Functional Programming in O'Caml
…continued

CS 4100
Gordon Stewart
Ohio University

(slides adapted with permission
from Dave Walker,
COS326 Princeton University)

Quizzes

- Every Tuesday, we'll have a quiz with probability 1/3

1

2

3

Quizzes

- Every Tuesday, we'll have a quiz with probability 1/3

Quiz 1

On a sheet of paper (or half a sheet borrowed from a friend), write

1. Your name
2. The answer to the following question:

What is the type of

the OCaml expression

```
if false then 42 else 27
```

?

RECAP:
TYPE CHECKING RULES

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Violating the rules:

"hello" : string	(By rule 2)
1 : int	(By rule 1)
1 + "hello" : ??	(NO TYPE! Rule 3 does not apply!)

Type Checking Rules

- Violating the rules:

```
# "hello" + 1;;
Error: This expression has type string but an
expression was expected of type int
```

- The type error message tells you the type that was expected and the type that it inferred for your subexpression
- By the way, this was one of the nonsensical expressions that did not evaluate to a value
- I consider it a good thing that this expression does not type check

Type Checking Rules

- Violating the rules:

```
# "hello" + 1;;
Error: This expression has type string but an
expression was expected of type int
```

- A possible fix:

```
# "hello" ^ (string_of_int 1);;
- : string = "hello1"
```

- *One of the keys to becoming a good ML programmer is to understand type error messages.*

Type Checking Rules

- More rules:

(7) true : bool

(8) false : bool

(9) if e1 : bool
and e2 : t and e3 : t (for some type t)
then if e1 then e2 else e3 : t

- Using the rules:

if ???? then ???? else ???? : int

Type Checking Rules

- More rules:

(7) true : bool

(8) false : bool

(9) if e1 : bool
and e2 : t and e3 : t (for some type t)
then if e1 then e2 else e3 : t

- Using the rules:

if true then ???? else ???? : int

Type Checking Rules

- More rules:

(7) true : bool

(8) false : bool

(9) if e1 : bool
and e2 : t and e3 : t (for some type t)
then if e1 then e2 else e3 : t

- Using the rules:

if true then 7 else ???? : int

Type Checking Rules

- More rules:

(7) true : bool

(8) false : bool

(9) if e1 : bool
and e2 : t and e3 : t (for some type t)
then if e1 then e2 else e3 : t

- Using the rules:

if true then 7 else 8 : int

Type Checking Rules

- More rules:

(7) true : bool

(8) false : bool

(9) if e1 : bool
and e2 : t and e3 : t (for some type t)
then if e1 then e2 else e3 : t

- Violating the rules

if false then "1" else 2 : ????

types don't agree -- one is a string and one is an int

Type Checking Rules

- Violating the rules:

```
# if true then "1" else 2;;
Error: This expression has type int but an
expression was expected of type string
#
```

Type Checking Rules

- What about this expression:

```
# 3 / 0 ;;
Exception: Division_by_zero.
```

- Why doesn't the ML type checker do us the favor of telling us the expression will raise an exception?

Type Checking Rules

- What about this expression:

```
# 3 / 0 ;;
Exception: Division_by_zero.
```

- Why doesn't the ML type checker do us the favor of telling us the expression will raise an exception?
  - In general, detecting a divide-by-zero error requires we know that the divisor evaluates to 0.
  - In general, deciding whether the divisor evaluates to 0 requires solving the halting problem:

```
# 3 / (if turing_machine_halts m then 0 else 1);;
```

- There are type systems that will rule out divide-by-zero errors, but they require programmers supply proofs to the type checker

OCAML BASICS:
RECAP FROM LAST WEEK

OCaml

OCaml is a *functional* programming language

- Java gets most work done by *modifying* data
- OCaml gets most work done by producing *new*, *immutable* data

OCaml is a typed programming language

- the *type* of an expression *correctly predicts* the kind of *value* the expression will generate when it is executed
- types help us *understand* and *write* our programs

PART II:
LET DECLARATIONS, TUPLES

Abstraction & Abbreviation

- Good programmers identify repeated patterns in their code and factor out the repetition into meaning components
- In O'Caml, the most basic technique for factoring your code is to use let expressions
- Instead of writing this expression:

```
(2 + 3) * (2 + 3)
```

- We write this one:

```
let x = 2 + 3 in
x * x
```

A Few More Let Expressions

```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
```

A Few More Let Expressions

```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
```

```
let a = "a" in
let b = "b" in
let ax = a ^ a ^ a in
let bx = b ^ b ^ b in
ax ^ bx
```

Abstraction & Abbreviation

- Two kinds of let:

```
if tuesday() then
     let x = 2 + 3 in
     x + x
else
     0
;;
```

let … **in** … is an *expression* that can appear inside any other *expression*

The scope of x does not extend outside the enclosing "in"

```
let x = 2 + 3 ;;

let y = x + 17 / x ;;
```

let … ;; without "in" is a top-level *declaration*

Variables x and y may be exported; used by other modules

(Don't need ;; if another let comes next; do need it if expression next)

Binding Variables to Values

- Each OCaml variable is *bound* to 1 value
- *The value to which a variable is bound to never changes*!

```
let x = 3 ;;

let add_three (y:int) : int = y + x ;;
```

Binding Variables to Values

- Each OCaml variable is *bound* to 1 value
- *The value to which a variable is bound to never changes*!

```
let x = 3 ;;

let add_three (y:int) : int = y + x ;;
```

It does not matter what I write next. add_three will always add 3!

Binding Variables to Values

- Each OCaml variable is bound to 1 value
- *The value a variable is bound to never changes*!

a distinct variable that "happens to be spelled the same"

```
let x = 3 ;;

let add_three (y:int) : int = y + x ;;

let x = 4 ;;

let add_four (y:int) : int = y + x ;;
```

Binding Variables to Values

- Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them

rename x to zzz if you want to, replacing its uses

```
let x = 3 ;;

let add_three (y:int) : int = y + x ;;

let zzz = 4 ;;

let add_four (y:int) : int = y + zzz ;;

let add_seven (y:int) : int =
  add_three (add_four y)
;;
```

Binding Variables to Values

- Each OCaml variable is bound to 1 value
- OCaml is a statically scoped language

we can use add_three without worrying about the second definition of x

```
let x = 3 ;;

let add_three (y:int) : int = y + x ;;

let x = 4 ;;

let add_four (y:int) : int = y + x ;;

let add_seven (y:int) : int =
  add_three (add_four y)
;;
```

How do let expressions operate?

General rule: evaluate **e1**

```
let x = e1 in e2
```

Then substitute the resulting value for the variable **x** everywhere **x** appears in **e2**

How do let expressions operate?

Example:

```
let x = 2 + 1 in x * x
```

How do let expressions operate?

Example:

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

How do let expressions operate?

Example:

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

-->

```
3 * 3
```

substitute
3 for x

How do let expressions operate?

Example:

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

substitute
3 for x

-->

```
3 * 3
```

-->

```
9
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4     in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4     in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4     in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

-->

```
8
```

Moral: Let operates by substituting computed values for variables

OCAML BASICS:
TYPE CHECKING AGAIN

Type-checking Rules

There are simple rules that tell you what the type of an expression is.

Those rules compute a type for an expression based on the *types* of its subexpressions (and the types of the variables that are in scope).

You don't have to know the details of how a subexpression is implemented to do type checking. You just need to know its type.

That's what makes OCaml type checking *modular*.

We write "e : t" to say that expression e has type t

Typing Simple Let Expressions

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression
takes on the type of e2

Typing Simple Let Expressions

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression
takes on the type of e2

x has type int
for use inside the
let body

```
let x = 3 + 4 in

string_of_int x
```

overall expression
has type string

Another example…

What’s the type of the following expression?

```
let b = true in
let x = if b then 3 else 4 in
let y = x * 7 in
if not b then x else y
```

Another example...

What's the type of the following expression?

b has type bool in

```
let b = true in
let x = if b then 3 else 4 in
let y = x * 7 in
if not b then x else y
```

Another example...

What's the type of the following expression?

b has type bool in
x has type int in

```
let b = true in
let x = if b then 3 else 4 in
let y = x * 7 in
if not b then x else y
```

Another example...

What's the type of the following expression?

b has type bool in
x has type int in
y has type int in

```
let b = true in
let x = if b then 3 else 4 in
let y = x * 7 in
if not b then x else y
```

Another example...

What's the type of the following expression?

b has type bool in
x has type int in
y has type int in

```
let b = true in
let x = if b then 3 else 4 in
let y = x * 7 in
if not b then x else y
```

The overall type of the expression is the type of the "if-then-else", i.e., int

TUPLES

Tuples

- A tuple is a fixed, finite, ordered collection of values
- Some examples with their types:

```
(1, 2)                        : int * int

("hello", 7 + 3, true)        : string * int * bool

('a', ("hello", "goodbye"))   : char * (string * string)
```

Tuples

- To use a tuple, we extract its components
- General case:

```
let (id1, id2, …, idn) = e1 in e2
```

- An example:

```
let (x,y) = (2,4) in x + x + y
```

Tuples

- To use a tuple, we extract its components
- General case:

```
let (id1, id2, …, idn) = e1 in e2
```

- An example:

```
let (x,y) = (2,4) in x + x + y
-->  2 + 2 + 4
```

substitute!

Tuples

- To use a tuple, we extract its components
- General case:

```
let (id1, id2, …, idn) = e1 in e2
```

- An example:

```
let (x,y) = (2,4) in x + x + y
-->  2 + 2 + 4
-->  8
```

Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

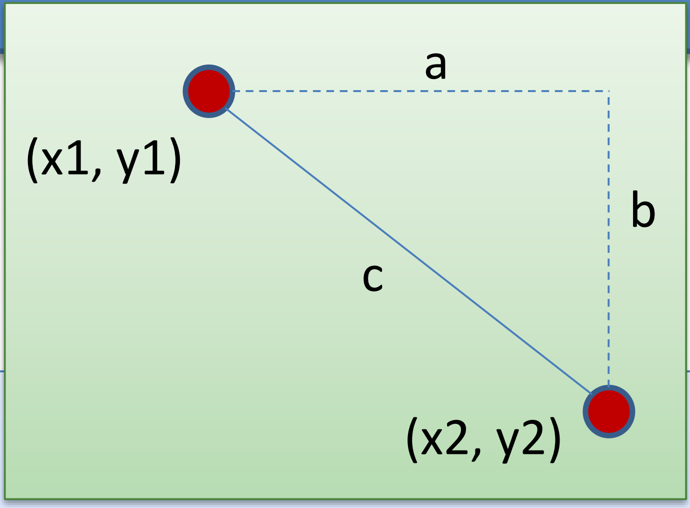
if e1 : t1 * t2 then
x1 : t1 and x2 : t2
inside the expression e2

```
let (x1,x2) = e1 in

e2
```

overall expression
takes on the type of e2

Distance between two points

$$c^2 = a^2 + b^2$$



Problem:

- A point is represented as a pair of floating point values.
- Write a function that takes in two points as arguments and returns the distance between them as a floating point number

Writing Functions Over Typed Data

Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)

Writing Functions Over Typed Data

Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   - *the argument types suggests how to do it*
5. Build new output values
   - *the result type suggests how you do it*

Writing Functions Over Typed Data

Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   - *the argument types suggests how to do it*
5. Build new output values
   - *the result type suggests how you do it*
6. Clean up by identifying repeated patterns
   - define and reuse helper functions
   - your code should be elegant and easy to read

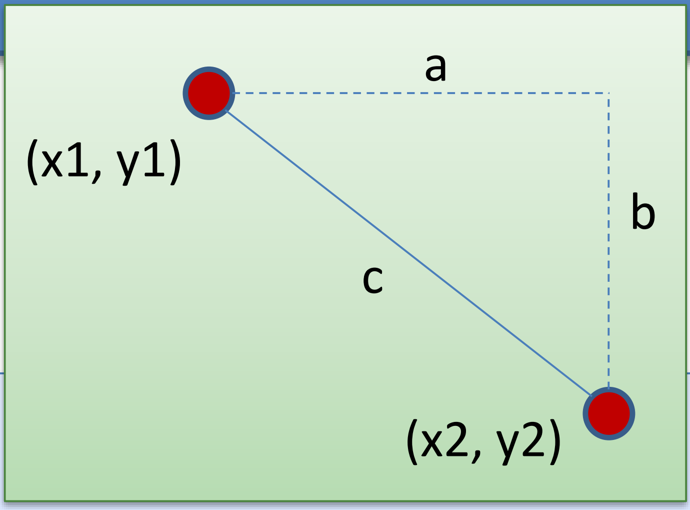
Writing Functions Over Typed Data

Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   - *the argument types suggests how to do it*
5. Build new output values
   - *the result type suggests how you do it*
6. Clean up by identifying repeated patterns
   - define and reuse helper functions
   - your code should be elegant and easy to read

Types help structure your thinking about how to write programs.

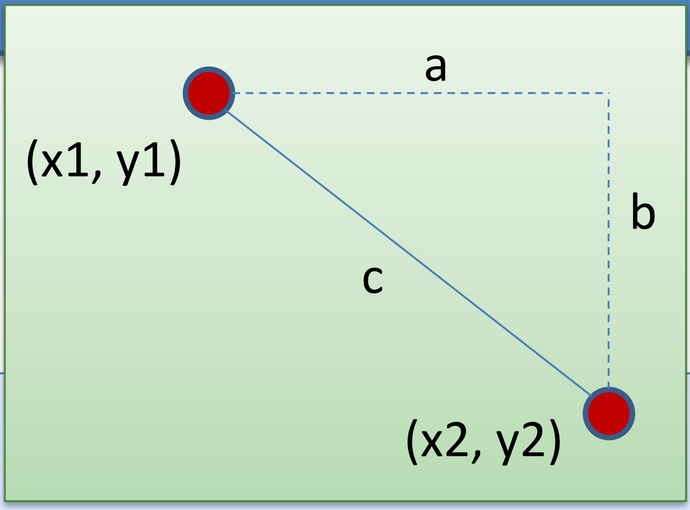
Distance between two points



a type abbreviation

```
type point = float * float
```

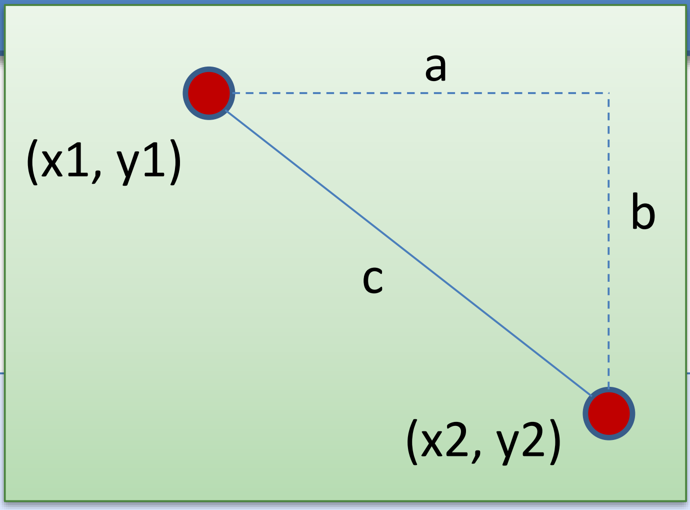
Distance between two points




```
type point = float * float

let distance (p1:point) (p2:point) : float =



;;
```

write down function name
argument names and types

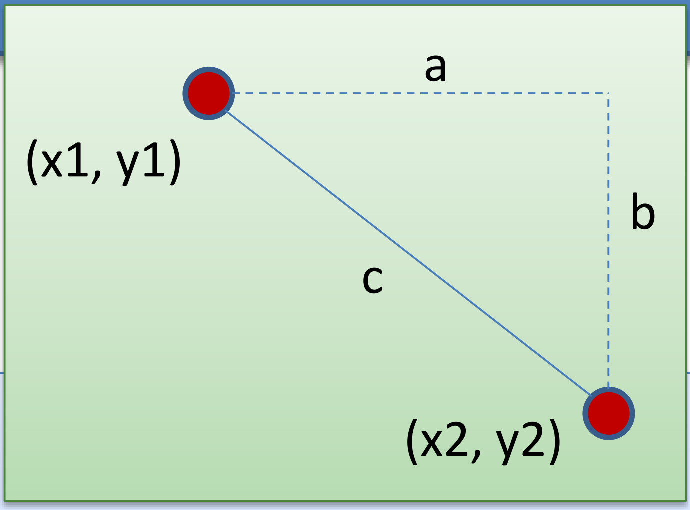
Distance between two points



examples

```
type point = float * float


(* distance (0.0,0.0) (0.0,1.0) == 1.0
 * distance (0.0,0.0) (1.0,1.0) == sqrt(1.0 + 1.0)
 *
 * from the picture:
 * distance (x1,y1) (x2,y2) == sqrt(a^2 + b^2)
 *)


let distance (p1:point) (p2:point) : float =
```

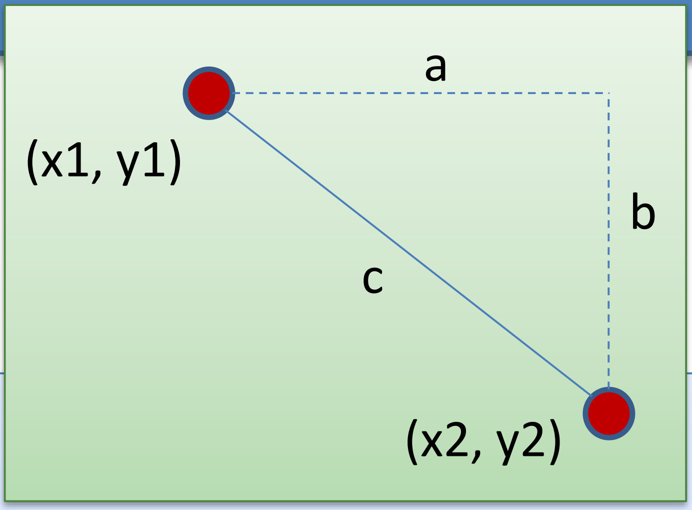
Distance between two points




```
type point = float * float

let distance (p1:point) (p2:point) : float =

  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  ...


;;
```

deconstruct
function inputs

Distance between two points

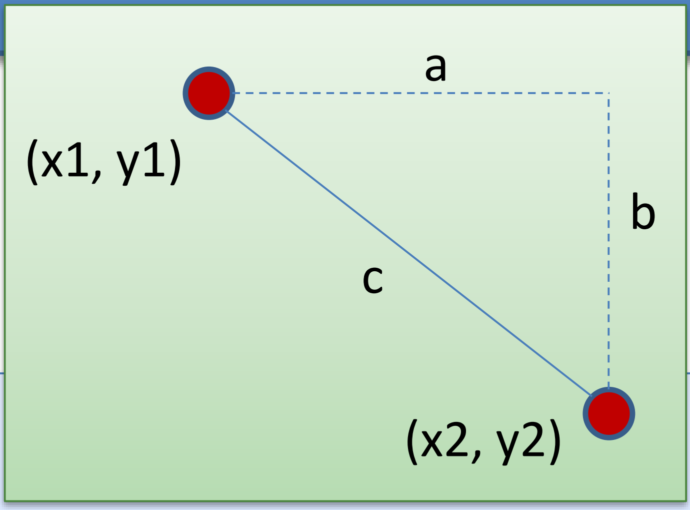



```
type point = float * float

let distance (p1:point) (p2:point) : float =

  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt ((x2 -. x1) *. (x2 -. x1) +.
        (y2 -. y1) *. (y2 -. y1))
;;
```

compute function results

notice operators on floats have a "." in them

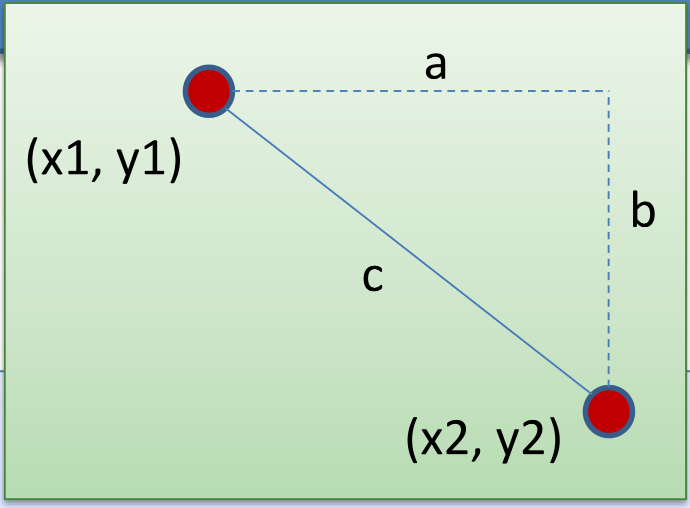
Distance between two points




```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1)) +.
        square (y2 -. y1))
;;
```

define helper functions to avoid repeated code

Distance between two points




```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;

let pt1 = (2.0,3.0);;
let pt2 = (0.0,1.0);;
let dist12 = distance pt1 pt2;;
```

testing

PART III:
LISTS, USER-DEFINED TYPES, POLYMORPHISM

Lists

- A list is a finite sequence of values, all of the same type
- Some examples with their types:

```
1 :: 2 :: 3 :: []             : int list

[1; 2; 3]                     : int list

“hello” :: “goodbye”          : string list

[true; false; false]          : bool list

[]                            : int list (or bool list, …)
```

Lists

- To use a list, we pattern-match it
- General case:

```
match e with
  | [] -> e1              (* nil case *)
  | hd :: tl -> e2        (* may depend on hd, tl *)
```

- An example:

```
let l = [1; 2; 3] in
match l with
  | [] -> 27
  | hd :: tl -> hd
```

Lists

- An example:

```
let l = [1; 2; 3] in
match l with
  | [] -> 27
  | hd :: tl -> hd
```

Lists

- An example:

```
     let l = [1; 2; 3] in
     match l with
       | [] -> 27
       | hd :: tl -> hd

-->  match [1; 2; 3] with
       | [] -> 27
       | hd :: tl -> hd
```

Lists

- An example:

```
    let l = [1; 2; 3] in
    match l with
      | [] -> 27
      | hd :: tl -> hd

--> match [1; 2; 3] with
      | [] -> 27
      | hd :: tl -> hd
```

hd = 1

tl = [2; 3]

Lists

- An example:

```
    let l = [1; 2; 3] in
    match l with
      | [] -> 27
      | hd :: tl -> hd

--> match [1; 2; 3] with
      | [] -> 27
      | hd :: tl -> hd

--> hd
```

hd = 1

tl = [2; 3]

Lists

- An example:

```
     let l = [1; 2; 3] in
     match l with
       | [] -> 27
       | hd :: tl -> hd

-->  match [1; 2; 3] with
       | [] -> 27
       | hd :: tl -> hd

-->  hd


-->  3
```

hd = 1

tl = [2; 3]

Lists under the hood

- The list type is built in to OCaml
- But that's no reason not to experiment with our own list data type
- In fact, O'Caml has a powerful system for defining all sorts of *user-defined data types*

Lists under the hood

- The list type is built in to OCaml
- But that's no reason not to experiment with our own list data type
- In fact, O'Caml has a powerful system for defining all sorts of *user-defined data types*

```
type intlist =
  | Nil                        (*the empty intlist*)
  | Cons of int * intlist      (*Cons: a pair of an int
                                 and an intlist*)
```

Lists under the hood

- The list type is built in to OCaml
- But that's no reason not to experiment with our own list data type
- In fact, O'Caml has a powerful system for defining all sorts of *user-defined data types*

`type` keyword declares a new user-defined type

```
type intlist =
   | Nil                         (*the empty intlist*)
   | Cons of int * intlist       (*Cons: a pair of an int
                                   and an intlist*)
```

`Cons` : int * intlist -> intlist

`Nil` : intlist

"constructors" of type intlist
But really, they're just functions
(`Nil` is nullary ☺)

Using User-Defined Lists

- To use a list, we pattern-match it
- General case:

```
match e with
   | Nil -> e1              (* nil case *)
   | Cons(hd, tl)-> e2      (* may depend on hd, tl *)
```

- An example:

```
let l = Cons(1, Cons(2, Cons(3, Nil))) in
match l with
   | Nil -> 27
   | Cons(hd, tl) -> hd
```

A more interesting function…

List Append

```
let rec app (l1 : intlist) (l2 : intlist) : intlist =
  match l1 with
    | Nil -> l2
    | Cons(x, l1’) -> Cons(x, app l1’ l2)
```

A more interesting function…

List Append

```
let rec app (l1 : intlist) (l2 : intlist) : intlist =
  match l1 with
    | Nil -> l2
    | Cons(x, l1') -> Cons(x, app l1' l2)
```

List Reverse

```
let rec rev (l : intlist) : intlist =
  match l with
    | Nil -> Nil
    | Cons(x, l') -> app (rev l') (Cons x Nil)
```

Polymorphism

Check out the types of `app` and `rev`:

- `app` : intlist -> intlist -> intlist
- `rev` : intlist -> intlist

Both functions operate over *intlists*, but they didn't really do anything with the contents of the intlists

Really, they just moved stuff around

The types we've given these functions *are a bit too precise*

Likewise the type intlist itself…

Polymorphic Lists

- The original intlist type

```
type intlist =
   | Nil                        (*the empty intlist*)
   | Cons of int * intlist      (*Cons: a pair of an int
                                  and an intlist*)
```

Polymorphic Lists

- The original intlist type

```
type intlist =
   | Nil                          (*the empty intlist*)
   | Cons of int * intlist        (*Cons: a pair of an int
                                     and an intlist*)
```

- A *polymorphic* version

```
type 'a list =
   | Nil                          (*the empty 'a list*)
   | Cons of 'a * 'a list         (*Cons: a pair of an 'a
                                     and an 'a list*)
```

Polymorphic Lists

- The original intlist type

```
type intlist =
   | Nil                       (*the empty intlist*)
   | Cons of int * intlist     (*Cons: a pair of an int
                                 and an intlist*)
```

- A *polymorphic* version

```
type 'a list =
   | Nil                       (*the empty 'a list*)
   | Cons of 'a * 'a list      (*Cons: a pair of an 'a
                                 and an 'a list*)
```

Type variables, can be instantiated to any type (e.g., int, bool, float, …)

Polymorphic Reverse, Append

Type variables

List Append

```
let rec app (l1 : 'a list) (l2 : 'a list) : 'a list =
  match l1 with
    | Nil -> l2
    | Cons(x, l1') -> Cons(x, app l1' l2)
```

List Reverse

```
let rec rev (l : 'a list) : 'a list =
  match l with
    | Nil -> Nil
    | Cons(x, l') -> app (rev l') (Cons x Nil)
```