2023-10-09 - Moduli, Interfacce sig, Strutture struct, Funtori

Moduli (modules)

I moduli sono utilizzati per realizzare dei **tipi di dato** *(abstract data type - ADT)*. Contengono sia **tipi** che **funzioni** del modulo. Il nome dei moduli inizia con la lettera maiuscola.

I moduli sono composti da due parti: l'**interfaccia** *(opzionale)* e l'**implementazione**.

Esistono anche moduli che descrivono *solo* un tipo di dato *astratto*, ovvero **solo la sua interfaccia**. Vengono definiti attraverso la sintassi `module type = sig ... end`.

Interfaccia *(signature)*

L'**interfaccia** pubblica (**signature**), che espone dei tipi e funzioni astratte.
È **opzionale**, in caso non venga specificata, è **inferita dall'implementazione** (`struct`) dichiarata: verranno *esposti* tutti i *tipi* e le *funzioni* dell'implementazione.

Non è possibile definire **funzioni concrete**, ma è **possibile** definire dei **tipi** già **concreti** *(alias o varianti)*.

Implementazione *(structure)*

L'**implementazione concreta** di tipi e funzioni. In caso sia definita un'*interfaccia* (`sig`), vengono *esposti pubblicamente* solo quelli che sono *definiti* al suo interno.

I file contenti *interfacce* e *implementazioni* possono essere **separati** e compilati separatamente *(devono avere un nome diverso per poter essere importati)*.

Esempio: modulo *coda con priorità*

Modulo con *interfaccia* ed *implementazione*:

```
module PrioQueue :
    sig (* interfaccia *)
        type priority = int (* concreto *)
        type char_queue (* astratto *)

        val empty : char_queue
        val insert : char_queue -> int -> char -> char_queue
        val extract : char_queue -> int * char * char_queue
        exception QueueIsEmpty
    end =
    struct (* implementazione *)
        type priority = int
        type char_queue = Empty | Node of priority * char * char_queue * char_queue
        exception QueueIsEmpty

        let empty = Empty

        let rec insert queue newPrio newElem =
            match queue with
            | Empty -> Node(newPrio, newElem, Empty, Empty)
            | Node(headPrio, headElem, left, right) ->
                if newPrio <= headPrio
                    then Node(newPrio, newElem, insert right headPrio headElem, left)
                    else Node(headPrio, headElem, insert right newPrio newElem, left)

        let rec remove_top queue =
            match queue with
            | Empty -> raise QueueIsEmpty
            | Node(prio, elem, left, Empty) -> left
```

```
            | Node(prio, elem, Empty, right) -> right
            | Node(prio, elem, (Node(leftPrio, leftElem, _, _) as left),
                               (Node(rightPrio, rightElem, _, _) as right)) ->
                if leftPrio <= rightPrio
                    then Node(leftPrio, leftElem, remove_top left, right)
                    else Node(rightPrio, rightElem, left, remove_top right)

        let extract queue =
            match queue with
            | Empty -> raise QueueIsEmpty
            | Node(prio, elem, _, _) as queue -> (prio, elem, remove_top queue)
    end;;

let pq = PrioQueue.empty;;
let pq = PrioQueue.insert pq 10 'a';;
let pq = PrioQueue.insert pq 35 'b';;
let pq = PrioQueue.insert pq 4 'c';;
let priority, head, pq = PrioQueue.extract pq;; (* 4, 'c' *)
PrioQueue.remove_top pq;; (* Unbound value PrioQueue.remove_top *)
```

Modulo con solo *interfaccia*:

```
module type PrioQueueADT =
    sig
        (* interfaccia *)
    end;;
```

Modulo che *implementa* un *interfaccia* (`module type`):

```
module PrioQueue : PrioQueueADT =
    struct
        (* implementazione *)
    end;;
```

Funtori (functors)

I funtori sono delle *"funzioni"* che prendono in input una *struttura* (**modulo**) e restituiscono una nuova struttura (**modulo**).
Questo permette di gestire senza *duplicazione di codice* **moduli generici**, che **adattano** le funzioni al proprio interno per **aderire al tipo** di dato preso in input dal *funtore*.

Esempio: `Map`

Per inizializzare una mappa in OCaML è necessario chiamare il **funtore** `Map.Make` del modulo `Map`, che prende in input un tipo di dato (**un modulo**) completamente ordinato (`Map.OrderedType`) e restituisce un **nuovo modulo** che è il modulo mappa *"funzionante"* su quel tipo.

```
let StringMap = Map.Make(string);;
```

Il modulo dato *in input* ad un funtore deve **rispettare l'interfaccia** specificata. In questo caso `Map.OrderedType`, che contiene `type t` e `val compare : t -> t -> int`, entrambi contenuti nel tipo `string`.
Ora è possibile *utilizzare il modulo* `StringMap` come mappa concreta (`.add`, `.update`, `.find`, ...).

Esempio: funtore custom `Matcher`

L'obiettivo è scrivere una funzione che verifica se le *parentesi* in una stringa sono *bilanciate*, tramite l'uso di uno **stack di appoggio** (se alla fine lo stack è *vuoto* allora la stringa è *bilanciata*).

```
let is_balanced str =
    let s = Stack.empty in try
        String.iter
```

```
            (fun c -> match c with
                | '(' -> Stack.push s c
                | ')' -> Stack.pop s
                | _ -> ()) str;
            Stack.is_empty s
    with Stack.EmptyStackException -> false
```

Vogliamo utilizzare una **qualsiasi implementazione** di uno stack, quindi scriviamo l'interfaccia di uno stack astratto (`StackADT`):

```
module type StackADT =
    sig
        type char_stack
        exception EmptyStackException

        val empty : char_stack
        val push : char_stack -> char -> unit
        val top : char_stack -> char
        val pop : char_stack -> unit
        val is_empty : char_stack -> bool
    end;;
```

E racchiudiamo la funzione `is_balanced` dentro un "modulo" **funtore** che prende in input un qualsiasi tipo di dato stack che *rispetta* l'interfaccia `StackADT` e lo *utilizza* dentro la funzione `is_balanced`:

```
module Matcher (Stack : StackADT) =
    struct
        let is_balanced str =
            let s = Stack.empty in try
                String.iter (fun c -> match c with
                    | '(' -> Stack.push s c
                    | ')' -> Stack.pop s
                    | _ -> ()) str;
                Stack.is_empty s
            with Stack.EmptyStackException -> false
    end;;
```

In questo modo **qualsiasi implementazione** di stack che rispetta l'interfaccia `StackADT` (ha il tipo `char_stack`, l'eccezione `EmptyStackException` e le funzioni `empty`, `push`, `top`, `pop`, `is_empty`) può essere **utilizzata** per la funzione `is_balanced`.

Utilizzo

Per utilizzare la **struttura concreta** "generata" dal funtore, è necessario **passare** al funtore un modulo che rispetta l'**interfaccia richiesta**.

```
(* StackX: implementazione concreta di StackADT *)
let MatcherStackX = Matcher(StackX);;
MatcherStackX.is_balanced "()))()";;
```

Compilazione

- `ocaml` avvia l'**interprete** *(REPL - Read, Evaluate, Print, and Loop)*
  - `#use "<nomefile>.ml"` importa il **file** nell'interprete *(facendo REPL)*
  - `#load "<liberia>.cma"` importa la **libreria** specificata
- `ocaml <nomefile>.ml` esegue il file *(senza fare REPL)*
- `ocamlc <nomefile>.ml` compila il file, generando il file **object** `.cmo`, il file **object dell'interfaccia** `.cmi` e l'**eseguibile** (`a.out` se non specificato `-o <output>`)
- `ocamlc -c <nomefile>.ml` compila il file, generando solo i file **oggetto** `.cmi`, `.cmo` (senza effettuare **linking**, quindi senza generare l'*eseguibile*)
- `ocamlc -c <nomefile>.mli` compila il file *(che contiene **solo interfacce**)*, generando solo il file oggetto `.cmi`

- `ocaml <modulo>.cmo <main>.ml` **includere** il modulo già compilato dentro un altro.
  - `Open <modulo>` usare il modulo importato