Programming in OCaml

Principles of Programming Languages

CSE 526

Compiled at 09:43 on 2020/04/13

Functional Programming

- Programs are viewed as functions transforming input to output
- Complex transformations are achieved by *composing* simpler functions (i.e. applying functions to results of other functions)

Purely Functional Languages: Values given to "variables" do not change when a program is evaluated

- "Variables" are names for values, not names for storage locations.
- Functions have *referential transparency*:
  - Value of a function depends solely on the values of its arguments
  - Functions do not have *side effects*.
  - Order of evaluation of arguments does not affect the value of a function's output.

Features of Functional Programming Languages

- Support for complex (recursive) data types
  ... with automatic memory management (e.g. garbage collection)
- Functions themselves may be treated as values
  - *Higher-order functions*: Functions that functions as arguments.
  - *Functions as first-class values*: no arbitrary restrictions that distinguish functions from other data types (e.g. `int`)

History of Functional Programming

- LISP ('60)
- Scheme ('80s): a dialect of LISP; more uniform treatment of functions
- ML ('80s): Strong typing and *type inference*
  - Standard ML (SML, SML/NJ: '90s)
  - Categorical Abstract Machine Language (CAML, CAML Light, O'CAML: late '90s)
- Haskell, Gofer, HUGS, ... (late '90s): "Lazy" functional programming

ML

Developed initially as a "meta language" for a theorem proving system (*Logic of Computable Functions*)

- The two main dialects, SML and CAML, have many features in common:
  - data type definition, type inference, interactive top-level, ...
- SML and CAML have different syntax for expressing the same things. For example:
  - In SML: variables are defined using `val` and functions using `fun`
  - In CAML: both variables and functions defined using *equations*.
- Both have multiple implementations (Moscow SML, SML/NJ; CAML, OCAML) with slightly different usage directives and module systems.

OCAML

- CAML with "object-oriented" features.
- Compiler and run-time system that makes OCAML programs run with performance comparable imperative programs!
- A complete development environment including libraries building UIs, networking (sockets), etc.
- *We will mainly use the non-oo part of OCAML*
  - Standard ML (SML) has more familiar syntax.
  - CAML has better library and runtime support and has been used in more "real" systems.

The OCAML System

- OCAML interactive toplevel
  - Invocation:
    - UNIX: Run `ocaml` from command line
    - Windows: Run `ocaml.exe` from Command window or launch `ocamlwin.exe` from windows explorer.
  - OCAML prompts with "#"
  - User can enter new function/value definitions, evaluate expressions, or issue OCAML directives at the prompt.
  - Control-D to exit OCAML
- OCAML compiler:
  - `ocamlc` to compile OCAML programs to object bytecode.
  - `ocamlopt` to compile OCAML programs to native code.

Learning OCAML

- We will use OCAML interactive toplevel throughout for examples.
- What we type in can be entered into a file (i.e. made into a "program") and executed.
- Read
  1. Tutorials at `https://ocaml.org`
  2. OCAML textbook: `https://realworldocaml.org/`
  3. OCAML manual:`http://caml.inria.fr/pub/docs/manual-ocaml/`

OCaml Expressions

- Syntax: ⟨*expression*⟩ `;;`
- Two semicolons indicate the end of expression
- Example:

User Input	OCAML's Response
`2 * 3;;`	`- : int = 6`

OCAML's response:

'`-`'	:	The last value entered
'`:`'	:	is of type
'`int`'	:	integer
'`=`'	:	and the value is
'`6`'	:	6

Expressions (contd.)

- More examples:

User Input	OCAML's Response
`2 + 3 * 4;;`	`- : int = 14`
`-2 + 3 * 4;;`	`- : int = 10`
`(-2 + 3) * 4;;`	`- : int = 4`
`4.4 ** 2.0;;`	`- : float = 19.36`
`2 + 2.2;;`	`... This expression has type float but is used here with type int`
`2.7 + 2.2;;`	`... This expression has type float but is used here with type int`
`2.7 +. 2.2;;`	`- : float = 4.9`

Operators

Operators	Types
`+` `-` `*` `/` `mod`	Integer arithmetic
`+.` `-.` `*.` `/.` `**`	Floating point arithmetic
`&&`, `||`, `not`	Boolean operations

Value definitions

- Syntax: `let` ⟨*name*⟩ `=` ⟨*expression*⟩ `;;`
- Examples:

User Input	OCAML's Response
`let x = 1;;`	`val x : int = 1`
`let y = x + 1;;`	`val y : int = 2`
`let x = x + 1;;`	`val x : int = 3`
`let z = "OCAML rocks!";;`	`val z : string = "OCAML rocks!"`
`let w = "21";;`	`val w : string = "21"`
`let v = int_of_string(w);;`	`val v : int = 21`

Functions

- Syntax: `let` ⟨*name*⟩ {⟨*argument*⟩} `=` ⟨*expression*⟩ `;;`
- Examples:

User Input	**OCAML's Response**
`let f x = 1;;`	`val f : 'a -> int = <fun>`
`let g x = x;;`	`val g : 'a -> 'a = <fun>`
`let inc x = x + 1;;`	`val inc : int -> int = <fun>`
`let sum(x,y) = x+y;;`	`val sum : int * int -> int = <fun>`
`let add x y = x+y;;`	`val add : int -> int -> int = <fun>`

Note the use of *parametric polymorphism* in functions `f` and `g`

More Examples of Functions

`let max(x, y) =`
`  if x < y`
`  then y`
`  else x;;`
	`val max : 'a * 'a -> 'a = <fun>`

`let mul(x, y) =`
`  if x = 0`
`  then 0`
`  else y+mul(x-1,y);;`
	`Unbound value mul`

`let rec mul(x, y) =`
`  if x = 0`
`  then 0`
`  else y+mul(x-1,y);;`
	`val mul : int * int -> int = <fun>`

`let rec mul(x, y) =`
`  if x = 0`
`  then 0`
`  else let i = mul(x-1,y)`
`  in y+i;;`
	`val mul : int * int -> int = <fun>`

An Aside on Polymorphism

- Java supports two kinds of polymorphism:
  - *Subtype* polymorphism: a method defined for objects of class A can be applied to objects of A's subclasses.
  - *Ad-hoc* polymorphism: a method name can be overloaded, with same name representing many different methods.
- *Templates* in C++ support an additional kind of polymorphism:

```
template <typename T>
  int f(T x) { return 1; }

template <typename T>
  T g(T x) { return x; }
```

Data structures

Examples of built-in data structures (lists and tuples):

User Input	**OCAML's Response**
`[1];;`	`- : int list = [1]`
`[4.1; 2.7; 3.1];;`	`- : float list = [4.1; 2.7; 3.1]`
`[4.1; 2];;`	`... This expression has type int but is used here with type float`
`[[1;2]; [4;8;16]];;`	`- : int list list = [[1;2], [4;8;16]]`
`1::2::[]`	`- : int list = [1; 2]`
`1::(2::[])`	`- : int list = [1; 2]`
`(1,2);;`	`- : int * int = (1, 2)`
`();;`	`- : unit = ()`
`let (x,y) = (3,7);;`	`val x : int = 3` `val y : int = 7`

Pattern Matching

- Used to "deconstruct" data structures.
- Example:

```
let rec sumlist l =
   match l with
        [] -> 0
      | x::xs -> x + sumlist(xs);;
```

- When evaluating `sumlist [2; 5]`
  - The argument `[2; 5]` matches the pattern `x::xs`,
  - ... setting `x` to 2 and `xs` to `[5]`
  - ... then evaluates `2 + sumlist([5])`

Match

`match` is analogous to a "switch" statement

- Each case describes
  - a pattern (lhs of '->') and
  - an expression to be evaluated if that pattern is matched (rhs of '->')
  - patterns can be constants, or terms made up of constants and variables
- The different cases are separated by '|'
- A matching pattern is found by searching in order (first case to last case)
- The first matching case is selected; *others are discarded*

```
let emptyList l =
   match l with
      [] -> true
    | _ -> false;;
```

Pattern Matching (contd.)

- Pattern syntax:
  - Patterns may contain "wildcards" (i.e. '_'); each occurrence of a wildcard is treated as a new anonymous variable.
  - Patterns are linear: any variable in a pattern can occur *at most once*.
- Pattern matching is used very often in OCAML programs.
- OCAML gives a shortcut for defining pattern matching in functions with one argument.
  Example:

```
let rec sumlist l =
   match l with
        [] -> 0
      | x::xs -> x +
                 sumlist(xs);;
```

```
let rec sumlist =
   function
        [] -> 0
      | x::xs -> x +
                 sumlist(xs);;
```

Functions on Lists

- Add one list to the end of another:

```
let rec append v1 v2 =
  match v1 with
      [] -> v2
    | x::xs -> x::(append xs v2);;
```

  - Note that this function has type

    `append: 'a list -> 'a list -> 'a list`

    and hence can be used to concatenate arbitrary lists, as long as the list elements are of the same type.
  - This function is implemented by builtin operator `@`
- Many list-processing functions are available in module `List`. Examples:
  - `List.hd`: get the first element of the given list
  - `List.rev`: reverse the given list

Enumerated Types

- A finite set of values
- Two values can be compared for equality
- There is no order among values
- Example:

```
type primaryColor = RED | GREEN | BLUE;;
type status = Freshman | Sophomore | Junior | Senior;;
```

- Syntax: `type` ⟨*name*⟩ `=` ⟨*name*⟩ { `|` ⟨*name*⟩ } `;;`
- A note about names:
  - Names of constants must begin with an *uppercase* letter.
  - Names of types, functions and variables must begin with a *lowercase* letter.
  - Names of constants are global within a module and not local to its type.

Records

- Used to define structures with named fields.
- Example:

```
type student = {name:string;
     gpa:float; year:status;};;
```

- Syntax: `type` ⟨*name*⟩ `=` `{` `{` ⟨*name*⟩ `{:` ⟨*name*⟩ `;` `}` `}` `;;`
- Usage:
  - Creating records:

    ```
    let joe = {name="Joe"; gpa=2.67; year=Sophomore;};;
    ```

  - Accessing fields:

    ```
    let x = joe.gpa;;  (* using "." operator *)
    let {gpa=x} = joe;; (* using pattern matching *)
    ```

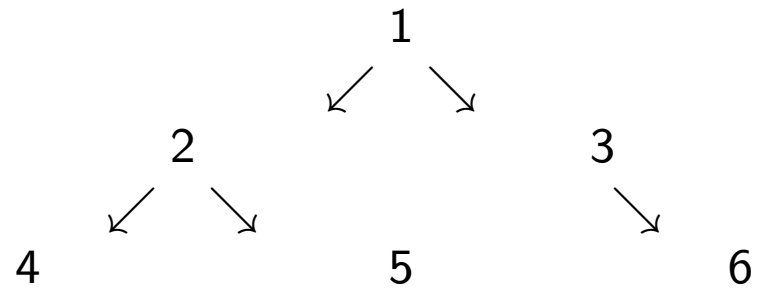
- Field names are global within a module and not local to its type.

Union types

- Used to define (possibly recursive) structured data with tags.
- Example:

```
type iTree = Node of int * iTree * iTree | Empty;;
```

- The empty tree is denoted by `Empty`
- The tree with one node, with integer 2, is denoted by `Node(2,Empty,Empty)`
- Tree



Denoted by

```
Node(1,
  Node(2,
    Node(4, Empty, Empty),
    Node(5, Empty, Empty))
  Node(3,
    Empty,
    Node(6, Empty, Empty)))
```

Union Types (contd.)

- Generalizes enumerated types
- Constants that tag the different structures in an union (e.g. `Node` and `Empty`) are called *data constructors*.
- Usage example: counting the number of elements in a tree:

```
let rec nelems tree =
  match tree with
      Node(i, lst, rst) ->
              (* 'i' is the value of the node;
                 'lst' is the left sub tree; and
                 'rst' is the right sub tree *)
              1 + nelems lst + nelems rst
    | Empty -> 0;;
```

Polymorphic Data Structures

- Structures whose components may be of arbitrary types.
- Example:

```
type 'a tree = Node of 'a * 'a tree * 'a tree | Empty;;
```

- 'a in the above example is a *type variable* ... analogous to the *typename* parameters of a C++ template
- Parameteric polymorphism enforces that all elements of the tree are of the same type.
- Usage example: traversing a tree in preorder:

```
let rec preorder tree =
  match tree with
      Node(i, lst, rst) -> i::(preorder lst)@(preorder rst)
    | Empty -> [];;
```

Exceptions

- **Total function:** function is defined for every argument value.
  Examples: `+`, `length`, etc.
- **Partial function:** function is defined only for a subset of argument values.
  - Examples: `/`, `Lists.hd`, etc.
  - Another example:

```
(* find the last element in a list *)
let rec last = function
    x::[] -> x
  | _::xs -> last xs;;
```

  - Exceptions can be used to signal invalid arguments.
  - Failed pattern matching (due to incomplete matches) is signalled with (predefined) `Match_failure` exception.
- Exceptions also signal unexpected conditions (e.g. I/O errors)

Exceptions (contd.)

- Users can define their own exceptions.
- Exceptions can be thrown using `raise`

```
(* Exception to signal no elements in a list *)
exception NoElements;;

let rec last = function
    [] -> raise NoElements
  | x::[] -> x
  | _::xs -> last xs;;
```

- Exceptions can be handled using `try ... with`.

```
exception DumbCall;;
let test l y =
   try (last l) / y
   with
        NoElements -> 0
      | Division_by_zero -> raise DumbCall;;
```

Functions of functions

- Add 1 to every element in list:

```
let rec add_one = function
      [] -> []
    | x::xs -> (x+1)::(add_one xs);;
```

- Multiply every element in list by 2:

```
let rec double = function
      [] -> []
    | x::xs -> (x*2)::(double xs);;
```

- Perform function f on every element in list:

```
let rec map f = function
      [] -> []
    | x::xs -> (f x)::(map f xs);;
```

- Now we can write add_one and double as:

```
let add_one = map ((+) 1);; let double = map (( * ) 2);;
```

More examples of higher-order functions

Sum all elements in a list	Multiply all elements in a list

`let rec sumlist = function`
`  [] -> 0`
`  | x::xs -> x + sumlist xs;;`
	`let rec prodlist = function`
	`  [] -> 1`
	`  | x::xs -> x * prodlist xs;;`

- Accumulate over a list:

```
let rec foldr f l b =
(* f is the function to apply at each element;
   l is the list
   b is the base case value *)
  match l with
    [] -> b
  | x::xs -> f x (foldr f xs b);;
```

- Using `foldr`:

Sum all elements in a list	Multiply all elements in a list
`let sumlist xs = foldr (+) xs 0;;`	`let prodlist xs = foldr ( * ) xs 1;;`

Writing Efficient OCAML Programs

- Using recursion to sum all elements in a list:

OCAML	C

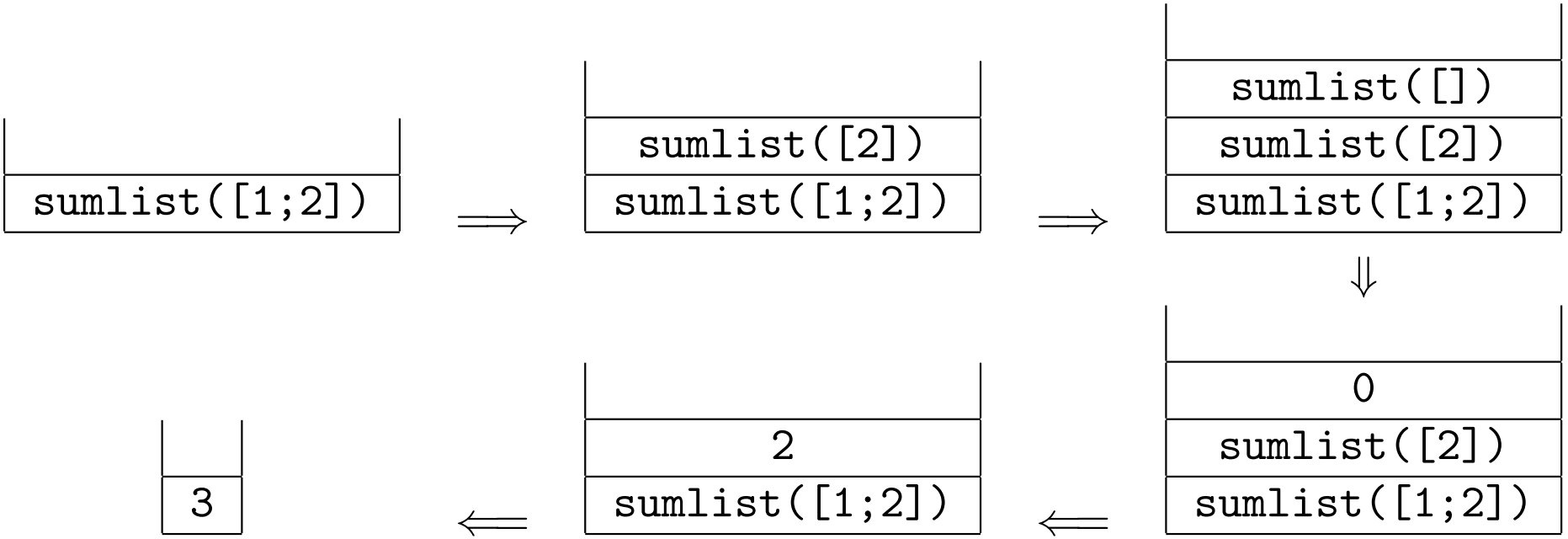
`let rec sumlist = function`
`  [] -> 0`
`  | x::xs -> x + sumlist xs;;`
	`int sumlist(List l) {`
	`  if (l == NULL)`
	`  return 0;`
	`  else`
	`  return (l->element) +`
	`  sumlist(l->next);`
	`}`

- Iteratively summing all elements in a list (C):

```
int acc = 0;
for(l=list; l!=NULL; l = l->next)
   acc += l->element;
```

Iteration vs. Recursion

- Recursive summation takes stack space proportional to the length of the list

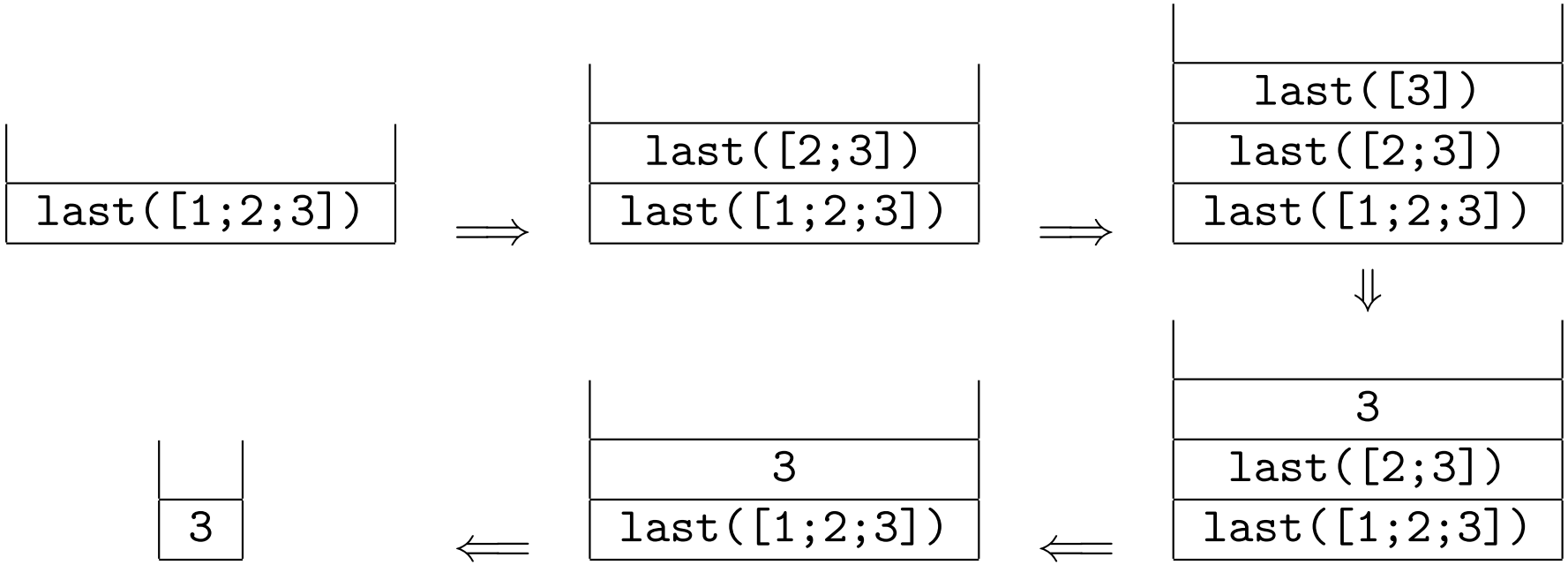


- Iterative summation takes constant stack space.

Tail Recursion

```
let rec last = function
    [] -> raise NoElements
  | x::[] -> x
  | _::xs -> last xs;;
```

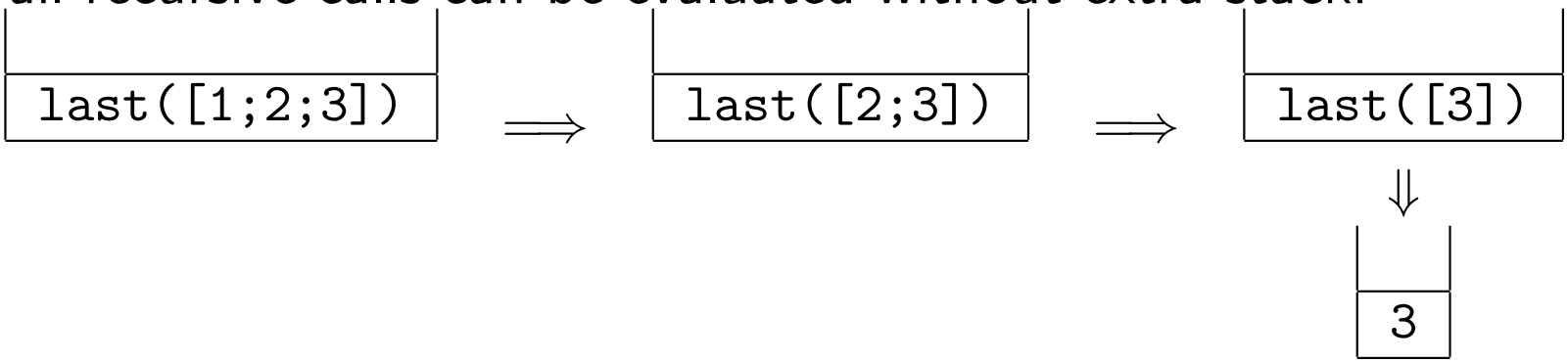
- Evaluation of `last [1;2;3];;`



Tail Recursion (contd.)

```
let rec last = function
    [] -> raise NoElements
  | x::[] -> x
  | _::xs -> last xs;;
```

- Note that when the 3rd pattern matches, the result of `last` is whatever is the result of `last xs`.
  Such calls are known as *tail recursive calls*.
- Tail recursive calls can be evaluated without extra stack:



Efficient programs using tail recursion

Example: efficient recursive function for summing all elements:

C	OCAML

`int acc_sumlist(int acc, List l) {`
`  if (l == NULL)`
`  return acc;`
`  else`
`  return acc_sumlist(`
`  acc + (l->element),`
`  l->next);}`
`int sumlist(List l) {`
`  return acc_sumlist(0, l);}`
	`let rec acc_sumlist acc =`
	`  function`
	`  [] -> acc`
	`  | x::xs -> acc_sumlist`
	`  (acc+x)`
	`  xs;;`
	`let sumlist l =`
	`  acc_sumlist 0 l;;`