CMSC 330: Organization of Programming Languages

Tuples, Types, Conditionals and Recursion

or

“How many different OCaml topics can we cover in a single lecture?”
Review

- **Functions**
  - `let [rec] <name> <inputs> = <output>`

- **Types**
  - `int, bool, string`

- **Lists**
  - `[i1; i2; i3]` or `i1::i2::i3::[]`

- **Matching**
  - `match <e> with
    - <p1> -> <e1>
    | <p2> -> <e2>`
Review

- let hd l = match l with (h::t) -> h
  - hd [1;2;3] (* evaluates to 1 *)

- let hd l = match l with (h::_) -> h
  - hd [] (* error! no pattern matches *)

- let tl l = match l with (h::t) -> t
  - tl [1;2;3] (* evaluates to [2; 3] *)
More Examples

- let \( f \ l = \)
  
  match \( l \) with \((h_1::(h_2::\_))\) -> \( h_1 + h_2 \)
  
  - \( f \ [1; 2; 3] \)
    - (* evaluates to 3 *)

- let \( g \ l = \)
  
  match \( l \) with \([h_1; h_2]\) -> \( h_1 + h_2 \)
  
  - \( g \ [1; 2] \)
    - (* evaluates to 3 *)
  
  - \( g \ [1; 2; 3] \)
    - (* error! no pattern matches *)

Exactly 2 element list \([h_1; h_2]\)

2+ element list \((h_1::(h_2::\_))\)
An Abbreviation

• `let f p = e`, where `p` is a pattern, is a shorthand for `let f x = match x with p -> e`.

• Examples
  - `let hd (h::_) = h`
  - `let tl (::_:t) = t`
  - `let f (x::y::_) = x + y`
  - `let g [x; y] = x + y`

• Useful if there’s only one acceptable input
Pattern Matching Lists of Lists

• You can do pattern matching on these as well

• Examples
  
  let addFirsts ((x::_) :: (y::_) :: _) = x + y
  • addFirsts [ [1; 2; 3]; [4; 5]; [7; 8; 9] ] = 5

  let addFirstSecond ((x::_) :: (_::y::_) :: _) = x + y
  • addFirstSecond [ [1; 2; 3]; [4; 5]; [7; 8; 9] ] = 6

• Note: You probably won’t do this much or at all
  – You’ll mostly write recursive functions over lists
  – We’ll see that soon
OCaml Functions Take One Argument

- Recall this example
  ```ocaml
  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
    - And using pattern matching

- Tuples are *constructed* using \((e_1, \ldots, e_n)\)
  - They’re like C structs but without field labels, and allocated on the heap
  - Unlike lists, tuples do *not* need to be homogenous
    - E.g., \((1, ["string1"; "string2"]\)) is a valid tuple

- Tuples are *deconstructed* using pattern matching
Examples with Tuples

- let plusThree (x, y, z) = x + y + z
  let addOne (x, y, z) = (x+1, y+1, z+1)
  - plusThree (addOne (3, 4, 5)) (* returns 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)

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

- Remember, semicolon for lists, comma for tuples
  - [1, 2] = [(1, 2)] = a list of size one
  - (1; 2) = a syntax error
Another Example

• let \( f \) \( l \) = match \( l \) with \( x::(_::y) \rightarrow (x,y) \)
• What is \( f \) \([1;2;3;4]\) ?
  
  \((1,[3;4])\)
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') :
  - [(1,2)] :
  - [(1, 2); (3, 4)] :
  - [(1,2); (1,2,3)] :
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
This form of “plus” is also valid:

```ocaml
let plus x y = x + y;;
plus 3 4;;
```

Its type is \texttt{int -> int -> int}

Recall the tuple version:

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

Its type is \texttt{int * int -> int}

The former is an example of \textit{currying} or \textit{partial evaluation}

We will discuss this later; for now, view it as OCaml’s special method of “multiple” parameter passing
Type declarations

• **type** can be used to create new names for types
  – useful for combinations of lists and tuples
  – like “typedef” in C

• 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
  ```
Polymorphic Types

- Some functions we saw require specific list types
  - let plusFirstTwo (x::y::_, a) = (x + a, y + a)
  - plusFirstTwo : int list * int -> (int * int)
- But other functions work for any list
  - let hd (h::_) = h
    - hd [1; 2; 3] (* returns 1 *)
    - hd ["a"; "b"; "c"] (* returns "a" *)
- OCaml gives such functions *polymorphic* types
  - hd : 'a list -> 'a
    - this says the function takes a list of any element type ‘a, and returns something of that type
Examples of Polymorphic Types

• let tl (_::t) = t
  - tl : 'a list -> 'a list

• 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
Tuples Are a Fixed Size

```ocaml
# let foo x = match x with
    (a, b) -> a + b
| (a, b, c) -> a + b + c;;
This pattern matches values of type 'a * 'b * 'c
but is here used to match values of type 'd * 'e
```

- Thus there's never more than one match case with tuples
  - So the shorted form is always usable:
    ```ocaml```
    ```
    let foo (a, b) = a + b
    ```
Conditionals

• Use *if...then...else* like C/Java
  – No parentheses and no end

```plaintext
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"
```
Conditionals (cont’d)

• In OCaml, conditionals return a result
  – The value of whichever branch is true/false
  – Like `?:` in C, C++, and Java

```ocaml
# 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
```
OCaml Equality

# let a1 = [1;2;3];;
val a1 : int list = [1; 2; 3]
# let a2 = [1;2;3];;
val a2 : int list = [1; 2; 3]
# a1 == a2;;
- : bool = false     (shallow equality)
# a1 = a2;;
- : bool = true      (deep equality)

- <> is negation of =
- != is negation of ==
Recursive Functions

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

- The `rec` part means “define a recursive function”
  - This is special for technical reasons
  - `let x = e1 in e2`  `x` in scope within `e2`
  - `let rec x = e1 in e2`  `x` in scope within `e2 and e1`
    - OCaml will complain if you use `let` instead of `let rec`
Recursion = Looping

• Recursion is essentially the only way to iterate
  – (The only way we’re going to talk about)

• Another example

```ml
let rec print_up_to (n, m) =
    print_int n; print_string "\n";
    if n < m then print_up_to (n + 1, m)
```
Lists and Recursion

• Lists have a recursive structure
  – And so most functions over lists will be recursive

```ocaml
let rec length l = match l with
  | [] -> 0
  | (_,::t) -> 1 + (length t)
```

– This is just like an inductive definition
  • The length of the empty list is zero
  • The length of a nonempty list is 1 plus the length of the tail

– Type of length function?
More examples of let (try to evaluate)

- `let x = 1 in x;;`
- `let x = x in x;;`
- `let x = 4 in
  let x = x + 1 in x;;`
- `let f n = 10;;
  let f n = if n = 0 then 1 else n * f (n - 1);;
  f 0;;
  f 1;;`
- `let g x = g x;;`
More examples of let

- `let x = 1 in x;; (* 1 *)`
- `let x = x in x;; (* error, x is unbound *)`
- `let x = 4 in
  let x = x + 1 in x;; (* 5 *)`
- `let f n = 10;;
  let f n = if n = 0 then 1 else n * f (n - 1);;
  f 0;; (* 1 *)
  f 1;; (* 10 *)`
- `let g x = g x;; (* error *)`
Recursion Exercises

• sum l (* sum of elts in l *)

• negate l (* negate elements in list *)

• last l (* last element of l *)
Recursion Exercises

• sum l (* sum of elts in l *)
  let rec sum l = match l with
      [] -> 0
    | (x::xs) -> x + (sum xs)

• negate l (* negate elements in list *)
  let rec negate l = match l with
      [] -> []
    | (x::xs) -> (-x) :: (negate xs)

• last l (* last element of l *)
  let rec last l = match l with
      [x] -> x
    | (x::xs) -> last xs