = 3.0;
    area = pi * r * r;
  }
  res = area / 2.0;
}
Quiz 1

Which of these expressions does not evaluate to 3?

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

Which of these expressions does not evaluate to 3?

A. `let x=3`  ---> not an expression
B. `let x=2 in x+1`
C. `let x=3 in x`
D. 3
E. `let f x = x+1 in f 2`
Quiz 2: What does this evaluate to?

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

A. 2
B. 3
C. 4
D. 5
Quiz 2: What does this evaluate to?

```
let x = 2 in
let y = 2 in
x + y
```
Quiz 3: What does this evaluate to?

```plaintext
let x = 3 in
let y = 4 in
let x = 8 in
x = 10 - y
```

A. 6
B. true
C. 12
D. false
Quiz 3: What does this evaluate to?

```
let x = 3 in
let y = 4 in
let x = 8 in
x = 10-y
```

A. 6  
B. true  
C. 12  
D. false
Quiz 4: What does this evaluate to?

```
let x = 10 in
let y = x+2 in
let x = 8 in
y
```

A. 5
B. 12
C. 10
D. false
Quiz 4: What does this evaluate to?

```
let x = 10 in
let y = x + 2 in
let x = 8 in
y
```

A. 5
B. 12
C. 10
D. false
Tuples

• Constructed using \((e1, \ldots, en)\)
• Deconstructed using pattern matching
  – Patterns involve parens and commas, e.g., \((p1,p2, \ldots)\)
• Tuples are similar to C structs
  – But without field labels
  – Allocated on the heap
• Tuples can be heterogeneous
  – 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 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
More Examples With Tuples

- let sum ((a, b), c) = (a+c, b+c)
  - sum ((1, 2), 3) = (4, 5)

- let plusFirstTwo (x::y::_, a) = (x + a, y + a)
  - plusFirstTwo ([1; 2; 3], 4) = (5, 6)

- let tls (_::xs, _::ys) = (xs, ys)
  - tls ([1; 2; 3], [4; 5; 6; 7]) = ([2; 3], [5; 6; 7])
Tuples Are A Fixed Size

- This OCaml definition
  ```ocaml
  let foo x = match x with
  (a, b)  -> a + b
  | (a, b, c)  -> a + b + c;;
  ```

- Would yield this error message
  - This pattern matches values of type 'a * 'b * 'c
  but is here used to match values of type 'd * 'e

- Tuples of different size have different types
  - Thus never more than one match case with tuples
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 }

• Construct a record
  – { f1=e1; ...; fn=en } : evaluates e1 to en, assigns results to the given fields
  • Fields do not have to be written in order

  # let today = { day=16; year=2017; month="f"^"eb" };;
today : date = { day=16; year=2017; month="feb" };;
Destructing Records

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

- **Access** by field name or pattern matching

```ocaml
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 patterns, you can skip or reorder fields
  - You can use the field name as the bound variable
Quiz 5: What does this evaluate to?

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

A. 3
B. type error
C. 2
D. 1
Quiz 5: What does this evaluate to?

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

A. 3
B. type error – get’s first argument must be a pair
C. 2
D. 1
Quiz 6: What does this evaluate to?

```haskell
let get (x,y) = 
  match x with 
    (a,b) -> a+y 
in 
get (1,2) 1
```

A. 3  
B. type error  
C. 2  
D. 1
Quiz 6: What does this evaluate to?

```
let get (x,y) =
  match x with
  (a,b) -> a+y
in
get (1,2) 1
```

A. 3
B. type error – get takes only one argument
C. 2
D. 1
Quiz 7: What is the type of \texttt{shift}?

type point = \{x:int; y:int\}

let shift \{ x=px; y=py \} =
\{x=px+1; y=py+1\};;

A. point \rightarrow \text{bool list}
B. int list \rightarrow \text{int list}
C. point \rightarrow \text{point}
D. point \rightarrow \text{int list}
Quiz 7: What is the type of \texttt{shift}?

```
type point = {x:int; y:int}
let shift { x=px; y=py } = 
    {x=px+1; y=py+1};;
```

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