Linguagens e Ambientes de Programação (2018/2019)

Teórica 05 (20/mar/2019)

Tipos produto (tuplos e registos).
Tratamento de exceções.
Funções parciais.

Tipos produto (tuplos e registos)

A maioria das linguagens de programação incluem nos seus sistemas de tipos uma construção específica que permite representar agrupamentos de dados heterogéneos. O nome genérico dessa construção, independente da linguagem particular, é **tipo produto**.

Numa linguagem com tipos produto é possível definir, por exemplo, um tipo de dados *pessoa* constituído por um nome (de tipo string), um ano de nascimento (de tipo int), uma morada (de tipo string), etc.

Os tipos produto do Pascal são os *registos*.
Os tipos produto do C são as *estruturas*.
Os tipos produto do Java, Smalltalk e C++ são as *classes*.
No Fortran, os tipos produto apareceram na versão Fortran 90 com o nome de *tipos derivados*.

Como é em OCaml?

Tipos produto em OCaml

Em OCaml há duas variedades de tipos produto: **tipos produto não etiquetados** e **tipos produto etiquetados**.

Tipos produto não etiquetados (tuplos)

Os produtos cartesianos do OCaml são exemplos de tipos produto, neste caso ditos **não-etiquetados**, e também conhecidos por **tuplos**. Por exemplo, para representar *pessoas*, pode usar-se em OCaml o seguinte tipos produto não etiquetado:

```
string * int * string
```

Literais: Para exemplificar, eis um valor do tipo anterior:

```
("João ", 1970, "Lisboa")
```

Construção: Para exemplificar vejamos uma função que muda a morada duma pessoa, criando um tuplo novo:

```
let moveTo (x,y,_) city = (x, y, city) ;;
```

Processamento: Como se processam tuplos? De duas formas:

- Usando emparelhamento de padrões:

```
let getName p =
    match p with
      (x, _, _) -> x
;;
```

- Ou usando as operações de acesso **fst** e **snd** predefinidas, se o registo tiver duas componentes:

```
não aplicável ao nosso exemplo
```

Tipos produto etiquetados (registos)

Mas o OCaml, também suporta **tipos produto etiquetados**, também conhecidos como **registos**, os quais requerem definição explícita. Eis um exemplo de tipo produto etiquetado:

```
type pessoa = { nome:string ; anoNasc:int ; morada:string } ;;
```

Literais: Eis um literal deste tipo:

```
{ nome = "João" ; anoNasc = 1970 ; morada = "Lisboa" }
```

Construção: Para exemplificar vejamos uma função que muda a morada duma pessoa, criando um registo novo:

```
let moveTo p city =
    { nome = p.nome ; anoNasc = p.anoNasc ; morada = city }
;;
```

Processamento: Como se processam registos? De duas formas:

- Usando emparelhamento de padrões:

```
let getNome p =
    match p with
      { nome = x ; anoNasc = _ ; morada = _ } -> x
;;
```

- Ou usando a operação de acesso "." e que funciona em OCaml exatamente como em Pascal ou C:

```
let getNome p = p.nome ;;
```

Tratamento de exceções em OCaml

Durante a execução de um programa, por vezes verificam-se determinadas condições (geralmente anómalas, mas nem sempre) às quais é necessário reagir alterando o fluxo de execução normal. Tais condições chamam-se **exceções**.

As exceções podem ser geradas:

- Ao nível do hardware. Por exemplo, em virtude duma divisão por zero.
- Ao nível do sistema operativo. Por exemplo, devido à impossibilidade de abrir um ficheiro inexistente, ou devido à tentativa de continuar a ler dum ficheiro que já chegou ao fim.
- Deliberadamente pelos próprios programas, usando a construção **raise**, ou usando a função **failwith**, que também gera uma exceção, mas duma forma mais prática de escrever. Por exemplo, para lidar explicitamente com **argumentos proibidos**:

```
let rec fact x =
    if x = 0 then 1
    else if x>0 then x * fact(x-1)
    else raise (Arg.Bad "fact")
;;

let rec fact x =
    if x = 0 then 1
    else if x>0 then x * fact(x-1)
    else failwith "fact"
;;
```

Captura e tratamento de exceções

Quando uma exceção é gerada, o controlo da execução do programa é transferido para o **tratador de exceções** (*exception handler*) mais recentemente cativado. É abortada a avaliação de todas as funções chamadas depois da ativação desse tratador de exceções.

Em OCaml, um tratador de exceções escreve-se usando uma expressão **try-with**, como se exemplifica de seguida. A expressão `exp`, no seu interior, diz-se uma **expressão protegida**.

```
try
    exp
with Sys_error _ -> exp1
   | Division_by_zero -> exp2
   | End_of_file -> exp3
   ...
;;
```

Como é avaliada uma expressão `try-with`?

- Começa-se por avaliar a expressão protegida `exp`.
- Caso nenhuma exceção seja gerada por `exp`, então o `try-with` não tem qualquer efeito e a expressão global `try-with` tem o mesmo valor da expressão `exp`.
- Mas se for gerada alguma exceção por `exp`, então o `try-with` recebe o controlo da execução e usa emparelhamento de padrões para descobrir qual das expressões `exp1`, `exp2`, `exp3`, etc., deve ser avaliada em substituição de `exp`.
- Finalmente, se o emparelhamento de padrões anterior falhar, então a exceção é propagada para o `try-with` cronologicamente anterior.

Existe um tratador de exceções de sistema que apanha as exceções não tratadas e aborta a execução do programa com uma mensagem de erro apropriada a cada caso. Exemplos:

```
# 4/0;;
Exception: Division_by_zero.

# open_in "fdsg" ;;
Exception: Sys_error "fdsg: No such file or directory".
```

Definição de novas exceções

A lista de exceções predefinidas na linguagem encontra-se aqui: Index of exceptions.

O programador pode definir novas exceções. A sintaxe da definição duma nova exceção é igual à sintaxe da definição duma variante dum tipos soma.

Exemplos. O primeiro exemplo define uma exceção com argumento; o segundo define uma exceção sem argumento.

```
exception Stack_overflow of string * int ;;
exception I_m_so_out_of_here ;;
```

Funções parciais

Uma **função parcial** é uma função que só está definida em parte do seu domínio. (Não confundir com *aplicação parcial*, que é outra coisa.)

Por exemplo, a função `fact` é parcial porque só está definida para valores não-negativos:

```
let rec fact n = (* pre: n >= 0 *)
    if n = 0 then 1
    else n * fact (n-1)
;;
```

Outro exemplo: A função `maxList` é parcial porque só está definida para listas não-vazias:

```
let rec maxList l = (* pre: l <> [] *)
    match l with
        [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Quando se escreve uma função parcial, espera-se que essa função seja sempre aplicada a argumentos válidos. Mas os programas podem ter erros e é importante que os programas não disfarcem esses erros - é muito melhor a execução dum programar abortar do que terminar produzindo resultados errados. Por isso, convém garantir que a função não produz qualquer resultado quando aplicada a argumentos inválidos (por outras palavras, temos de obrigar o resultado a ser realmente *indefinido*).

A melhor forma de impedir uma função de produzir resultado, ao mesmo tempo gerando uma mensagem de erro clara, é gerar uma exceção. Assim:

```
let rec fact n =  (* pre: n >= 0 *)
    if n = 0 then 1
    else if n > 0 then n * fact (n-1)
    else failwith "fact: negative argument"
;;

let rec maxList l =  (* pre: l <> [] *)
    match l with
         [] -> failwith "maxList: empty list"
       | [x] -> x
       | x::xs -> max x (maxList xs)
;;
```

Mas, repare, mesmo sem lançar exceções explícitas, as versões originais destas duas funções já garantem a não produção de resultados: a primeira aborta com "Stack overflow" e a segunda gera a exceção `Match_failure`.

#120