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 = e1 in e2`

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

• Example: `let x = 3+27 in x*3`
  - `3+27 : int`
  - `x*3 : int` (assuming `x:int`)
  - So `let x = 3+27 in x*3 : 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;;
```

**error:** `pi` not bound

**bind `pi` (only) in body of `let`**

(which is `pi *. 3.0 *. 3.0`)
Binding in other languages

- Compare to similar usage in Java/C

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

```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

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

• (let x = 1 in x + 1) ;; x;;
  – (* Unbound value x *)

• let x = 4 in (let x = x + 1 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);;

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 is **not** an expression that evaluates to 3?

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

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

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

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

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

\[
\begin{align*}
&\text{let } x = 2 \text{ in} \\
&\text{let } y = 3 \text{ in} \\
&x + y
\end{align*}
\]

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

```
let x = 6 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 = 6 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 = 3 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?

```plaintext
let x = 3 in
let y = x+2 in
let x = 8 in
y
```

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

- Constructed using \((e_1, \ldots, e_n)\)
- Deconstructed using pattern matching
  - Patterns involve parens and commas, e.g., \((p_1, p_2, \ldots)\)
- 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) : \texttt{int * int}
    - (1, "string", 3.5) : \texttt{int * string * float}
    - (1, ["a"; "b"], 'c') : \texttt{int * string list * char}
    - [(1,2)] : \texttt{(int * int) list}
    - [(1, 2); (3, 4)] : \texttt{(int * int) list}
    - [(1,2); (1,2,3)] : \texttt{error}

Because the first list element has type \texttt{int * int}, but the second has type \texttt{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;;`

- 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

```plaintext
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

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

```
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?

```ocaml
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?

```ocaml
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?

```ocaml
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 `shift`?

```plaintext
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
Quiz 7: What is the type of `shift`?

```plaintext
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