This Lecture

- Tuples
- Polymorphic functions & types
- User-defined data types
- Defining recursive functions with `let rec`
- Recursive functions

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
  ```ocaml
  let plus t = match t with
  (x, y) -> x + y;
  plus (3, 4);;
  ```
  - And using pattern matching to extract its contents

Tuples

- Constructed using `(el, ..., 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

Tuples – Examples

- `let plusThree (x, y, z) = x+y+z`
  ```ocaml
  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)`
  ```ocaml
  sum ((1, 2), 3) = (4,5)
  ```

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

Tuples – More Examples

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

- Remember
  - Semicolon for lists
  - Comma for tuples

- Example
  - `[1, 2] = [[1, 2]]` is a list of size one
  - `(1, 2)` is a syntax error
Another Tuple Example

- Given
  - \( \text{let } f \ 1 = \text{match } \ 1 \text{ with } x :: (_.;:y) \rightarrow (x,y) \)
- What is the value of
  - \( f \ [1;2;3;4] \)
- Possibilities
  - \((1,[3])\)
  - \((1,3)\)
  - \((1,[3])\)
  - \((1,4)\)
  - \((1,[3;4])\)

List and Tuple Types

- Tuple types use * to separate components
- Examples
  - \((1,2) : \text{int * int}\)
  - \((1,"string",3.5) : \text{int * string * float}\)
  - \((1,"a","b","c") : \text{int * string list * char}\)
  - \((1,2) : \text{(int * int) list}\)
  - \((1,2);(3,4)) : \text{(int * int) list}\)
  - \((1,2);(1,2,3)) : \text{error}\)

Polymorphic Functions

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

Polymorphic Types

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

Polymorphic Types (cont.)

- More Examples
  - \text{let swap (x, y) = (y, x)}
    - \text{swap : 'a * 'b} \rightarrow \text{'b * 'a}
  - \text{let tls (___::xs, ___::ys) = (xs, ys)}
    - \text{tls : 'a list * 'b list} \rightarrow \text{'a list * 'b list}

Tuples Are a Fixed Size

- This OCaml definition
  - \text{# let foo } x = \text{match } x \text{ with}
    - \text{(a, b)} \rightarrow \text{a + b}
    - \text{| (a, b, c)} \rightarrow \text{a + b + c};
- \text{Would yield this error message}
  - This pattern matches values of type \text{'a * 'b * 'c}
    - but is here used to match values of type \text{'d * 'e}
- Tuples of different size have different types
  - Thus never more than one match case with tuples
OCaml Data

- So far, we’ve seen the following kinds of data
  - Basic types (int, float, char, string)
  - Lists
    - One kind of data structure
    - A list is either [] or h:t, deconstructed with pattern matching
  - Tuples
    - Let you collect data together in fixed-size pieces
  - Functions

- How can we build other data structures?
  - Building everything from lists and tuples is awkward

Data Types

- `type` can also be used to create variant types
  - Equivalent to C-style unions

```ocaml
type shape =
  | Rect of float * float (*) width * length *)
  | Circle of float (*) radius *)
```

- `Rect` and `Circle` are type constructors
  - Here a `shape` is either a `Rect` or a `Circle`

Data Types (cont.)

```ocaml
let area s =
  match s with
  | Rect (w, l) -> w * l
  | Circle r -> r * r * 3.14
area (Rect (3.0, 4.0))
area (Circle 3.0)
```

- Use pattern matching to deconstruct values
  - `s` is a `shape`
  - Do different things for `s` depending on its constructor

Data Types (cont.)

- What’s the type of `l`?
  - `shape list`

- What’s the type of `l`’s first element?
  - `shape`

User Defined Types

- `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`

Data Types Constructor

- Constructors must begin with an uppercase letter
- The arity of a constructor
  - Is the number of arguments it takes
  - A constructor with no arguments is nullary

```ocaml
type optional_int =
  None
  | Some of int
```

- Example
  - Arity of `None` = 0
  - Arity of `Some` = 1
Polymorphic Data Types

• This option type can work with any kind of data
  • In fact, this option type is built into OCaml

```
type optional_int =
  None
| Some of int
let add_with_default a = function
  None -> a + 42
| Some n -> a + n
add_with_default 3 None (* 45 *)
add_with_default 3 (Some 4) (* 7 *)
```

Recursive Data Types

• We can build up lists this way

```
type 'a list =
  Nil
| Cons of 'a * 'a list
let rec len = function
  Nil -> 0
| Cons (_, t) -> 1 + (len t)
len (Cons (10, Cons (20, Cons (30, Nil))))
```

Data Type Representations

• Values in a data type are stored
  1. Directly as integers
  2. As pointers to blocks in the heap

```
type t =
  A of int
| B
| C of int * int
| D
```

Exercise: A Binary Tree Data Type

• Write type bin_tree for binary trees over int
  • Trees should be ordered (binary search tree)
  • Implement the following
    - empty : bin_tree
    - is_empty : bin_tree -> bool
    - member : int -> bin_tree -> bool
    - insert : int -> bin_tree -> bin_tree
    - remove : int -> bin_tree -> bin_tree
    - equal : bin_tree -> bin_tree -> bool
    - fold : (int -> 'a -> 'a) -> bin_tree
      -> 'a
```

Type Annotations

• The syntax (e : t) asserts that “e has type t”
  • This can be added anywhere you like
    let (x : int) = 3
    let z = (x : int) + 5
  • Use to give functions parameter and return types
    let fn (x:int):float = (float_of_int x) *. 3.14
    • Note special position for return type
    • Thus let g x:int = ... means g returns int
  • Not needed for this course
  • But can be useful for debugging
    • Especially for more complicated types

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"
```
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
    ```

- The `let` 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 e2
    ```

Examples – Let

- `let x = … in x ; ;`
  - (* Unbound value x *)

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

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

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 *)

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

Examples – Let

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

- `let f x = … f … in … f …`
  - (* Unbound value f *)

- `let rec f x = … f … in … f …`
  - (* Bound value f *)

Let – More Examples
Recursion = Looping

- Recursion is essentially the only way to iterate
  - The only way we're going to talk about, anyway
  - Feature of functional programming languages

- Another example

```
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

```
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

Examples – Recursive Functions

- 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)

Examples – Recursive Functions

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

- append (l, m)
  (* list containing all elements in list l followed by all elements in list m *)
  let rec append (l, m) = match l with
    [] -> m
    | (x::xs) -> x::(append (xs, m))

Examples – Recursive Functions

- rev l (* reverse list; hint: use append *)
  let rec rev l = match l with
    [] -> []
    | (x::xs) -> append ((rev xs), [x])

- rev takes O(n^2) time. Can you do better?

A Clever Version of Reverse

```
let rec rev_helper (l, a) = match l with
  [] -> a
  | (x::xs) -> rev_helper (xs, (x::a))
let rev l = rev_helper (l, [])
```

- Let's give it a try
  rev [1; 2; 3] →
  rev_helper ([1;2;3], [1]) →
  rev_helper ([2;3], [1]) →
  rev_helper ([3], [2;1]) →
  rev_helper ([], [3;2;1]) →
  [3;2;1]
Examples – Recursive Functions

- `flattenPairs l (* ('a * 'a) list -> 'a list *)`
  ```ocaml```
  let rec flattenPairs l = match l with
  | [] -> []
  | ((a, b)::t) -> a :: b :: (flattenPairs t)
  ```

- `take (n, l) (* return first n elements of l *)`
  ```ocaml```
  let rec take (n, l) =
  | (0, []) -> []
  | (n, []) -> []
  | (n, x::xs) -> x :: (take (n-1, xs))
  ```

Working with Lists

- Several of these examples have the same flavor
  - Walk through the list and do something to every element
  - Walk through the list and keep track of something

- Recall the following example code from Ruby:
  ```ruby```
  ```
  a = [1,2,3,4,5]
  b = a.collect { |x| -x }
  ```
  - Here we passed a code block into the `collect` method
  - Wouldn’t it be nice to do the same in OCaml?