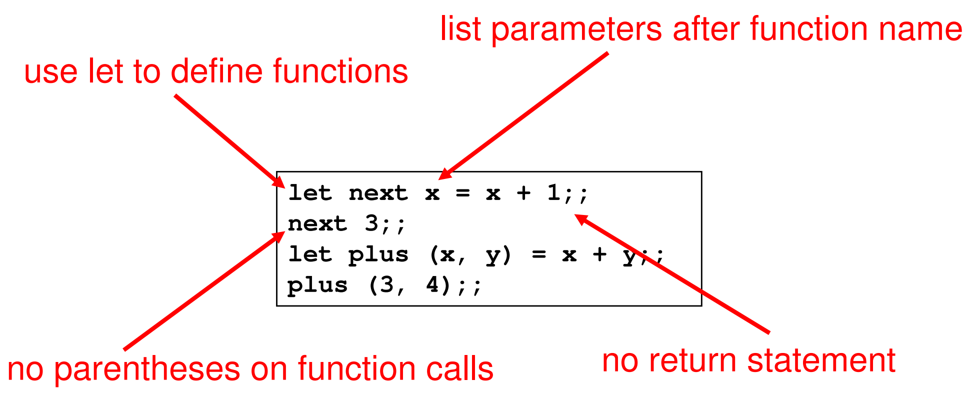
CMSC 330: Organization of Programming Languages

Functional Programming with OCaml, con't.

Defining Functions


list parameters after function name
use let to define functions
no parentheses on function calls
no return statement

```
let next x = x + 1;;
next 3;;
let plus (x, y) = x + y;;
plus (3, 4);;
```

Local Variables

- You can use let inside of functions for locals

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

  – and you can use as many lets as you want

```
let area d =
  let pi = 3.14 in
  let r = d /. 2.0 in
  pi *. r *. r
```

Function Types

- In OCaml, `->` is the function type constructor
  – The type `t1 -> t2` is a function with argument or *domain* type `t1` and return or *range* type `t2`

- Examples

```
– let next x = x + 1 (* type int -> int *)
– let fn x = (float_of_int x) *. 3.14
                     (* type int -> float *)
– print_string       (* type string -> unit *)
```

- Type a function name at top level to get its type

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`
- Very useful for debugging, especially for more complicated types

;; versus ;

- ;; ends an expression in the top-level of OCaml
  - use it to say: "Give me the value of this expression"
  - not used in the body of a function
  - not needed after each function definition
    - though for now it won't hurt if used there
- e1; e2 evaluates e1 and then e2, and returns e2

```
let print_both (s, t) = print_string s; print_string t;
                        "Printed s and t."
```

  - notice no ; at end– it's a *separator*, not a *terminator*

```
print_both ("Larry Herman ", "is amazing!")
```

Prints `"Larry Herman is amazing!"`, and returns `"Printed s and t."`

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 ; *)

Lists in OCaml

- The basic data structure in OCaml is the list
  - lists are written as [e1; e2; ...; en]

```
# [1;2;3];;
- : int list = [1;2;3]
```

  - notice int list – lists must be *homogeneous*
  - the empty list is []

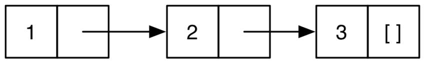
```
# [];;
- : 'a list
```

  - the 'a means "a list containing anything"
    - we'll see more about this later
  - warning: don't use a comma instead of a semicolon (it means something different; we'll see in a bit)

Consider a Linked List in C

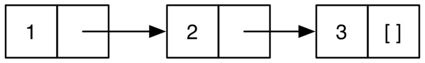
```
struct list {
  int elt;
  struct list *next;
};
…
struct list *l;
…
i= 0;
while (l != NULL) {
  i++;
  l= l->next;
}
```

Lists in OCaml are Linked



- [1;2;3] is represented above
  - a nonempty list is a pair (element, rest of list)
  - the element is the *head* of the list
  - the pointer is the *tail* or *rest* of the list- which is itself a list!
- Thus in math a list is either
  - the empty list []
  - or a pair consisting of an element and a list
    - this recursive structure will come in handy shortly

Lists are Linked (cont'd)



- :: prepends an element to a list
  - h::t is the list with h as the element at the beginning and t as the "rest"
  - :: is called a *constructor*, because it builds a list
  - although it's not emphasized, :: does allocate memory
- Examples

3::[]	(* The list [3] *)
2::(3::[])	(* The list [2; 3] *)
1::(2::(3::[]))	(* The list [1; 2; 3] *)

More Examples

```
# let y = [1;2;3] ;;
val y : int list = [1; 2; 3]
# let x = 4::y ;;
val x : int list = [4; 1; 2; 3]
# let z = 5::y ;;
val z : int list = [5; 1; 2; 3]
```

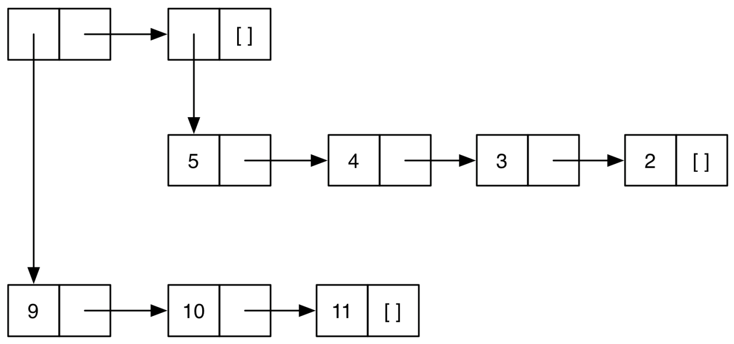
- ***not* modifying existing lists, just creating new lists**

```
# let w = [1;2]::y ;;
This expression has type int list but is here
  used with type int list list
```

- the left argument of :: is an element
- can you construct a list y such that [1;2]::y is valid?

Lists of Lists

- Lists can be nested arbitrarily
  - example: `[ [9; 10; 11]; [5; 4; 3; 2] ]`
    - (type `int list list`)



Practice

- What is the type of
  - [1;2;3] `int list`
  - [[[ ]; [ ]; [1.3;2.4]]] `float list list list`
  - let func x = x::(0::[ ]) `int -> int list`

Pattern Matching

- To pull lists apart, use the match construct

  match e with p_1 -> e_1 | ... | p_n -> e_n
- $p_1 \ldots p_n$ are *patterns* made up of [], ::, and *pattern variables*
- match finds the first pattern p_k that matches the shape of e
- Then the pattern variables in p_k are bound to the corresponding parts of e while e_k is evaluated and returned
- An underscore (_) is a wildcard pattern that matches with anything, and doesn't bind the value of what it matches to (useful when you want to know something matches, but don't care what its value is)

More on the match Construct

```
   match e with p1 -> e1 | ... | pn -> en


let is_empty l = match l with
     [] -> true
  | (h::t) -> false

   is_empty []          (* evaluates to true *)
   is_empty [1]         (* evaluates to false *)
   is_empty [1;2;3]     (* evaluates to false *)
```

Pattern Matching (cont'd)

- `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] *)`

Pattern Matching, con't. – Wildcards

- Code using _
  - `let is_empty l = match l with`
    `[] -> true | (_::_) -> false`
  - `let hd l = match l with (h::_) -> h`
  - `let tl l = match l with (_::t) -> t`
- Outputs
  - `is_empty [1] (* evaluates to false *)`
  - `is_empty []  (* evaluates to true  *)`
  - `hd [1;2;3]   (* evaluates to 1     *)`
  - `tl [1;2;3]   (* evaluates to [2;3] *)`
  - `hd [1]       (* evaluates to 1     *)`
  - `tl [1]       (* evaluates to []    *)`

Missing Cases

- Exceptions for inputs that don't match any pattern
  - OCaml will warn you about non-exhaustive matches
- Example:

```
# let hd l = match l with (h::_) -> h;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]

# hd [];;
Exception: Match_failure ("", 1, 11).
```

More Examples

- `let f l =`
  `match l with (h1::(h2::_)) -> h1 + h2`
  - `f [1;4;8]     (* evaluates to 5 *)`
- `let g l =`
  `match l with [h1; h2] -> h1 + h2`
  - `g [1; 2]      (* evaluates to 3 *)`
  - `g [1; 2; 3]   (* error!  no pattern matches *)`

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; 3; 5]; [2; 4]; [9; 8; 7] ] = 5`

- 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

```
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 `(e1, ..., en)`
  - 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