Let bindings

We use `let` to bind name (identifier) to a value:

```
# let x = 100;; (* x is an immutable binding 100 *)
val x : int = 100
```

Since functions are values, just like `ints` or `strings`, `let` is also used to define functions:

```
#let add x y = x + y;;
val add : int -> int -> int
```
Type Annotations

- OCaml compiler infers the types. But type inference is tricky. It gives vague error messages. We can annotate types manually.

- The syntax \((e : t)\) asserts that “\(e\) has type \(t\)”.
  
  ```ocaml
  let (x : int) = 3
  let z = (x : int) + 5
  ```

- Define functions’ parameter and return types
  
  ```ocaml
  let add (x:int) (y:int):int = x + y
  let id x = x (* 'a → 'a *)
  let id (x:int) = x (* int → int *)
  ```

- Checked by compiler: Very useful for debugging.
CMSC 330: Organization of Programming Languages

Functional Programming with Lists
Lists in OCaml

• The basic data structure in OCaml
  – Lists can be of *arbitrary length*
    • Implemented as a linked data structure
  – Lists must be *homogeneous*
    • All elements have the same type

• Operations
  – Construct lists
  – Destruct them via pattern matching
Constructing Lists: Syntax

Syntax

• `[]` is the empty list (pronounced “nil”)
• `e₁ :: e₂` prepends element `e₁` to list `e₂`
  – `e₁` is the head, `e₂` is the tail

• `[e₁; e₂; …; en]` is syntactic sugar for `e₁ :: e₂ :: … :: en :: []`

Examples

3 :: [] (* [3] *)
2 :: (3 :: []) (* [2; 3] *)
[1; 2; 3] (* 1 :: (2 :: (3 :: [])) *)
Constructing Lists: Evaluation

Evaluation

• [] is a value
• [e₁; ...; eₙ] evaluates to a list of [v₁; ...; vₙ]
  - Where
  - e₁ ⇒ v₁,
  - ...,
  - eₙ ⇒ vₙ
Constructing Lists: Examples

# let y = [1; 1+1; 1+1+1] ;;
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]

# let m = "hello"::"bob"::[];;
val m : string list = ["hello"; "bob"]
Constructing Lists: Typing

Nil:
[]: 'a list (* empty list *)

Cons:
If \(e_1 : t\) and \(e_2 : t\ list\) then \(e_1 :: e_2 : t\ list\)
Examples

```ocaml
# let x = [1;"world"] ;;
This expression has type string but an expression was expected of type int

# let m = [[1];[2;3]];;
val y : int list list = [[1]; [2; 3]]

# let y = 0::[1;2;3] ;;
val y : int list = [0; 1; 2; 3]

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

\[ [1;2;3] \] is represented as:

```
1 → 2 → 3 → []
```

head

tail
Lists of Lists

• Lists can be nested arbitrarily
  – Example: `[ [9; 10; 11]; [5; 4; 3; 2] ]`
  • Type `int list list`, also written as `(int list) list`
Lists are Immutable

• No way to *mutate* (change) an element of a list
• Instead, build up new lists out of old, e.g., using `::`

```plaintext
let x = [1;2;3;4]
let y = 5::x
let z = 6::x
```

![Diagram showing list operations](image-url)
Quiz 1

What is the type of the following expression?

\[ [1.0; 2.0; 3.0; 4.0] \]

A. array
B. list
C. float list
D. int list
What is the type of the following expression?

[1.0; 2.0; 3.0; 4.0]

A. array
B. list
C. float list
D. int list
Quiz 2

What is the type of the following expression?

\[ 10::[20] \]

A. int  
B. int list  
C. int list list  
D. error
Quiz 2

What is the type of the following expression?

10::[20]

A. int
B. int list
C. int list list
D. error
Quiz 3

What is the type of the following definition?

```ocaml
let f a = "umd" :: [a]
```

A. string -> string
B. string list
C. string list -> string list
D. string -> string list
What is the type of the following definition?

```haskell
let f a = "umd"::[a]
```

A. string -> string
B. string list
C. string list -> string list
D. string -> string list
Pattern Matching

• To pull lists apart, use the `match` construct
• Syntax

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

• `p1...pn` are *patterns*
• `e1...en` are *branch expressions*
Pattern Matching Example

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

Example runs

- `is_empty []` (* true *)
- `is_empty [1]` (* false *)
- `is_empty [1;2]` (* false *)
Pattern Matching Example (cont.)

let hd l =
    match l with
    (h::t) -> h

• Example runs
  - hd [1;2;3] (* 1 *)
  - hd [2;3] (* 2 *)
  - hd [3] (* 3 *)
  - hd [] (* Exception: Match_failure *)
Pattern Matching Example (cont.)

let neg n =  
    match n with  
      |true-> false  
      |_-> true

let is_empty l =  
    match l with  
      [] -> true  
      |_-> false

• An underscore _ is a wildcard pattern. It matches anything
To what does the following expression evaluate?

```haskell
match [1;2;3] with
  [[]] -> [0]
  [h::t] -> t
```

A. []
B. [0]
C. [1]
D. [2;3]
Quiz 4

To what does the following expression evaluate?

```plaintext
match [1;2;3] with
  [[]] -> [0]
  [h::t] -> t
```

A. []
B. [0]
C. [1]
D. [2;3]
"Deep" pattern matching

• \texttt{a::b} matches lists with \textbf{at least one} element

• \texttt{a::[]} matches lists with \textbf{exactly one} element

• \texttt{a::b::[]} matches lists with \textbf{exactly two} elements

• \texttt{a::b::c::d} matches lists with \textbf{at least three} elements
Quiz 5

To what does the following expression evaluate?

```
match [1;2;3] with
  | 1::[]    -> [0]
  | _::_     -> [1]
  | 1::_::[] -> []
A. []
B. [0]
C. [1]
D. [2;3]
```
Quiz 5

To what does the following expression evaluate?

```match [1;2;3] with
  | 1::[]    -> [0]
  | _::_     -> [1]
  | 1::_::_::[] -> []
```

A. []
B. [0]
C. [1]
D. [2;3]
Pattern Matching – An Abbreviation

• \texttt{let } f \ p = e, \textit{ where } p \textit{ is a pattern}
  – is shorthand for \texttt{let } f \ x = \texttt{match } x \texttt{ with } p \rightarrow e

• Examples
  – \texttt{let } hd \ (h::_\) = h
  – \texttt{let } tl \ (_::t\) = t

• Useful if there’s only one acceptable input
Polymorphic Types

• The `hd` function works for any type of list
  - `hd [1; 2; 3]`  (* 1 *)
  - `hd ["a"; "b"; "c"]`  (* "a" *)

• OCaml gives such functions polymorphic types
  - `hd : 'a list -> 'a`

• These are basically generic types in Java
  - `'a list` is like `List<T>`
Examples Of Polymorphic Types

let tl (_::t) = t

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

# tl [1.0; 2.0];;
- : float list = [2.0]
(* tl : 'a list -> 'a list *)
Examples Of Polymorphic Types

```ocaml
let eq x y = (x = y)

# eq 1 2;;
- : bool = false

# eq "hello" "there";;
- : bool = false

# eq "hello" 1 -- type error

(* eq : 'a -> 'a -> bool *)
```
Quiz 6

What is the type of the following function?

```
let f x y =
  if x = y then 1 else 0
```

A. 'a -> 'b -> int
B. 'a -> 'a -> bool
C. 'a -> 'a -> int
D. int
Quiz 6

What is the type of the following function?

```ml
let f x y = if x = y then 1 else 0
```

A. ‘a -> ‘b -> int
B. ‘a -> ‘a -> bool
C. ‘a -> ‘a -> int
D. int
Missing Cases

• Exceptions for inputs that don’t match any pattern
  – OCaml will warn you about non-exhaustive matches

• Example:

  ```ocaml
  # 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).
  ```
Pattern matching is **AWESOME**

1. You can’t forget a case
   - Compiler issues inexhaustive pattern-match warning
2. You can’t duplicate a case
   - Compiler issues unused match case warning
3. You can’t get an exception
   - Can’t do something like `List.hd []`
4. Pattern matching leads to elegant, concise, beautiful code
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`?
    • `'a list -> int`
More Examples

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

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

• **last l** (* last element of l *)
  
  ```ocaml
  let rec last l = match l with
    [x] -> x
  | (x::xs) -> last xs
  ```
More Examples (cont.)

(* return a list containing all the elements in the list l followed by all the elements in list m *)

• append l m

  let rec append l m = match l with
      [] -> m
    | (x::xs) -> x::(append xs m)

• rev l  (* reverse list; hint: use append *)

  let rec rev l = match l with
      [] -> []
    | (x::xs) -> append (rev xs) (x::[])