

Recursion and Lists

Concepts of Programming Languages

Lecture 2

Practice Problem

None today . . .

Outline

- » Cover the **basic expressions** we need to start programming in OCaml, look at some examples
- » Discuss **lists** so we can write actually interesting programs
- » Talk about **tail recursion** and how that affects performance of OCaml programs

Recall: Functional vs. Imperative

OCaml is a **functional language**. This means a couple things:

- » No state (which means no loops!)
- » We don't think of a program as **describing a procedure**, but as **defining a value using an expression**

Recall: The Three Components

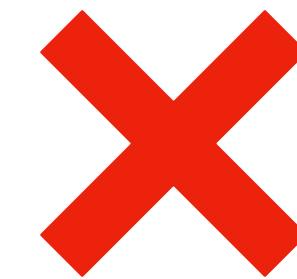
Recall: The Three Components

Syntax: What a *well-formed* program in your PL?

```
def f():  
    return 3
```



```
define f():  
    3 return
```



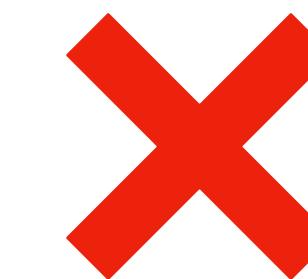
Recall: The Three Components

Syntax: What a *well-formed* program in your PL?

```
def f():  
    return 3
```



```
define f():  
    3 return
```



Type System (Static Semantics): What is a *valid* program in your PL?

```
x = 2 + 2
```



```
x = 2 + "two"
```



Recall: The Three Components

Syntax: What a *well-formed* program in your PL?

```
def f():  
    return 3
```



```
define f():  
    3 return
```

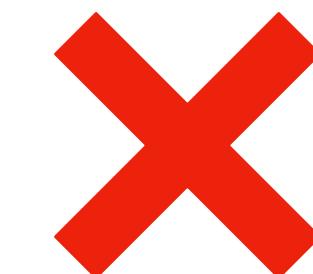


Type System (Static Semantics): What is a *valid* program in your PL?

```
x = 2 + 2
```



```
x = 2 + "two"
```



Semantics (Dynamic Semantics): What is the *output* of a (valid) program?

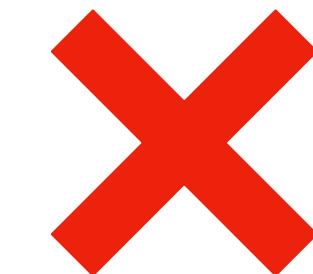
```
>>> 2 + 2
```

4



```
>>> 2 + 2
```

False



For every possible expression, we'll define the syntax rules, the typing rules, and the semantic rules

One Last Point: Building Interpreters

One Last Point: Building Interpreters

PL is math but we still like to use PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

One Last Point: Building Interpreters

PL is math but we still like to use PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

» **Syntax** is implemented by a **parser**

`parse : string -> expr`

One Last Point: Building Interpreters

PL is math but we still like to use PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

» **Syntax** is implemented by a **parser**

`parse : string -> expr`

» **Type system** is implemented by a **type checker**

`type_check : expr -> bool (* valid or not *)`

One Last Point: Building Interpreters

PL is math but we still like to use PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

» **Syntax** is implemented by a **parser**

`parse : string -> expr`

» **Type system** is implemented by a **type checker**

`type_check : expr -> bool (* valid or not *)`

» **(Dynamic) semantics** is implemented by an **evaluator**

`eval : expr -> value`

Basic Expressions

Basic Expressions

- » Literals
- » Let-expressions (local variables)
- » If-expressions
- » Functions
- » Applications

Basic Expressions

» **Literals**

» Let-expressions (local variables)

» If-expressions

» Functions

» Applications

Primitive Types and Literals

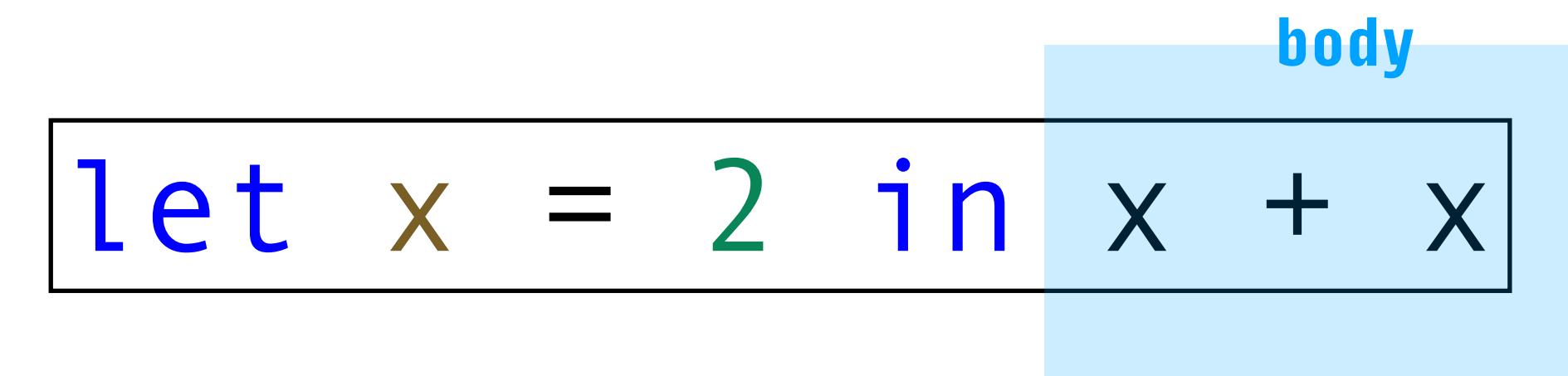
OCaml has a collection of standard literals and types

Type	Literals	Operators
int	0, -2, 13, -023	+, -, *, /, mod
float	3., -1.01	+., -., *. , /.
bool	true, false	&&, , not
char	'b', 'c'	
string	"word", "@*&#"	^

Basic Expressions

- » Literals
- » Let-expressions (local variables)
- » If-expressions
- » Functions
- » Applications

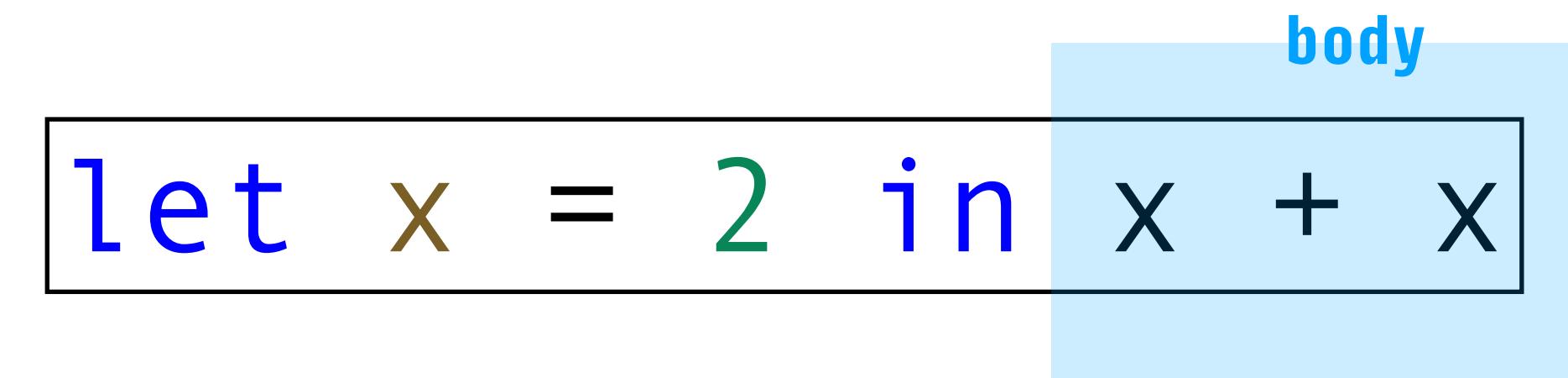
Let-Expressions (Informal)



A diagram illustrating a let-expression. The code is shown in a black-bordered box: `let x = 2 in x + x`. A light blue rectangular box, labeled "body" in blue text at its top right, covers the entire `x + x` portion of the expression.

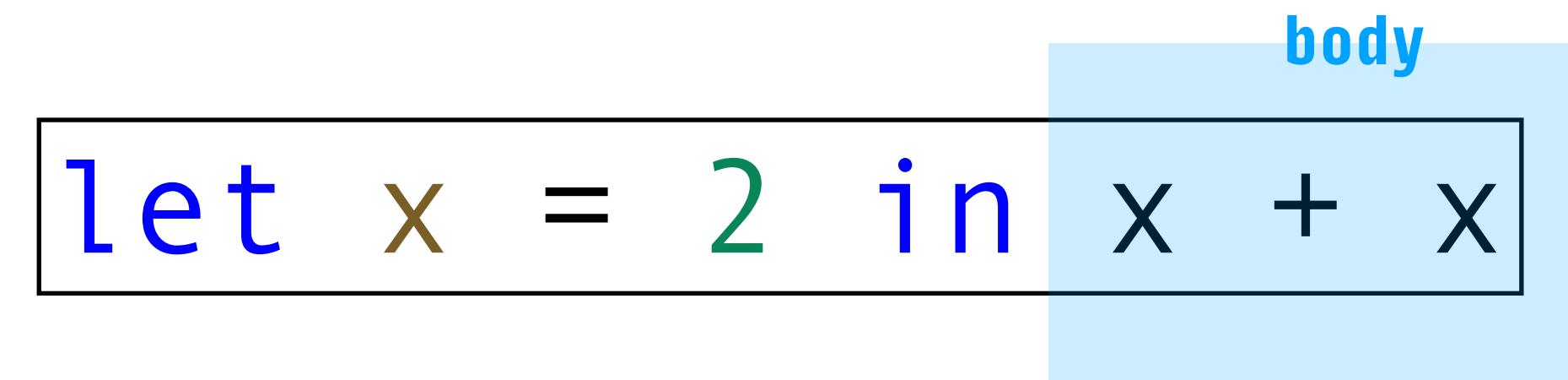
```
let x = 2 in x + x
```

Let-Expressions (Informal)



syntax: let VARIABLE = EXPRESSION in BODY

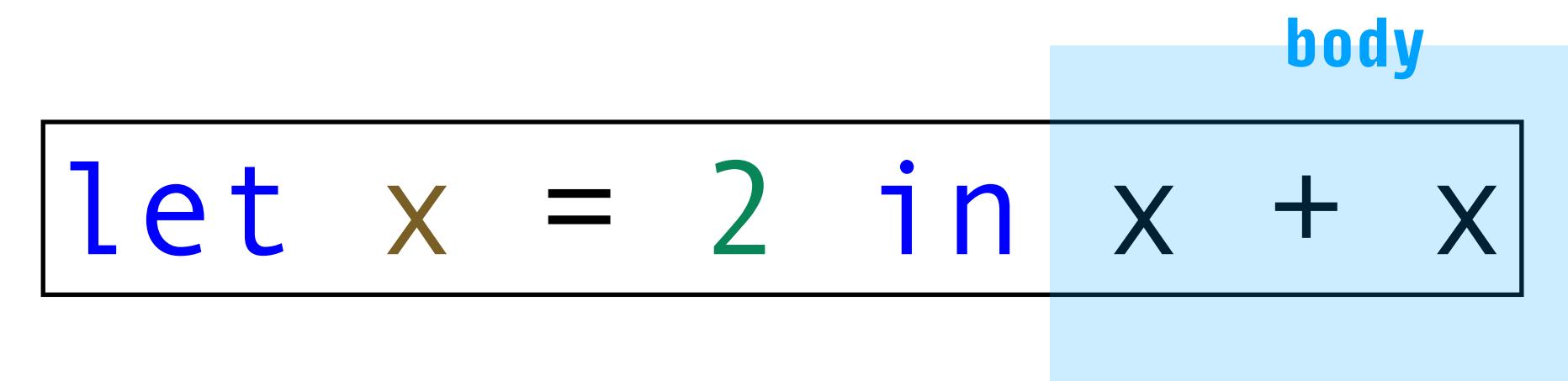
Let-Expressions (Informal)



syntax: let VARIABLE = EXPRESSION in BODY

typing: the type is the same as that of BODY *given BODY is well-typed after substituting the VARIABLE in BODY*

Let-Expressions (Informal)



syntax: let VARIABLE = EXPRESSION in BODY

typing: the type is the same as that of BODY *given BODY is well-typed after substituting the VARIABLE in BODY*

semantics: the is the same as the value of BODY *after substituting the VARIABLE in BODY*

Example: Ill-Typed Let-Expression

```
let x = 2 in "two" <> x
```

An ill-typed expression will throw a type error when you type it into utop

Note that the body of a let-expression may be ill-typed *depending on the value assigned to its variable*

A Note on Substitution

```
let x = 2 in x + x
```



```
2 + 2
```

A Note on Substitution

```
let x = 2 in x + x
```



```
2 + 2
```

Formally, we write $[v/x]e$ to mean "substitute v for x in e ", e.g., $[3/x](x+x)$ is the same as $3+3$

A Note on Substitution

```
let x = 2 in x + x
```



```
2 + 2
```

Formally, we write $[v/x]e$ to mean "substitute v for x in e ", e.g., $[3/x](x+x)$ is the same as $3+3$

Intuitively, substitution is simple: **replace the variable**

A Note on Substitution

```
let x = 2 in x + x —————→ 2 + 2
```

Formally, we write $[v/x]e$ to mean "substitute v for x in e ", e.g., $[3/x](x+x)$ is the same as $3+3$

Intuitively, substitution is simple: **replace the variable**

Turns out, this is **very hard** to do correctly, it's subtle and a source of a lot of mistakes in PL implementations

Recall: Anatomy of an OCaml Program

```
let x = 3

let y = "string"

(* function definition *)
let square x = x * x

(* recursive function definition *)
let rec f x = if x = 0 then 0 else x + f (x - 1)

(* We can't just print , we assign to wildcard *)
let _ = print_endline("Hello world")
```

An OCaml Program consists of top-level let-expressions, i.e., it is a **collection of named expressions**

OCaml Programs are Expressions

```
let x = 3 in  
  
let y = "string" in  
  
(* function definition *)  
let square x = x * x in  
  
(* recursive function definition *)  
let rec f x = if x = 0 then 0 else x + f (x - 1) in  
  
(* We can't just print , we assign to wildcard *)  
let _ = print_endline("Hello world") in  
  
()
```

This sequence of top-level let expressions is really shorthand for a **collection of nested local variables**

(This is a lie, but its a useful one for now)

Basic Expressions

- » Literals
- » Let-expressions (local variables)
- » **If-expressions**
- » Functions
- » Applications

If-Expressions

```
let abs x = if x > 0 then x else -x
```

Note: OCaml is whitespace agnostic!

If-Expressions

```
let abs x = if x > 0 then x else -x
```

Note: OCaml is whitespace agnostic!

OCaml has expressions for conditional reasoning

If-Expressions

```
let abs x = if x > 0 then x else -x
```

Note: OCaml is whitespace agnostic!

OCaml has expressions for conditional reasoning

Note: The **else** case is *required* and the **then** and **else** cases must be the *same type* (why?)

If-Expressions

```
let foo x =  
  if x < 0 then  
    "negative"  
  else if x = 0 then  
    "zero"  
  else  
    "positive"
```

Answer: Remember, all we have is expressions. So every if-expression must have a value and a type (and therefore, an **else** case of the same type)

We can do **else if** just by nesting if-expressions! (neat)

Aside: If-Expressions in Python

```
if x < 0:  
    return -1  
else:  
    return 1
```

if-stmt (Python)

```
return (-1 if x < 0 else 1)
```

if-expr (Python)

If-*statements* in Python are different from if-expressions,
but **both are available**

Statements don't have a value, expressions do

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE–CASE else FALSE–CASE

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE-CASE else FALSE-CASE

Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE-CASE else FALSE-CASE

Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

Semantics: If CONDITION holds, then we get the TRUE-CASE, otherwise we get the FALSE-CASE

Basic Expressions

- » Literals
- » Let-expressions (local variables)
- » If-expressions
- » **Functions**
- » Applications

Functions

```
let f x y z = x + y + z  
let f (x : int) (y : int) (z : int) : int = x + y + z
```

There are a couple ways of defining functions in OCaml

Note that let-expression can take arguments. *How should we interpret this? If everything is an expression?*

Anonymous Functions

```
let f = fun x -> fun y -> fun z -> x + y + z
```

Answer: It must be that **functions are expressions as well!**

In OCaml, we can define *anonymous* functions, which are just **functions without names**

You should think of:

```
let f x y z = x + y + z
```

as shorthand for the above

Aside: Type Annotations

```
let rec fact (n : int) : int =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

Aside: Type Annotations

```
let rec fact (n : int) : int =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

OCaml has type inference which means we rarely have to *specify* the types of expression in our program

Aside: Type Annotations

```
let rec fact (n : int) : int =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

OCaml has type inference which means we rarely have to *specify* the types of expression in our program

That said, you **should** include type annotations, especially at the beginning, because they're useful for *documentation* and for *code clarity*

Aside: Anonymous Functions in Python

```
Lambda x: x + 1
```

Python

```
fun x -> x + 1
```

OCaml

$$\lambda x. x + 1$$

Math

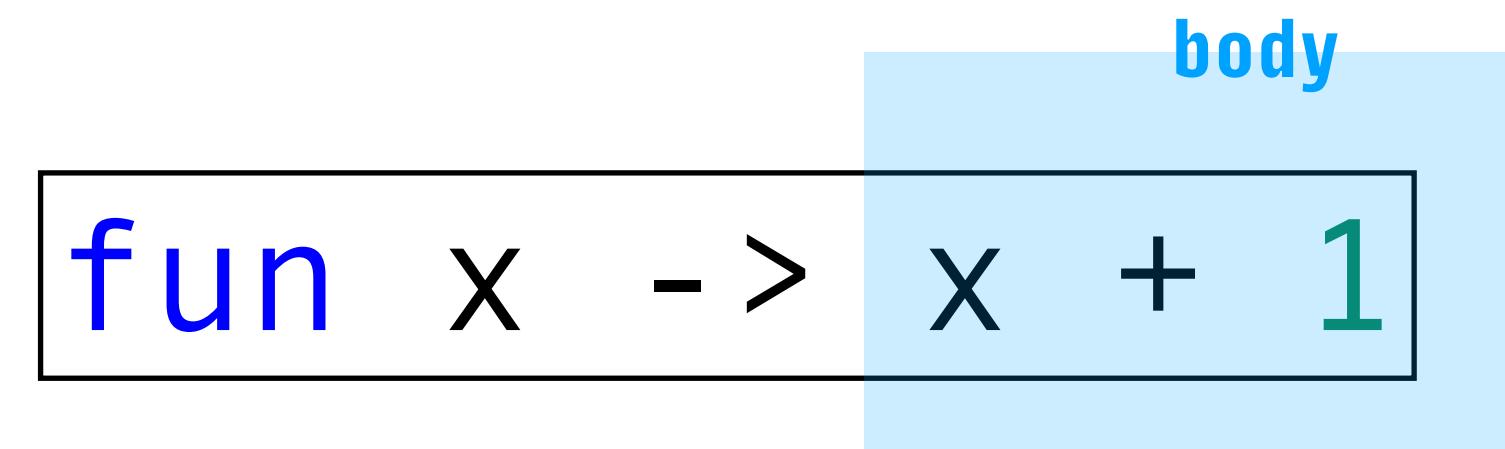
There are also anonymous functions in Python!

They're called *lambdas*, based on the **lambda calculus**, a mathematical formulation of a functional PL that dates back to the 1930s, invented by **Alonzo Church**

(You'll find a lot of functional ideas hidden in languages like Python)



Functions (Informal)



Syntax: fun VAR-NAME -> EXPR

Typing: the type of a function is $T_1 \rightarrow T_2$ where T_1 is the type of the input and T_2 is the type of the output

Semantics: A function will evaluate to special *function value* (printed as <fun> by utop)

Important: Curried Functions

```
let f = fun x -> fun y -> fun z -> x + y + z
```

The only kind of function we have is *single argument*

This seems restrictive, but ultimately it doesn't affect us at all

We can *simulate* multi-argument functions with nested functions. This is called **Currying** after Haskell Curry

Important: Curried Functions

```
let f = fun x -> fun y -> fun z -> x + y + z
```

We should think of the above function as something which takes an input and returns **another function**

In other words, we *partially apply* the function

Basic Expressions

- » Literals
- » Let-expressions (local variables)
- » If-expressions
- » Functions
- » Applications

Application

```
(fun x -> fun y -> x + y + 1) 3 2
```

Application is done by *juxtaposition* which means we put the arguments next to the function

Application is *left-associative*, which means we pass arguments from left to right

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Typing: If FUNCTION-EXPR is of type $T_1 \rightarrow T_2$,
and ARG-EXPR is of type T_1 , then the type is T_2

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Typing: If FUNCTION-EXPR is of type $T_1 \rightarrow T_2$,
and ARG-EXPR is of type T_1 , then the type is T_2

Semantics: Substitute the value of ARG-EXPR into
the body of FUNCTION-EXPR and evaluate that

Application (Example)

Application (Example)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 *is the same as*

(fun x -> (fun y -> x + y + 1)) 3 2

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 *is the same as*

(fun x -> (fun y -> x + y + 1)) 3 2 *is the same as*

((fun x -> (fun y -> x + y + 1)) 3) 2

Application (Example)

$(\text{fun } x \rightarrow \text{fun } y \rightarrow x + y + 1) \ 3 \ 2$ is the same as

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$((\text{fun } \boxed{x} \rightarrow (\text{fun } y \rightarrow \boxed{x} + y + 1)) \boxed{3}) \ 2$

Application (Example)

$(\text{fun } x \rightarrow \text{fun } y \rightarrow x + y + 1) \ 3 \ 2$ is the same as

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3) \ 2$ evaluates to

$(\text{fun } y \rightarrow 3 + y + 1) \ 2$

Application (Example)

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3) \ 2$ evaluates to

$(\text{fun } y \rightarrow 3 + y + 1) \ 2$

Application (Example)

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3) \ 2$ evaluates to

$(\text{fun } y \rightarrow 3 + y + 1) \ 2$ evaluates to

$3 + 2 + 1$

Application (Example)

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3) \ 2$ evaluates to

$(\text{fun } y \rightarrow 3 + y + 1) \ 2$ evaluates to

$3 + 2 + 1$ is the same as

$(3 + 2) + 1$

Application (Example)

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$(\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3 \ 2$ is the same as

$((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y + 1)) \ 3) \ 2$ evaluates to

$(\text{fun } y \rightarrow 3 + y + 1) \ 2$ evaluates to

$3 + 2 + 1$ is the same as

$(3 + 2) + 1$ evaluates to

$5 + 1$

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 is the same as

(fun x -> (fun y -> x + y + 1)) 3 2 is the same as

((fun x -> (fun y -> x + y + 1)) 3) 2 evaluates to

(fun y -> 3 + y + 1) 2 evaluates to

3 + 2 + 1 is the same as

(3 + 2) + 1 evaluates to

5 + 1 evaluates to

demo
(extended example)

Lists

What is a list?

```
let _ = 1 :: 2 :: 3 :: []
let _ = 1 :: (2 :: (3 :: []))
let _ = [1; 2; 3]
```

A list is an ordered *variable-length homogeneous* collection of data

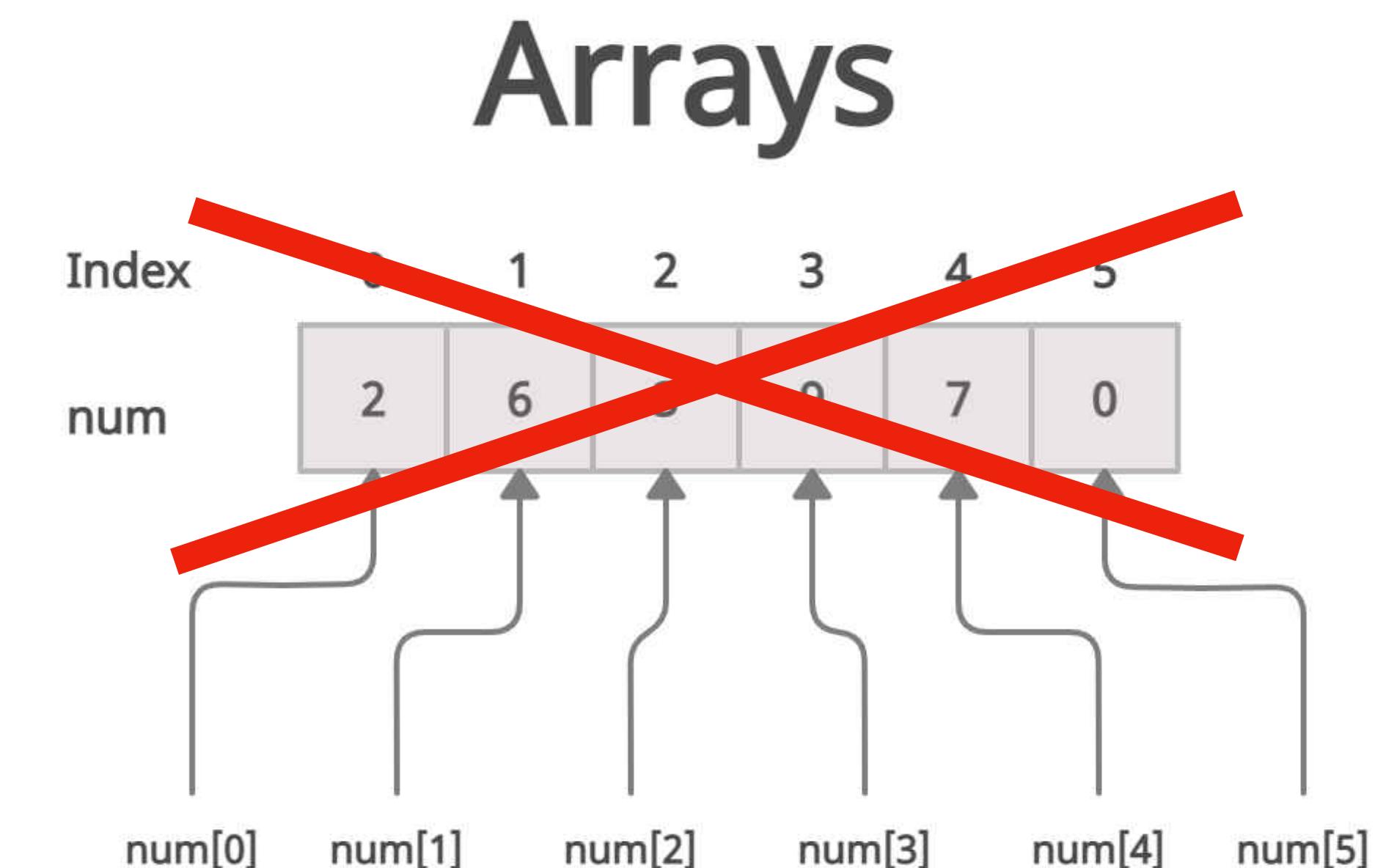
Many important operations on data can be represented as operations on lists (e.g., updating all users in a database)

What is a list not?

A list is *not* an array. We don't have constant-time indexing

A list is *not* mutable. **No data structures in FP are mutable**

(You should think of a list structurally as more like a linked list, sort of)



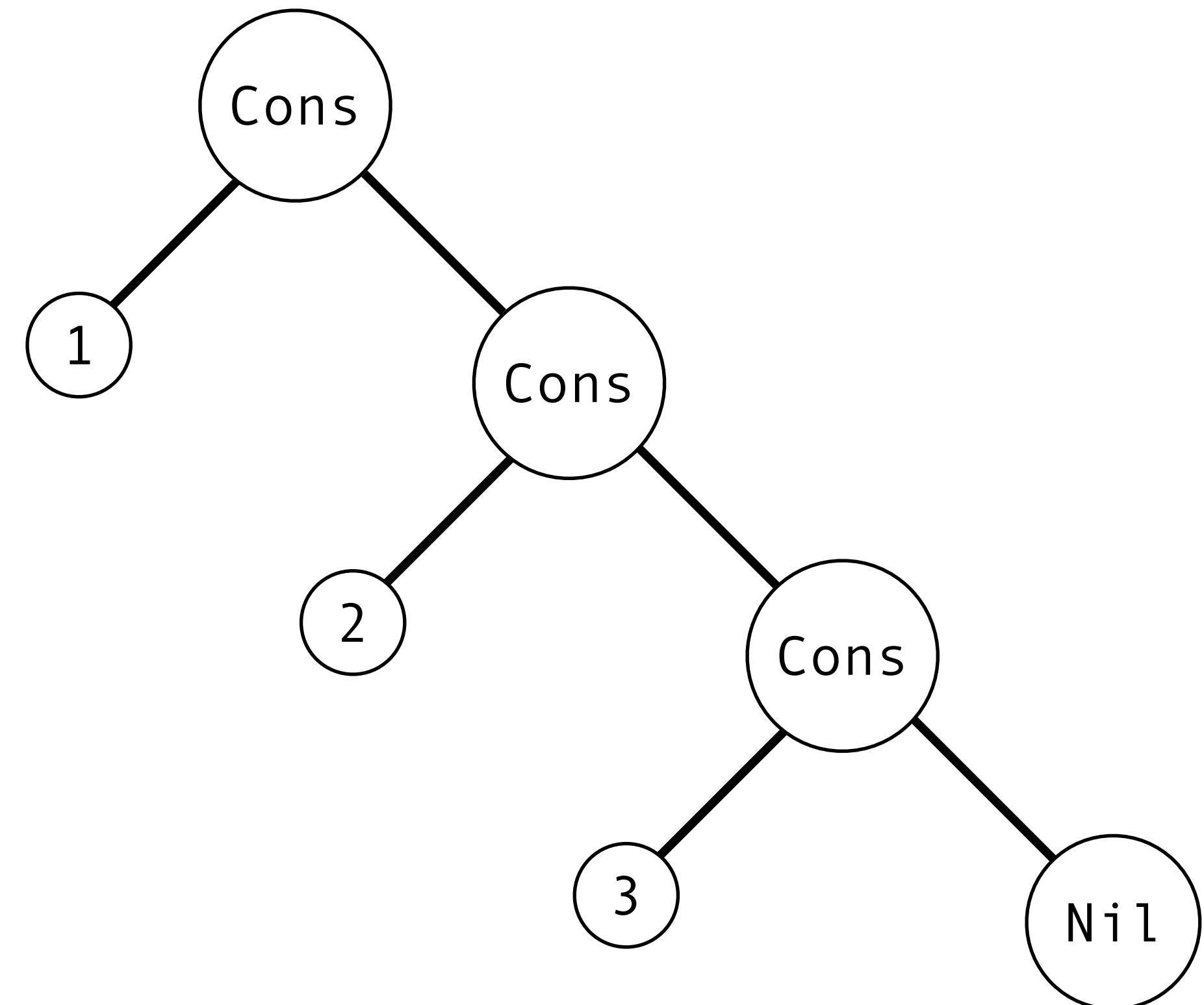
The Picture

We can think of the list

```
1 :: 2 :: 3 :: []
```

as a leaning tree with data
as leaves

(this will generalize to
other *algebraic* data types)



List Syntax (Informally)

```
let _ = 1 :: 2 :: 3 :: []
let _ = 1 :: (2 :: (3 :: []))
let _ = [1; 2; 3]
```

List Syntax (Informally)

```
let _ = 1 :: 2 :: 3 :: []
let _ = 1 :: (2 :: (3 :: []))
let _ = [1; 2; 3]
```

`[]` stands for the empty list (a.k.a. `nil`), the list with no elements

List Syntax (Informally)

```
let _ = 1 :: 2 :: 3 :: []
let _ = 1 :: (2 :: (3 :: []))
let _ = [1; 2; 3]
```

`[]` stands for the empty list (a.k.a. **nil**), the list with no elements

`x :: xs` stands for the list `xs` with `x` prepended to it. The symbol `::` is pronounced "cons" and is a *right associative* operator

List Syntax (Informally)

```
let _ = 1 :: 2 :: 3 :: []
let _ = 1 :: (2 :: (3 :: []))
let _ = [1; 2; 3]
```

`[]` stands for the empty list (a.k.a. **nil**), the list with no elements

`x :: xs` stands for the list `xs` with `x` prepended to it. The symbol `::` is pronounced "cons" and is a *right associative* operator

`[x1; x2; ...; xn]` is a list literal. It's shorthand for a list of a known length

Example

*Construct a function **generate** which, given integers n , returns a list consisting of the first n positive integers*

Lists Semantics (Informally)

$$[2 + 3; 4 * 12; 2 - 1] \downarrow [5; 48; 1]$$

We evaluate the list $[e_1; e_2; \dots; e_k]$ by evaluating each element of the list (from right to left)

Destructuring Lists

```
match l with
| [] -> (* something *)
| x :: xs -> (* something else *)
| ... (* other patterns?? *)
```

As with any type in OCaml, we can use **pattern matching** to destruct lists

With pattern matching, we describe the value we want based on the *shape* of the list we're matching on

Example

*Implement the function **double** where **double l** is the same as the list **l** but with every element doubled*

Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

All data structures in FP are immutable

Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

All data structures in FP are immutable

If we want to "update" a list, we have to produce an *entirely new* list

Lists are Immutable

```
val remove_all_negatives : int list -> int list
```

All data structures in FP are immutable

If we want to "update" a list, we have to produce an *entirely new* list

In reality the data is not literally duplicated, there are optimizations which allow for shared data

Practice Problem

Implement the function

remove_all_negatives : int list -> int list

*where **remove_all_negative** l is the same as the list l but with all negative numbers removed*

Deep Pattern Matching

```
match <expr> with
| [] -> <expr>
| [h1; h2] -> <expr>
| h1::h2::t -> <expr>
| h::t -> <expr>
| . . . . .
```

Pattern matching is very general. We can match on more complex patterns than just empty and nonempty

Example

Implement the function

delete_every_other : int list -> int list

*such that **delete_every_other l** is the first, third, fifth, . . . , and so on elements of **l***

A Note on Polymorphism

```
let rec length l =  
  match l with  
  | [] -> 0  
  | x :: xs -> 1 + length xs
```

What is the type of the length function?

Does this function depend on the values in the list?

The List Type

[1;2;3]

int list

["1";"2";"3"]

string list

[[1;1];[2;2];[3;3]]

int list list

The List Type

[1;2;3]

int list

["1";"2";"3"]

string list

[[1;1];[2;2];[3;3]]

int list list

The list type is an example of a **parametrized** type

The List Type

[1;2;3]

int list

["1";"2";"3"]

string list

[[1;1];[2;2];[3;3]]

int list list

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

The List Type

[1;2;3]

int list

["1";"2";"3"]

string list

[[1;1];[2;2];[3;3]]

int list list

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

For this, we need *type parameters* to stand for *any* type:

The List Type

[1;2;3]

int list

["1";"2";"3"]

string list

[[1;1];[2;2];[3;3]]

int list list

The list type is an example of a **parametrized** type

A function on lists is *polymorphic* (with respect to the list parameter) if it can be apply to a list parametrized by *any* type

For this, we need *type parameters* to stand for *any* type:

'a, 'b, 'c, ...

Not all functions can be polymorphic

```
let rec sum l =  
  match l with  
  | [] -> 0  
  | x :: xs -> x + sum xs
```

Not all functions can be polymorphic

```
let rec sum l =  
  match l with  
  | [] -> 0  
  | x :: xs -> x + sum xs
```

Can this function be applied to a list parametrized by any type?

Not all functions can be polymorphic

```
let rec sum l =  
  match l with  
  | [] -> 0  
  | x :: xs -> x + sum xs
```

Can this function be applied to a list parametrized by any type?

Answer: No, it can only be applied to **int lists**

Not all functions can be polymorphic

```
let rec sum l =  
  match l with  
  | [] -> 0  
  | x :: xs -> x + sum xs
```

Can this function be applied to a list parametrized by any type?

Answer: No, it can only be applied to **int lists**

OCaml's type inference is good at "guessing" when functions are polymorphic

Practice Problem

Implement the function

reverse : 'a list -> 'a list

*such that reverse l is the same as l but in
reverse order*

Tail Recursion

demo
(even the wrong way)

Tail Recursion

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

not tail recursive

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
  in loop 1 n
```

tail recursive

A recursive function is **tail recursive** if it does not perform any computations on the result of a recursive call

Why do we care?

Why do we care?

Recursive functions are *expensive* with respect to the call-stack

Why do we care?

Recursive functions are *expensive* with respect to the call-stack

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames

Why do we care?

Recursive functions are *expensive* with respect to the call-stack

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames

Tail-recursive functions "behave iteratively"

The Picture

fact 5

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

The Picture

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

fact 5

fact 4

The Picture

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

fact 5

fact 4

fact 3

The Picture

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

fact 5

fact 4

fact 3

fact 2

The Picture

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

fact 5

fact 4

fact 3

fact 2

fact 1

The Picture

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

fact 5

fact 4

fact 3

fact 2

fact 1

fact 0

The Picture

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

fact 5

fact 4

fact 3

fact 2

fact 1

fact 0

$\Rightarrow 1$

The Picture

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

fact 5

fact 4

fact 3

fact 2

fact 1

$$\Rightarrow 1 * 1 = 1$$

The Picture

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

fact 5

fact 4

fact 3

fact 2

$$\Rightarrow 2 * 1 = 2$$

The Picture

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

fact 5

fact 4

fact 3

$$\Rightarrow 3 * 2 = 6$$

The Picture

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

fact 5

fact 4

$$\Rightarrow 4 * 6 = 24$$

The Picture

fact 5

$$\Rightarrow 5 * 24 = 120$$

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

The Picture

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

fact 5

$$\Rightarrow 5 * 24 = 120$$

fact 4

$$\Rightarrow 4 * 6 = 24$$

fact 3

$$\Rightarrow 3 * 2 = 6$$

fact 2

$$\Rightarrow 2 * 1 = 2$$

fact 1

$$\Rightarrow 1 * 1 = 1$$

fact 0

$$\Rightarrow 1$$

1 frame per
recursive call

The Picture

```
loop 1 5
```

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
  in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2

fact 120 1

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2

fact 120 1

fact 120 0

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2

fact 120 1

fact 120 0

⇒ 120

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2

fact 120 1
⇒ 120

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2
⇒ 120

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

$\Rightarrow 120$

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

loop 5 4
⇒ 120

The Picture

```
loop 1 5  
⇒ 120
```

```
let fact n =  
let rec loop acc n =  
  if n <= 0  
  then acc  
  else loop (n * acc) (n - 1)  
in loop 1 n
```

The Picture

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
      then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

⇒ 120

loop 5 4

⇒ 120

loop 20 3

⇒ 120

fact 60 2

⇒ 120

fact 120 1

⇒ 120

fact 120 0

⇒ 120

1 frame per
recursive call

BUT THE VALUE
DOESN'T CHANGE
ON IT'S WAY UP
THE CALL STACK

The Picture (Optimized)

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 1 5

The Picture (Optimized)

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 5 4

The Picture (Optimized)

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 20 3

The Picture (Optimized)

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

```
loop 120 1
```

The Picture (Optimized)

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

loop 120 0

The Picture (Optimized)

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

```
loop 120 0  
⇒ 120
```

The Picture (Optimized)

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
in loop 1 n
```

```
loop 120 0  
⇒ 120
```

1 frame
for every
recursive
call

Tail Position

```
let rec fact n =  
  if n <= 0  
  then 1 computation after the recursive call  
  else n * fact (n - 1)  
  
not tail recursive
```

```
let fact n =  
  let rec loop acc n =  
    if n <= 0  
    then acc  
    else loop (n * acc) (n - 1)  
  in loop 1 n  
  
tail recursive
```

Tail-call optimizations apply to functions whose recursive calls are in **tail position**

Intuition: A call is in tail position if there is no computation *after* the recursive call

Aside: Tail Position More Formally

```
let rec f x1 x2 ... xk = e
```

A recursive call `f e1 e2 ... ek` is in tail position in `e` if:

- » it does not appear in `e`, or `e` is the recursive call itself
- » `e = if e1 then e2 else e3` and the call does not appear in `e1` and it is in tail position in `e2` and `e3`
- » `e` is a **match-expression** and the call is in tail position in every branch, and does not appear in the matched expression
- » `e = let x = e1 in e2` and the call does not appear in the `e1` and it is in tail position in `e2`

Aside: Tail Position More Formally

```
let rec f x1 x2 ... xk = e
```

A recursive call $f \ e_1 \ e_2 \ \dots \ e_k$ ^{*} is in tail position in e if:

- » it does not appear in e , or e is the recursive call itself
- » $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$ and the call does not appear in e_1 and it is in tail position in e_2 and e_3
- » e is a **match-expression** and the call is in tail position in every branch, and does not appear in the matched expression
- » $e = \text{let } x = e_1 \text{ in } e_2$ and the call does not appear in the e_1 and it is in tail position in e_2

* f cannot appear in $e_1 \dots e_k$

Tail Recursion and Lists

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

Tail Recursion and Lists

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

We need to take care with tail-recursion and lists

Tail Recursion and Lists

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

We need to take care with tail-recursion and lists

Does the above program concatenate two lists?

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

append [1;2;3] [4;5;6]

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [1;2;3] [4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

```
match [1;2;3] with
| [] -> [4;5;6]
| x :: xs -> loop xs (x :: [4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match 1 :: [2;3] with
| [] -> [4;5;6]
| x :: xs -> loop xs (x :: [4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [2;3] (1 :: [4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [2;3] [1;4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match [2;3] with
| [] -> [1;4;5;6]
| x :: xs -> loop xs (x :: [1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
match 2 :: [3] with
| [] -> [1;4;5;6]
| x :: xs -> loop xs (x :: [1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

```
loop [3] (2 :: [1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

```
loop [3] [2;1;4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

```
match [3] with
| [] -> [2;1;4;5;6]
| x :: xs -> loop xs (x :: [2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

```
match 3 :: [] with
| [] -> [2;1;4;5;6]
| x :: xs -> loop xs (x :: [2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [] (3 :: [2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

```
loop [] [3;2;1;4;5;6]
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
in loop l r
```

```
match [] with
| [] -> [3;2;1;4;5;6]
| x :: xs -> loop xs (x :: [3;2;1;4;5;6])
```

```
let append l r =
  let rec loop acc l =
    match l with
    | [] -> acc
    | x :: xs -> loop xs (x :: acc)
  in loop l r
```

[3;2;1;4;5;6]

whoops!

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

We need to take care with tail-recursion and lists

Does the above program concatenate two lists?

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
should be (List.rev l)
```

We need to take care with tail-recursion and lists

Does the above program concatenate two lists?

Accumulators

```
let fact n =  
    let rec loop acc n =  
        if n <= 0  
        then acc  
        else loop (n * acc) (n - 1)  
in loop 1 n
```

Our accumulator pattern is almost always tail recursive

Code Example

Implement the function

reverse : 'a list -> 'a list

The implementation must be tail recursive

Summary

OCaml is a language of **expressions**, everything is an expression

Lists are used to process collections of homogeneous data

We can use **tail-recursion** to make our implementations more memory efficient, but we have to be careful when working with lists