Exceptions in OCaml

In a nutshell

- exceptions have general type `exn` and are created with constructors
- exception generation:
  predefined function `raise : exn -> 'a`
- exception handling:
  try e **with** p_1 `->` e_1 `| ... |` p_n `->` e_n

Remarks

- `raise` does not actually return any value
- the returned type `'a` allows `raise` to be used in any context

Exceptions in OCaml

Declaration of exception constructors: syntax

```
Dec ::= 'exception' CONS_ID ('of' Type)?
```

Remark: `CONS_ID` must start with an uppercase letter

Declaration of exception constructors: examples

```
exception Fault;;                (* constant constructor *)
exception Fault1 of string;;     (* a unary constructor *)
exception Fault2 of string*exn;; (* a binary constructor *)
let exc=Fault;;
let exc1=Fault1 "error message";;
let exc2=Fault2 ("msg",exc);;
```

Remarks

- the usual laws for constructors apply to exception constructors
- constructors are always uncurried

Exceptions in OCaml

Predefined exceptions and functions (a selection)

```
(* self-explanatory, no additional info *)
exception Division_by_zero;;

(* general exception with an error message *)
exception Failure of string;;

(* self-explanatory, associated info: function  name *)
exception Invalid_argument of string;;

(* self-explanatory, associated info: file name, code line and column *)
exception  Match_failure of string*int*int;;

(* predefined function failwith *)
let failwith msg = raise (Failure msg);; (* failwith : string -> 'a *)
```

Exceptions in OCaml

Examples of change of the control flow due to exceptions

```
let hd = function
    hd::_ -> hd
  | _ -> failwith "hd";;

(* failed to compute the list of [], x+1 is not evaluated *)
let x=hd [] in x+1;;
Exception: Failure "hd".

(* failed to find an element at index 4, x+1 is not evaluated *)
let x=List.nth [1;2;3] 4 in x+1;;
Exception: Failure "nth".

(* index -1 is not valid, x+1 is not evaluated *)
let x=List.nth [1;2;3] (-1) in x+1;;
Exception: Invalid_argument "List.nth".

(* exceptions due to wrong input are handled with try _ with *)
let x=try List.nth [1;2;3] (read_int()) with Invalid_argument _ -> 1 | _ -> 3
in x+1;;
```

Standard floating-point numbers

In a nutshell

- predefined type **`float`**
- literals (= constant constructors) with the standard syntax
- standard binary operators `+.` `-.` `*.` `/.` `**`
- global variables `nan`, `infinity`, `neg_infinity`
- many other features in Stdlib (implicitly imported)
- more features in module Float

Remarks

- **`int`** and **`float`** not compatible, no implicit conversions
- example

```
(+): int -> int -> int
(+.): float -> float -> float
```

```
3.14 * 2;;
^^^^
Error: This expression has type float but an expression was expected of type
    int
```

Variant types

In a nutshell

They allow users to define new types with their constructors

Example with only constant constructors

```
type color = Red | Green | Blue;; (* just constant constructors *)

let to_string = function (* to_string : color -> string *)
    Red -> "red"
  | Green -> "green"
  | Blue -> "blue";;

List.map to_string [Red; Blue; Green; Blue];;
- : string list = ["red"; "blue"; "green"; "blue"]
```

Remarks

- type identifiers must start with a lowercase letter
- constructor identifiers must start with an uppercase letter

Variant types

Example with non-constant constructors

```
type shape = Square of float | Circle of float
      | Rectangle of float * float;;
```

Intutition: a shape is a union of different kinds of values

```
let perimeter = function  (* perimeter : shape -> float *)
    Square side -> 4.0 *. side
  | Circle ray -> 2.0 *. Float.pi *. ray
  | Rectangle (width,height) -> 2.0 *. (width +. height);;

perimeter (Square 4.0);;
- : float = 16.
```

Remarks

constructors cannot be curried

Variant types

Recursive variant types

- declarations of variant types can be recursive
- typical use: definition of tree structures
- intuition: each constructor corresponds to a different kind of node

Example: abstract syntax trees

Implementation of abstract syntax trees (AST) for expressions with integer literals, unary minus, and binary addition and multiplication

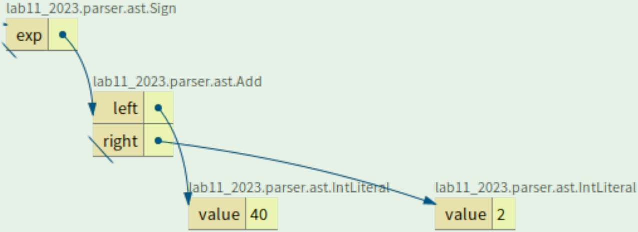
```
type exp_ast =
      IntLiteral of int          (* integer literals *)
    | Sign of exp_ast            (* unary minus *)
    | Add of exp_ast * exp_ast (* binary addition *)
    | Mul of exp_ast * exp_ast (* binary multiplication *);;
```

Recursive variant types

Example of abstract syntax tree

```
type exp_ast =
      IntLiteral of int         (* integer literals *)
    | Sign of exp_ast           (* unary minus *)
    | Add of exp_ast * exp_ast  (* binary addition *)
    | Mul of exp_ast * exp_ast  (* binary multiplication *);;

(* tree corresponding to concrete syntax -(40+2) *)
let ast = Sign(Add(IntLiteral 40,IntLiteral 2));;
```



Recursive variant types

Example: evaluation of expressions

- type of the function: `eval : exp_ast -> int`
- specification: `eval t` returns the value of the expression represented by the tree `t`
- implementation: a recursive depth-first visit of the tree `t`

```
# let rec eval = function (* eval : exp_ast -> int *)
    IntLiteral n -> n
  | Sign exp -> - eval exp
  | Mul (exp1,exp2) -> eval exp1 * eval exp2
  | Add (exp1,exp2) -> eval exp1 + eval exp2;;
val eval : exp_ast -> int = <fun>
# let ast = Sign(Add(IntLiteral 40,IntLiteral 2));;
val ast : exp_ast = Sign (Add (IntLiteral 40, IntLiteral 2))
# eval ast;;
- : int = -42
```

Polymorphic variant types

Example: option type

```
type 'a option = None | Some of 'a;; (* bult-in type in OCaml *)
```

Module Option

Some useful functions defined in the module:

```
let is_none = function (* is_none : 'a option -> bool *)
    None -> true
  | _ -> false;;

let is_some = function (* is_some : 'a option -> bool *)
    Some _ -> true
  | _ -> false;;

let get = function (* get : 'a option -> 'a *)
    Some v -> v
  | _ -> raise (Invalid_argument "get");;
```

Polymorphic variant types

Example with Option

- implementation of the function:
  `find : ('a -> bool)-> 'a list -> 'a option`
- specification: `find p ls` returns
  - `Some e` if `e` is the first element in `ls` such that `p e = true`
  - `None` if there are no elements `e` in `ls` such that `p e = true`

```
let find p =
    let rec aux = function
        hd::tl -> if p hd then Some hd else aux tl
      | _ -> None
    in aux;;
```

Polymorphic variant types

Example with Option

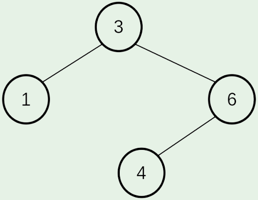
```
# let v=find ((<) 0) [-1;-2;3];; (* (<) 0 means fun x -> 0 < x  *)
val v : int option = Some 3
# Option.is_some v;;
- : bool = true
# Option.get v;;
- : int = 3
# let v=find ((<) 0) [-1;-2;-3];;
val v : int option = None
# Option.is_none v;;
- : bool = true
# Option.get v;;
Exception:  Invalid_argument "get".
```

Polymorphic and recursive variant type

Example: binary search trees

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

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



Polymorphic and recursive variant type

Example: binary search trees

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

(* member and insert for binary search trees *)

let rec member el = function (* member : 'a -> 'a btree -> bool *)
    Node(label,left,right) ->
      el=label ||
      if el<label then member el left else member el right
  | _ -> false;; (* the remaining case is Empty *)

let rec insert el = function (* insert : 'a -> 'a btree -> 'a btree *)
    Node(label,left,right) as t ->
        if el=label then t (* t abbreviates Node(label,left,right) *)
        else if el<label then Node(label,insert el left,right)
        else Node(label, left, insert el right)
  | _ -> Node(el,Empty,Empty);; (* the remaining case is Empty *)
```