CSE 307: Principles of Programming Languages

Syntax

R. Sekar

Topics

Section 1

Introduction

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.

Functional Programming (Contd.)

- Usually support complex (recursive) data types

  ... with automatic allocation and deallocation of memory (e.g. garbage collection)
- No loops: recursion is the only way to structure repeated computations
- 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

- 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.

Section 2

Basics

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 focus on 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 David Matuszek's tutorial for a quick intro, then go to Jason Hickey's tutorial. To clarify syntax etc. see OCAML manual. (http://caml.inria.fr/tutorials-eng.html)

Expression Evaluation

- 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

Expression Evaluation (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`

Section 3

Functions

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 example 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>`

Currying

- Named after H.B. Curry
- Curried functions take arguments one at a time, as opposed to taking a single tuple argument
- When provided with number of arguments less than the requisite number, result in a closure
- When additional arguments are provided to the closure, it can be evaluated

Currying Example

- Tuple version of a function

```
fun add(x,y) = x+y:int;
val add = fn int * int -> int
```

- Curried version of the same function

```
fun addc x y = x+y:int;
val addc = fn : int -> int -> int
```

- When addc is given one argument, it yields a function with type `int -> int`

```
- add 2 3;          - add 2;
it = 5 : int;       it = fn : int->int
                    - it 3;
                    it = 5 : int
```


Recursion

- Recursion is the means for iteration
- Consider the following examples

```
fun f(0) = 0
|   f(n) = 2*f(n-1);

fun g(0) = 1
|   g(1) = 1
|   g(n) = g(n-1)+g(n-2);

fun h(0) = 1
|   h(n) = 2*h(n div 2);
```

Section 4

Data Structures

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`

Tuples

```
(2,"Andrew") : int * string
(true,3.5,"x") : bool * real * string
((4,2),(7,3)) : (int * int) * (int * int)
```

- Tuple components can be of different types, but lists must contain elements of same type

```
[1,2,3] : int list
["Andrew","Ben"] : string list
[(2,3),(2,2),(9,1)] : (int * int) list
[[],[1],[1,2]] : int list list
```

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])`

Pattern Matching (Contd.)

- `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 Syntax

- 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 `@`

Functions on Lists (Contd.)

- Many list-processing functions are available in module `Lists`. Examples:
  - `Lists.hd`: get the first element of the given list
  - `Lists.rev`: reverse the given list

User-defined Types

- *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.

Record types

- 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 {id=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

<pre>   1
           ↙   ↘
      2   3
    ↙   ↘   ↘
  4   5   6</pre>
	<pre>Node(1,
	  Node(2,
	    Node(4, Empty, Empty),
	    Node(5, Empty, Empty))
	  Node(3,
	    Empty,
	    Node(6, Empty, Empty)))</pre>

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;;
```

Recursive Types

- Direct definition of recursive types is supported in SML using datatype declarations.

```
- datatype intBtree =
      LEAF of int
      | NODE of int * intBtree * intBtree;
   datatype intBtree =
      LEAF of int
      | NODE of int * intBtree * intBtree
```

- We are defining a binary tree type inductively:
  - Base case: a binary tree with one node, called a LEAF
  - Induction case: construct a binary tree by constructing a new node that sores an integer value, and has two other binary trees as children

Recursive Types (Contd.)

- We may construct values of this type as follows:

```
- val l = LEAF(1);
 val l = LEAF 1 : intBtree
 - val r = LEAF(3);
 val r = LEAF 3 : intBtree
 - val n = NODE(2, l, r);
 val n = NODE (2,LEAF 1,LEAF 3) : intBtree
```


Recursive Types (Contd.)

- Types can be mutually recursive. Consider:

```
-datatype expr = PLUS of expr * expr |
=                PROD of expr * expr |
=               FUN  of (string * exprs) |
=                IVAL of int
=and
=          exprs = EMPTY
=               | LIST of expr * exprs;
datatype expr = FUN of string * exprs
             | PLUS of expr * expr
             | PROD of expr * expr
datatype exprs = EMPTY | LIST of expr * exprs
```

- The key word and is used for mutually recursive type definitions.

Recursive Types (Contd.)

- We could also have defined expressions using the predefined list type:

```
- datatype expr=PLUS of expr*expr|PROD of expr*expr
=                |FUN of string * expr list;
datatype expr
  = FUN of string * expr list | PLUS of expr * expr
  | PROD of expr * expr
```

- Examples: The expression `3 + (4 * 5)` can be represented as a value of the above datatype expr as follows

Recursive Types (Contd.)

- The following picture illustrates the structure of the value `pl` and how it is constructed from other values.

```
 Pl ------>  PLUS
            /    \
           /      \
v3 ---> IVAL     PROD <----- pr
         |        /\
         |       /  \
         3   /->IVAL IVAL <--- v4
            /     |    |
         v5       |    |
                  5    4
```

```
val  v3 = IVAL(3);
val  v5 = IVAL(5);
val  v4 = IVAL(4);
val  pr = PROD(v5, v4);
val  pl = PLUS(v3, pr);
```


Recursive Types (Contd.)

- Similarly, `f(2,4,1)` can be represented as:

```
val a1 = EMPTY;
val a2 = ARG(IVAL(4), a1);
val a3 = ARG(IVAL(2), a2);
val fv = FUN("f", a3);
```

- Note the use of `expr list` to refer to a list that consists of elements of type `expr`

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 -> [];;
```


Parameterized Types

```
type (<typeParameters>) <typeName> = <typeExpression>
type ('a, 'b) pairList = ('a * 'b) list;
```

Datatype declarations for parameterized data types: Define Btree:

```
- datatype ('a,'b) Btree = LEAF of 'a
                         | NODE of 'b * ('a,'b) Btree * ('a,'b) Btree;
datatype ('a,'b) Btree   = LEAF of 'a
                         |  NODE of 'b * ('a,'b) Btree * ('a,'b) Btree
- type intBTree = (int, int) Btree;
type intBTree   = (int,int) Btree
```


Example Functions and their Type

```
- fun leftmost(LEAF(x)) = x
   =         | leftmost(NODE(y, l, r)) = leftmost(l);
   val leftmost = fn : ('a,'b) Btree -> 'a
- fun discriminants(LEAF(x)) = nil
   =     | discriminants(NODE(y, l, r)) =
   =       let
   =          val l1 = discriminants(l)
   =          val l2 = discriminants(r)
   =       in
   =          l1 @ (y::l2)  (* âĂIJ@âĂİ is list concatenation operator *)
   =       end;
   val discriminants = fn : ('a,'b) Btree -> 'b list
```

Example Functions (Contd.)

```
- fun append(x::xs, y) = x::append(xs, y)
   =      | append(nil, y) = y;
  val append = fn : 'a list * 'a list -> 'a list
- fun f(x::xs, y) = x::f(xs, y)
   =      | f(nil, y)    = nil;
  val f = fn : 'a list * 'b -> 'a list
```

- SML Operators that restrict polymorphism:
  - Arithmetic, relational, boolean, string, type conversion operators
- SML Operators that allow polymorphism
  - tuple, projection, list, equality (= and <>)

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 (Contd.)

- 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;;
```


Higher Order Functions

- Functions that take other functions as arguments, or return newly constructed functions

```
fun map f nil = nil
|   map f x::xs=(f x)::(map f xs);
```

- Map applies a function to every element of a list

```
fun filter f nil = nil
|   filter f x::xs=
       if (f x) then x::(filter f xs)
       else (filter f xs)
```


Higher Order Functions (Contd.)

```
fun zip f nil nil = nil
|   zip f (x::xs) (y::ys)=f(x,y)::(zip f xs ys);
fun reduce f b nil = b
|    reduce f b x::xs = f(x, (reduce f b xs));
```

Examples of Higher Order 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);;
```


Examples of Higher Order Functions (Cont.d)

- 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

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 b = function
(* f is the function to apply at element;
   b is the base case value *)
     []    -> b
   | x::xs -> f x (foldr f b xs);;
```

More Examples (Contd.)

- Using `foldr`:

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

Anonymous Functions

- You can define an unnamed function

```
-((fn x => 2*x) 5);
val it=10 : int
```

- Is handy with higher order functions

Section 5

Overview

Summary

- OCAML *definitions* have the following syntax:

⟨*def*⟩ ::= `let` [`rec`] ⟨*letlhs*⟩ `=` ⟨*expr*⟩
    (value definitions)
  | `type` ⟨*typelhs*⟩ `=` ⟨*typeexpr*⟩
    (type definitions)
  | *exception definitions . . .*

⟨*letlhs*⟩ ::= ⟨*id*⟩ [{⟨*pattern*⟩}]
    (patterns specify "parameters")

⟨*typelhs*⟩ ::= [{⟨*typevar*⟩}]⟨*id*⟩
    (typevars specify "parameters")

- OCAML programs are a sequence of definitions separated by `;;`

Summary

- OCAML *expressions* have the following syntax:

⟨*expr*⟩ ::= ⟨*const*⟩
    (constants)
  | ⟨*id*⟩
    (value identifiers)
  | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
    (expressions with binary operators)
  | ⟨*expr*⟩ ⟨*expr*⟩
    (function application)
  | `let` [`rec`] {⟨*letlhs*⟩ `=` ⟨*expr*⟩`;;`}`in` *expr*
    (let definitions)
  | `raise` ⟨*expr*⟩
    (throw exception)

Summary (Contd.)

  | `match` *expr* `with` ⟨*case*⟩ [{ `|`⟨*case*⟩ }]
    (pattern matching)
  | `fun` ⟨*case*⟩
    (function definition)
  | `function` ⟨*case*⟩ [{ `|`⟨*case*⟩ }]
    (function definition with pattern matching)
  | `try` *expr* `with` ⟨*case*⟩ [{ `|`⟨*case*⟩ }]
    (exception handling)

⟨*case*⟩ ::= ⟨*pattern*⟩ `->` ⟨*expr*⟩
    (pattern matching case)

Section 6

OCAML Performance

Writing Efficient OCAML Programs

- Using recursion to sum all elements in a list:

OCAML	C

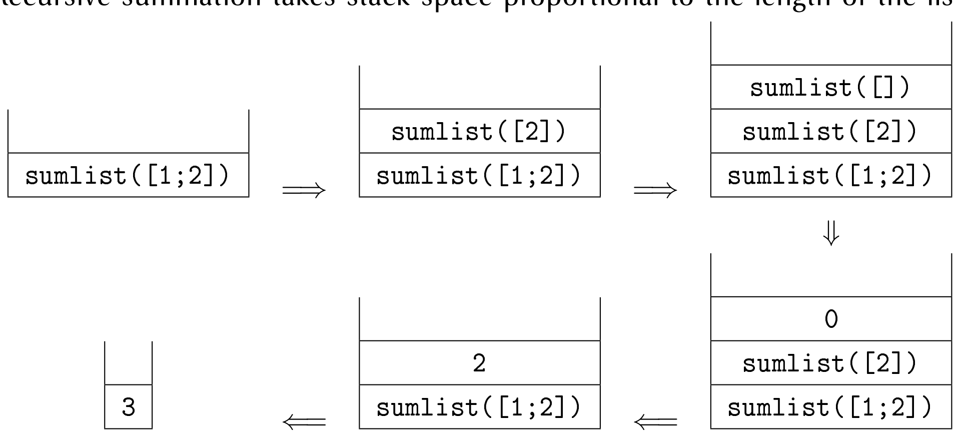
`let rec sumlist = function`
`    []     -> 0`
`  | x::xs -> x + sumlist xs;;`
	`int sumlist(List l) {`
	`   if (l == NULL)`
	`     return 0;`
	`   else`
	`     return (l->element) +`
	`                sumlist(l->next);`
	`}`

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

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


Writing Efficient OCAML Programs (Contd.)

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

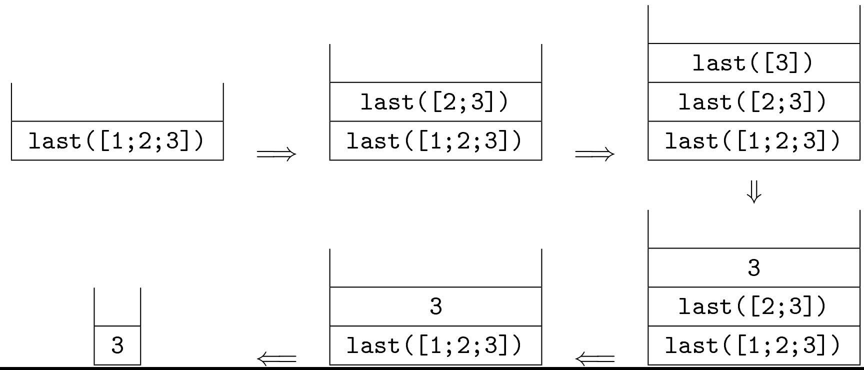


- Iterative summation takes constant stack space.

Tail Recursion

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

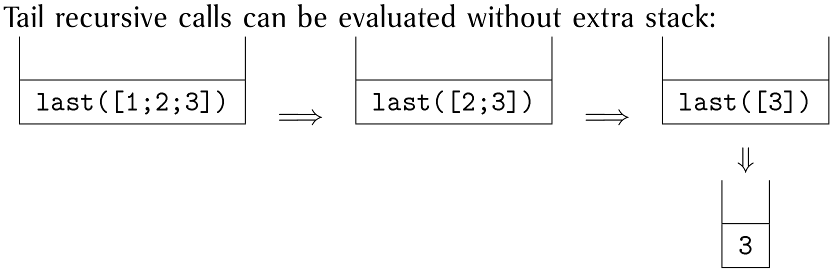
- Evaluation of `last [1;2;3];;`



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:



Taking Efficiency by the Tail

- An 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;;`