

CMSC 330: Organization of Programming Languages

OCaml Features

Tuples

- ▶ **Constructed** using `(e1, ..., en)`
- ▶ **Deconstructed** using pattern matching
- ▶ Tuples are like 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

CMSC 330

3

Tuples – More Examples

- ▶ `let t1s (_::xs, _::ys) = (xs, ys)`
 - `t1s ([1;2;3],[4;5;6;7]) = ([2;3],[5;6;7])`
- ▶ Remember
 - Semicolon for lists
 - Comma for tuples
- ▶ Example
 - `[1, 2] = [(1, 2)]` = a list of size one
 - `(1; 2)` = a syntax error

CMSC 330

5

OCaml Functions Take One Argument

- ▶ Recall this example

```
let plus (x, y) = x + y;;
plus (3, 4);;
```

- It looks like you're passing in two arguments
- ▶ Actually, you're passing in a **tuple** instead

```
let plus t = match t with
  (x, y) = x + y;;
plus (3, 4);;
```

- And using pattern matching to extract its contents

CMSC 330

2

Tuples – Examples

- ▶ `let plusThree (x, y, z) = x+y+z`
`let addOne (x, y, z) = (x+1, y+1, z+1)`
 - `plusThree (addOne (3,4,5)) = 15`
- ▶ `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)`

CMSC 330

4

Another Tuple Example

- ▶ Given
 - `let f l = match l with x::_::y -> (x,y)`
- ▶ What is the value of
 - `f [1;2;3;4]`
- ▶ Possibilities
 - `[[1],[3]]`
 - `(1,3)`
 - `(1,[3])`
 - `(1,4)`
 - `(1,[3;4])` ←

CMSC 330

6

List and Tuple Types

- ▶ Tuple types use `*` to separate 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`

CMSC 330

7

Type declarations

- ▶ `type` can be used to create new names for types
 - Useful for combinations of lists and tuples
- ▶ Examples
 - `type my_type = int * (int list)`
`(3, [1; 2]) : my_type`
 - `type my_type2 = int * char * (int * float)`
`(3, 'a', (5, 3.0)) : my_type2`

CMSC 330

8

Polymorphic Functions

- ▶ Some functions require specific list types
 - `let plusFirstTwo (x::y::_, a) = (x + a, y + a)`
 - `plusFirstTwo : int list * int -> (int * int)`
- ▶ But other functions work for a list of any type
 - `let hd (h::_) = h`
 - `hd [1; 2; 3] (* returns 1 *)`
 - `hd ["a"; "b"; "c"] (* returns "a" *)`
- ▶ These functions are **polymorphic**

CMSC 330

9

Polymorphic Types

- ▶ OCaml gives such functions **polymorphic** types
 - `hd : 'a list -> 'a`
 - Read as
 - > Function takes a list of any element type 'a
 - > And returns something of that type
- ▶ Example
 - `let tl (_::t) = t`
`tl : 'a list -> 'a list`

CMSC 330

10

Polymorphic Types (cont.)

- ▶ More Examples
 - `let swap (x, y) = (y, x)`
`swap : 'a * 'b -> 'b * 'a`
 - `let tls (_::xs, _::ys) = (xs, ys)`
`tls : 'a list * 'b list -> 'a list * 'b list`

CMSC 330

11

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

CMSC 330

12

Conditionals

- ▶ Use `if...then...else` just like C/Java
 - No parentheses and no end

```
if grade >= 90 then
  print_string "You got an A"
else if grade >= 80 then
  print_string "You got a B"
else if grade >= 70 then
  print_string "You got a C"
else
  print_string "You're not doing so well"
```

CMSC 330

13

Conditionals (cont.)

- ▶ In OCaml, conditionals return a result
 - The value of whichever branch is true/false
 - Like `?:` in C, C++, and Java
 - # `if 7 > 42 then "hello" else goodbye;;`
 - `: string = "goodbye"`
 - # `let x = if true then 3 else 4;;`
 - `x : int = 3`
 - # `if false then 3 else 3.0;;`
 - This expression has type float but is here used with type int

CMSC 330

14

The Factorial Function

- ▶ Using conditionals & functions
 - Can you write `fact`, the factorial function?

```
let rec fact n =
  if n = 0 then
    1
  else
    n * fact (n-1);;
```

- ▶ Notice no return statements
 - This is pretty much how it needs to be written

CMSC 330

15

Let Rec

- ▶ The `rec` part means “define a recursive function”
- ▶ Let vs. let rec
 - `let x = e1 in e2` `x` in scope within `e2`
 - `let rec x = e1 in e2` `x` in scope within `e2` and `e1`
- ▶ Why use let rec?
 - If you used `let` instead of `let rec` to define `fact`

```
let fact n =
  if n = 0 then 1
  else n * fact (n-1) in e1
```

Fact is not bound here!

CMSC 330

16

Examples – Semicolon

- ▶ Definition
 - `e1 ; e2` (* evaluate e1, evaluate e2, return e2)
- ▶ `1 ; 2 ;;`
 - (* 2 – value of 2nd expression is returned *)
- ▶ `(1 + 2) ; 4 ;;`
 - (* 4 – value of 2nd expression is returned *)
- ▶ `1 + (2 ; 4) ;;`
 - (* 5 – value of 2nd expression is returned to 1 + *)
- ▶ `1 + 2 ; 4 ;;`
 - (* 4 – because + has higher precedence than ; *)

CMSC 330

17

Examples – Let

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

CMSC 330

18

Examples – Let

- ▶ `let x = 1 in (x + 1 ; x) ;;`
 - (* 1 - ; has higher precedence than let ... in *)
- ▶ `(let x = 1 in x + 1) ; x;;`
 - (* Unbound value x *)
- ▶ `let x = 4 in (let x = x + 1 in x);;`
 - (* 5 *)

CMSC 330

19

Let – More Examples

- ▶ `let f n = 10;;`
 - `let f n = if n = 0 then 1 else n * f (n - 1);;`
 - `f 0;;` (* 1 *)
 - `f 1;;` (* 10 *)
- ▶ `let f x = f x;;`
 - (* Unbound value f *)

CMSC 330

20