Linguagens e Ambientes de Programação (2018/2019)

Teórica 06 (25/mar/2019)

Efeitos laterais em OCaml. Motivação do tipo "unit" e do operador ";".
Input/Output em OCaml. Utilização do método indutivo na programação de funções sobre ficheiros.
Introdução aos módulos em OCaml.

Efeitos laterais

O OCaml usa um modelo de input/output imperativo, no qual as operações de input/output são tratadas como *efeitos laterais*.

O que é um **efeito lateral**? Um *efeito lateral* é qualquer atividade que uma função desenvolva para além de calcular um resultado a partir dos argumentos. Por exemplo:

- Escrever num ficheiro.
- Escrever no ecrã do computador.
- Ler dum ficheiro.

Como sabemos, no paradigma funcional é suposto uma função limitar-se a calcular um resultado a partir dos seus argumentos. O facto é que a linguagem OCaml ultrapassa os limites estritos do paradigma funcional, já que admite efeitos laterais nas funções.

Operador de sequenciação

Para sequenciar os efeitos laterais, o OCaml dispõe dum *operador de sequenciação* que se escreve ";" (como em Pascal!). Assim, por exemplo, para escrever no ecrã a string "ola", e logo a seguir a string "ole" usamos a seguinte função:

```
let hello () =
    print_string "ola" ; print_string "ole"
;;
```

Tecnicamente, ";" é o nome duma função com o seguinte tipo:

```
";": unit -> 'b -> 'b
```

e a seguinte definição usada internamente pelo OCaml:

```
";" x y = y ;;
```

Tipo unit e valor ()

Havendo efeitos laterias em OCaml, faria sentido definir funções sem argumentos ou sem resultados. Mas o OCaml não o permite, pois obriga todas as funções a ter argumentos e um resultado. O tipo **unit** foi inventado para ser usado nessas situações.

O tipo **unit** é um tipo básico com apenas um valor, que se escreve **()**. Para comparação, note que o tipo **bool** é um tipo com apenas dois valores, que se escrevem **true** e **false**. Por seu lado, o tipo **void** do C é um tipo sem valores e por isso não resolveria o nosso problema em OCaml.

Exemplos de utilização de unit e ().

```
print_string: string -> unit
read_int: unit -> int
print_newline: unit -> unit

let x = read_int () in
    print_int (x+1)
```

Exemplo de função para escrever uma lista de inteiros no ecrã:

```
let rec printList l =
    match l with
       [] -> ()
     | x::xs -> print_int x ; print_newline () ; printList xs
;;
```

Input-Output em OCaml

Canais

As operações sobre ficheiros são efetuadas através de **canais**. Os canais são valores dos tipos predefinidos *in_channel* e *out_channel*.

Os "canais" do OCaml correspondem às "streams" do Java.

Em OCaml existem três canais predefinidos que são automaticamente abertos quando o programa começa a correr:

- `stdin : in_channel`
- `stdout : out_channel`
- `stderr : out_channel`

Primitivas de input/output

- **Primitivas para escrita em stdout**

```
print_char: char -> unit
print_string: string -> unit
print_int: int -> unit
print_float: float -> unit
print_newline: unit -> unit
```

- **Primitivas para leitura de stdin**

```
read_line: unit -> string
read_int: unit -> int
read_float: unit -> float
```

- **Primitivas para abertura e fecho de canais**

```
open_in: string -> in_channel
open_out: string -> out_channel
close_in: in_channel -> unit
close_out: out_channel -> unit
```

- **Primitivas para escrita em canais**

```
output_char: out_channel -> char -> unit
output_string: out_channel -> string -> unit
flush: out_channel -> unit
```

- **Primitivas para leitura de canais**

```
input_char: in_channel -> char
input_line: in_channel -> string
```

Esta lista de primitivas não é exaustiva. Há mais primitivas listadas no manual de referência (ver Basic Operations).

Exemplo de função sobre ficheiros programada usando o método indutivo

Cópia de ficheiros:

```
(* copyChannel: copia o canal de input ci para o canal de output co *)

let rec copyChannel ci co =
    try
        let s = input_line ci in
            output_string co s ;
            output_string co "\n" ;
            copyChannel ci co
    with End_of_file -> ()
;;

(* copyFile: abre os ficheiros ni e depois usa a função copyChannel *)

let copyFile ni no =
    let ci = open_in ni in
        let co = open_out no in
            copyChannel ci co ;
            close_in ci ;
            close_out co
;;
```

Esquema geral de utilização do método indutivo no tratamento de ficheiros linha a linha:

```
let rec f ci =
    try
        let s = input_line ci in
            ... f ci ...
    with End_of_file -> ...
;;
```

Introdução aos módulos em OCaml

Nenhuma linguagem moderna dispensa um sistema de módulos. Um sistema de módulos permite:

- Agrupar definições relacionadas: por exemplo permite agrupar um tipo de dados com as suas operações associadas.
- Forçar uma forma coerente de nomear essas definições para evitar conflitos de nomes.
- Ocultar a representação interna dos dados e ocultar as operações auxiliares.
- Está na base duma conceção modular de software.

O sistema de módulos do OCaml é muito rico. Para não dedicarmos excessivo tempo a este assunto, vamos ignorar quase todos os aspetos técnicos e vocabulário especializado associados ao sistema. Focamos a nossa atenção no estudo de alguns cenários de utilização que, afinal, cobrem a maioria das necessidades práticas.

Utilização de módulos existentes

Dentro do interpretador `ocaml` estão sempre disponíveis todos os módulos de biblioteca do OCaml (Index of modules).

Mas para aceder a módulos criados pelo utilizador dentro do interpretador `ocaml`, usa-se a diretiva `#load` assim:

```
# #load "MySet.cmo" ;;
```

Para referenciar elementos dum módulo, podem usar-se referências qualificadas, como se exemplifica:

```
# Sys.os_type ;;
- : string = "Unix"

# Sys.command "ls -l" ;;
total 24
-rw-r--r-- 1 amd amd  259 2008-03-11 10:26 main.ml
-rw-r--r-- 1 amd amd  440 2008-03-11 10:28 MySet.cmi
-rw-r--r-- 1 amd amd  654 2008-03-11 10:28 MySet.cmo
-rw-r--r-- 1 amd amd  394 2008-03-11 10:11 MySet.ml
-rw-r--r-- 1 amd amd  164 2008-03-11 10:11 MySet.mli
- : int = 0

# List.rev [1;2;3] ;;
- : int list = [3; 2; 1]

#
```

Mas é bastante mais prático abrir (importar) primeiro o módulo, para depois não ter de usar prefixos qualificadores:

```
# open Sys ;;

# os_type ;;
- : string = "Unix"

# command "ls -l" ;;
total 24
-rw-r--r-- 1 amd amd  259 2008-03-11 10:26 main.ml
-rw-r--r-- 1 amd amd  440 2008-03-11 10:28 MySet.cmi
-rw-r--r-- 1 amd amd  654 2008-03-11 10:28 MySet.cmo
-rw-r--r-- 1 amd amd  394 2008-03-11 10:11 MySet.ml
-rw-r--r-- 1 amd amd  164 2008-03-11 10:11 MySet.mli
- : int = 0
#
```

Criação dum módulo aberto

Para criar um módulo aberto, no qual todas as definições são públicas, basta criar um ficheiro com o nome pretendido e extensão "ml", digamos "MySet.ml". Depois é necessário compilar o módulo usando o comando

```
ocamlc -c MySet.ml
```

sendo gerados os ficheiros "MySet.cmi" e "MySet.cmo".

Exemplo de utilização do módulo aberto MySet:

```
$ ocaml
        Objective Caml version 3.09.2

# #load "MySet.cmo" ;;
# open MySet ;;
```

```
# empty ;;
- : 'a list = []
# make [1;2;3] ;;
- : int list = [1; 2; 3]
#
```

Eis o conteúdo do ficheiro "MySet.ml":

```
(* Module body MySet *)

type 'a set = 'a list ;;

let empty = [] ;;

let rec belongs v l =
    match l with
      [] -> false
    | x::xs -> x = v || belongs v xs
;;

let rec union l1 l2 =
    match l1 with
      [] -> l2
    | x::xs -> (if belongs x l2 then [] else [x])@union xs l2
;;

let rec clear l =
    match l with
      [] -> []
    | x::xs -> let cl = clear xs in
                (if belongs x cl then [] else [x])@cl
;;

let make l =
    clear l
;;
```

Criação dum módulo fechado

Para ocultar a representação interna dos dados e as operações auxiliares é necessário criar adicionalmente um **ficheiro interface** "MySet.mli" onde se declaram todas as entidades públicas da forma que se pretende que sejam vistas do exterior. Depois compila-se o módulo assim

```
ocamlc -c MySet.mli MySet.ml
```

sendo gerados os ficheiros "MySet.cmi" e "MySet.cmo".

Exemplo de utilização do módulo fechado MySet:

```
$ ocaml
        Objective Caml version 3.09.2

# #load "MySet.cmo" ;;
# open MySet ;;
# empty ;;
- : 'a MySet.set = <abstr>
# make [1;2;3] ;;
- : int MySet.set = <abstr>
# clear [1;2;3] ;;
Unbound value clear
#
```

Eis o conteúdo do ficheiro "MySet.mli":

```
(* Module interface MySet *)
```

```
type 'a set                (* abstract *)
val empty : 'a set
val belongs : 'a -> 'a set -> bool
val union : 'a set -> 'a set -> 'a set
val make : 'a list -> 'a set
```

Repare que neste ficheiro interface omitimos a representação interna do tipo `'a set` assim como a declaração da função auxiliar `clear`. Repare ainda na subtileza do tipo da função `make` (recebe uma lista mas retorna um set).

Quando se esconde a representação interna dum tipo de dados, diz-se que esse tipo é **abstracto** ou **opaco**.

#100