Linguagens e Ambientes de Programação (2018/2019)

Teórica 07 (27/mar/2019)

Computação, Algoritmos, Programas, Linguagens e Paradigmas.
Funções com múltiplos parâmetros. Formas curried e não-curried. Aplicação parcial de funções curried.
Formas equivalentes de escrita de funções.
Funções como valores de primeira classe.
Funções de ordem superior sobre listas.

Computação

Computação

- Computação significa: processamento automático de informação.
- É a atividade realizada pelos computadores.
- O objetivo da informática é estudar a computação e formas úteis de tirar partido dela para resolver problemas importantes.

Facetas da computação

- **Faceta interna:** a informação é codificada usando símbolos (e.g. bytes), sendo a computação uma atividade de manipulação e transformação automática de símbolos.
- **Faceta externa:** ... mas, geralmente, as computações também estabelecem um diálogo interativo com o ambiente exterior (constituído por humanos e por outras máquinas).

Automatismos e sua especificação

- Qualquer computação é realizada de acordo com regras estabelecidas antes desse processamento se iniciar. É isso que significa o processamento ser automático.
- Daí a necessidade que especificar as computações de forma rigorosa e exaustiva, prevendo todas as eventualidades.

Algoritmos, Programas e Linguagens de Programação

Algoritmo

- Um algoritmo é um conjunto de regras abstratas que determinam, passo a passo, como uma computação vai decorrer.

- Em geral um problema pode ser resolvido usando diversos algoritmos.
- Um algoritmo é independente de qualquer linguagem de programação particular. Podemos imaginá-lo como se fosse para ser executado num computador ideal com memória infinita.

Muhammad ibn Musa al-Khwarizmi

A palavra "algoritmo" deriva da palavra **Algoritmi** que por sua vez corresponde à Latinização do nome do matemático Persa Muhammad ibn Musa al-Khwarizmi, nascido por volta de 780DC. Este cientista, que trabalhou quase toda a sua vida em Bagdad, deu importantes contribuições para a Álgebra, Trigonometria e Aritmética.

Programa

- Um programa é a expressão concreta dum algoritmo.
- Um programa implementa um algoritmo numa linguagem de programação concreta.

Linguagem de programação

- É uma notação para escrever programas.

Exemplo de algoritmo

Algoritmo de Euclides - mdc(m,n)

- Usar dois contadores x e y e inicializar x com m e y com n.
- Se x > y então x recebe x - y
- Se x < y então y recebe y - x
- Repetir os dois passos anteriores até que os valores de x e y fiquem iguais. Quando isso acontecer, esse é o resultado final.

Implementação em C do algoritmo anterior

```
#include <stdio.h>

int main() /* Implementação do algoritmo de Euclides */
{
    int m, n, x, y ;
    printf(">> ") ;
    scanf("%d %d", &m, &n) ;
    x = m ;
    y = n ;
    do {
        if( x > y ) x = x - y ;
        else if( x < y ) y = y - x ;
    } while( x != y ) ;
    printf("mdc(%d,%d) = %d\n", m, n, x) ;
    return 0 ;
}
```

Antiguidade dos Algoritmos

- Os algoritmos mais antigos que se conhecem devem-se aos Babilónios (3000AC-1500AC). Os seus livros de Matemática eram acima de tudo receitas sobre como efetuar os cálculos para resolver determinados problemas. Esses algoritmos eram para executar "à mão".

- A principal motivação para se usarem máquinas para executar algoritmos é a grande velocidade de execução que elas permitem.

Duas questões importantes sobre computação e linguagens

- Existem limites para a computação?
  - Sim, realmente existem problemas com solução para os quais não se consegue inventar qualquer algoritmo que descubra essa solução.

- Existe alguma linguagem de programação que permita implementar todos os algoritmos que se possam imaginar?
  - Sim, qualquer linguagem "normal" permite isso. Do ponto de vista do poder computacional, todas as linguagens normais são equivalentes.

Paradigmas de Programação

Já sabemos que computação significa processamento automático de informação, mas esta noção é demasiado vaga. Podemos interrogar-nos sobre quais os mecanismos concretos através dos quais esse processamento se efetua.

Na verdade, uma linguagem não pode deixar de se comprometer com algum conjunto de mecanismos primitivos para processar informação. Ao efetuar essa escolha, a linguagem adere a um paradigma de programação particular.

É surpreendente a grande variedade de paradigma de programação que têm sido inventados: PARADIGMAS.

Exemplos de paradigmas de programação e respetivos conceitos de base

- **Paradigma imperativo**
  - Conceitos: estado, atribuição, sequenciação
  - Linguagens: Basic, Pascal, C, Assembler.

- **Paradigma funcional**
  - Conceitos: função, aplicação, avaliação
  - Linguagens: Lisp, ML, OCaml, Haskell.

- **Paradigma lógico**
  - Conceitos: relação, dedução
  - Linguagens: Prolog.

- **Paradigma orientado pelos objetos**
  - Conceitos: objetos, mensagem
  - Linguagens: C++, Java, Eiffel.

- **Paradigma concorrente**
  - Conceitos: processo, comunicação (síncrona ou assíncrona)
  - Linguagens: Occam, Ada, Java.

Aspetos práticos

- Cada paradigma de programação determina uma forma particular de abordar os problemas e de formular as respetivas soluções. De facto, diferentes paradigmas de programação representam muitas vezes visões irreconciliáveis do processo de resolução de problemas.

- O grau de sucesso dum programador depende em parte da coleção de paradigmas que domina e da sua arte em escolher o modelo conceptual (paradigma) mais indicado para analisar e resolver cada problema.

Uma linguagem de programação pode combinar mais do que um paradigma

- C++ --- Paradigma imperativo + paradigma orientado pelos objetos.
- Java --- Paradigma imperativo + paradigma orientado pelos objetos + paradigma concorrente.
- OCaml --- Paradigma funcional + Paradigma imperativo + paradigma orientado pelos objetos.
- Ada --- Paradigma imperativo + paradigma concorrente.

Paradigma imperativo *versus* paradigma funcional

Paradigma imperativo

- A computação baseia-se na existência de um "estado" (conjunto de variáveis mutáveis) que vai evoluindo ao longo do tempo.
- O estado é modificado usando um "comando de atribuição" e existe um "operador de sequenciação".
- Linguagens: Basic, Pascal, C, etc., mas o Java também tem uma base imperativa, visto suportar variáveis mutáveis.

Paradigma funcional

- A computação consiste na avaliação de "expressões" nas quais podem ocorrer chamadas de funções.
- O objetivo da avaliação duma expressão é a produção dum "resultado" ou "resposta".
- Não há "atribuição" nem variáveis cujo valor possa ser alterado.
- Também não faz falta uma noção de sequenciação.
- Cada função limita-se a produzir um resultado a partir dos seus argumentos.
- Linguagens: Lisp, OCaml, Haskell, Scheme.

Vantagens do paradigma funcional

- Convida a resolver cada problema com base nas propriedades da solução pretendida e ajuda a encontrar essa solução.

- Suporta programação declarativa, i.e. permite exprimir a lógica das soluções sem ter de se entrar na descrição detalhada dos fluxos de execução.

- O resultado de cada função é exclusivamente determinado pelos valores dos seus argumentos, o que introduz clareza e ausência de complicações. Facilita os testes e os programas ficam mais confiáveis.

- Suporta transparência referencial, uma propriedade que permite trocar uma expressão (e.g. `f()+f()`) por outra com o mesmo resultado teórico (e.g. `2*f()`) sem alterar o comportamento do programa. Tudo o que acontece durante a execução dum programa resulta literalmente do texto do programa, não existindo nenhuma entidade oculta a influenciar a execução.

- As linguagens funcionais modernas rejeitam a noção de **`null`**. Suportar null é um problema porque as variáveis dum programa já validado podem afinal conter null, pondo em causa a validação efetuada. O null indica a inexistência de valor e a forma correta e segura de obter esse efeito em OCaml é usando o tipo **`'a option`** nos pontos em que interessa representar a inexistência de valor - note que para processar um valor de tipo **`'a option`** é obrigatório tratar os dois casos - **`None`** e **`Some(v)`** - o que garante segurança.

- As funções de ordem superior permitem capturar padrões genéricos de código (e.g. funções map, filter, etc.), o que ajuda a programar de forma concisa e sem repetição constante dos mesmos esquemas.

- Sem estado e sem sequencialidade, os programas ficam fáceis de paralelizar.

Desvantagens do paradigma funcional

- É mais difícil controlar detalhes relacionados com a eficiência dos programas.

- Há problemas que envolvem estado de forma essencial, como por exemplo programar uma calculadora com registos de memória. Além disso o mundo exterior é imperativo (tem estado). Numa linguagem funcional pura, o estado precisa de ser simulado usando argumentos suplementares nas funções, o que obscurece o código. Contudo, a generalidade das linguagens funcionais não são puras, ou seja oferecem construções imperativas para lidar com certas situações. Deve é tentar-se minimizar e localizar o uso dessas construções.

Minorar os problemas do estado

Quando se programa com uma linguagem imperativa, há uma forma de minorar as desvantagens do estado. O uso de **estado local privado** é seguro de forma geral. Por exemplo, em Java, recomenda-se o uso de variáveis locais nos métodos, de variáveis de instância privadas nos objetos, e convém evitar todas as outras formas.

Funções com múltiplos argumentos em OCaml

Em OCaml há duas formas de escrever funções com mais do que um argumento: a **forma não-curried** e a **forma curried**. A segunda forma é elaborada mas tem vantagens técnicas.

Forma não-curried

Os vários argumento agrupam-se num único tuplo ordenado. Exemplos:

```
let nAdd (x,y) = x+y ;;
nAdd: (int * int) -> int

let nAdd3 (x,y,z) = x+y+z ;;
nAdd3: (int * int * int) -> int

nAdd (3,4) = 7
nAdd3 (2,8,1) = 11
```

Tecnicamente, a função `nAdd` tem apenas um parâmetro, de tipo `int*int`. Mas claro, através desse único parâmetro conseguimos passar duas unidades de informação.

Forma curried

Os argumentos ficam separados. Exemplos:

```
let cAdd x y = x+y ;;
cAdd: int -> int -> int

let cAdd x y z = x+y+z ;;
cAdd: int -> int -> int -> int

cAdd 3 4 = 7
cAdd 2 8 1 = 11
```

Escrever *funções curried* não tem nada que saber: basta separar os parâmetros por espaços em branco, na cabeça de cada função. Contudo o tipo destas funções afigura-se surpreendente para quem o observa pela primeira vez. Isso é resultado da representação interna das funções em OCaml. Segue-se a explicação...

Representação interna das funções em OCaml

Por uma questão de simplicidade e regularidade de implementação, o OCaml só usa internamente funções anónimas com um único argumento. Uma função com múltiplos argumentos é convertida para um formato interno especial - chamado **forma interna** - que envolve apenas funções anónimas com um único argumento.

Por exemplo, a função

```
let cAdd x y = x+y ;;
```

é internamente convertida em:

```
let cAdd = fun x -> (fun y -> x+y) ;;
```

Repare na engenhosa a ideia que está por detrás do esquema de tradução.

Vejamos mais alguns exemplos de tradução, lado a lado:

```
let cAdd x y z = x+y+z ;;     let cAdd = fun x -> (fun y -> (fun z -> x+y+z)) ;;

let f x = x+1 ;;              let f = fun x -> x+1 ;;

let nAdd (x,y) = x+y ;;       let nAdd = fun (x,y) -> x+y ;;
```

Avaliação de expressões envolvendo funções com múltiplos argumentos

Compare a avaliação das duas seguintes expressões:

```
nAdd (2,3)
= (fun (x,y) -> x+y) (2,3)
= 2 + 3
= 5

cAdd 2 3
= (fun x -> (fun y -> x+y)) 2 3
= (fun y -> 2+y) 3
= 2 + 3
= 5
```

Aplicação parcial

As funções curried têm a vantagem de poderem ser **aplicadas parcialmente** ou seja, poderem ser invocadas omitindo alguns dos argumentos do final. Eis um exemplo de aplicação parcial:

```
let succ = cAdd 1 ;;
succ: int -> int
```

Associatividades

- O **operador de aplicação** é associativo à esquerda. (Note que este operador é *invisível*, pois nunca se escreve).
    - Portanto, a expressão `f a b` deve ser interpretada como `(f a) b`.
- O **construtor de tipos funcionais ->** é associativo à direita.
    - Portanto, o tipo `int -> int -> int` deve ser interpretado como `int -> (int -> int)`.

Formas equivalentes de escrever a mesma função

As seguintes quatro declarações são equivalentes, no sentido em que declaram exatamente a **mesma** função `f:int->int->int`.

```
let f x y = x + y ;;                 (* formato externo preferido *)
let f x = (fun y -> x+y) ;;          (* formato externo *)
let f = (fun x y -> x+y) ;;          (* formato externo *)
let f = (fun x -> (fun y -> x+y)) ;; (* formato interno *)
```

Todas são convertidas para a mesma forma interna.

Funções como valores de primeira classe

Nas linguagens funcionais as funções têm o estatuto de **valores de primeira classe**. Isso significa que as funções têm um estatuto tão importante como o dos inteiros, reais, e outros tipos predefinidos.

Concretamente, numa linguagem funcional as funções podem ...

- Ser passadas como argumento para outras funções;
- Podem ser retornadas por outras funções;
- Podem ser usadas como elementos constituintes de estruturas de dados;
- Têm uma representação literal própria. Exemplo: `(fun x->x+1)`

Exemplos

Exemplo 1. Composição de funções.

```
let compose f g = fun x -> f (g x) ;;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Também se pode escrever:

```
let compose f g x = f (g x) ;;
```

Exemplo 2. Função para converter do formato não-curried para o formato curried.

```
let curry f = fun x -> fun y -> f (x,y) ;;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Também se pode escrever:

```
let curry f x y = f (x,y) ;;
```

Exercício: Escrever a função inversa `uncurry`.

Exercício: Que funções são as seguintes e quais os seus tipos:

- `uncurry compose`
- `compose curry uncurry`
- `compose curry uncurry`

Exemplo 3. Como representar conjuntos usando apenas funções?

Um conjunto é uma entidade cuja principal característica é a possibilidade de se poder saber se um valor lhe pertence ou não lhe pertence. Assim vamos representar cada conjunto por uma função booleana. Essa função, quando aplicada a um valor produz `true` se esse valor pertence ao conjunto e `false` se não pertence. Trata-se da chamada **função característica** do conjunto.

- Conjunto vazio:
  ```
  let set0 = fun x -> false ;;
  ```
- Conjunto universal:
  ```
  let setu = fun x -> true ;;
  ```
- Criação de conjunto singular:
  ```
  let set1 x = fun y -> y = x ;;
  ```
- Teste de pertença:
  ```
  let belongs v s = s v ;;
  ```
- União de conjuntos:
  ```
  let union s1 s2 = fun x -> s1 x || s2 x ;;
  ```
- Interseção de conjuntos:
  ```
  let inter s1 s2 = fun x -> s1 x && s2 x ;;
  ```

Exemplo 4. Estrutura de dados com funções: lista de funções.

```
let mylist = [(fun x -> x+1); (fun x -> x*x)] ;;
```

Uma limitação das funções

No caso geral, saber se duas funções dadas são iguais (i.e. saber se produzem sempre os mesmos resultados para os mesmos argumentos) é um problema que não pode ser resolvido por computador. Numa tal situação, costuma dizer-se que o problema é indecidível.

```
# (fun x -> x+1) = (fun x -> x+1) ;;
Exception: Invalid_argument "equal: functional value".
```

Funções de ordem superior sobre listas

Programar bem numa linguagem funcional envolve dois aspetos:

- Saber usar bem o método indutivo;
- Sabem reconhecer oportunidades de utilização das funções de biblioteca mais importantes.

Algumas das funções de biblioteca de ordem superior sobre listas já incorporam os esquemas mais usados no método indutivo.

Quando possível, é melhor usar essas funções de ordem superior do que programar o método indutivo diretamente. O código resulta mais legível e compacto.

É isso o que fazem os programadores mais experientes.

Função `map`

Aplica uma função `f: 'a -> 'b` a todos os elementos duma lista, para produzir a lista das imagens.

A função `f` define uma relação de um-para-um, pelo que a lista dos resultados tem o mesmo comprimento da lista de entrada.

```
(* map : ('a -> 'b) -> 'a list -> 'b list *)
let rec map f l =
    match l with
        [] -> []
      | x::xs -> (f x) :: map f xs
;;
```

Esta função está disponível na biblioteca do OCaml como **`List.map`**.

Exemplo de chamada:

```
List.map (fun x -> x + 1) [1;2;3]
```

Função `flatMap`

Aplica uma função `f: 'a -> 'b list` a todos os elementos duma lista, para produzir a lista das imagens.

A função `f` define uma relação de um-para-n, pelo que a lista dos resultados tem um comprimento sem qualquer relação com comprimento da lista de entrada.

```
(* flatMap : ('a -> 'b list) -> 'a list -> 'b list *)
let rec flatMap f l =
    match l with
        [] -> []
      | x::xs -> (f x) @ flatMap f xs
;;
```

Exemplo de chamada:

```
flatMap (fun x -> [x + 1; x + 2]) [1;2;3]
```

Esta função não está diretamente disponível na biblioteca do OCaml, pode ser obtida como a combinação de **`List.flatten`** com **`List.map`**. Concretamente, a seguinte definição também é válida.

```
(* flatMap : ('a -> 'b list) -> 'a list -> 'b list *)
let flatMap f l =
    List.flatten (List.map f l)
;;
```

Mais funções de ordem superior sobre listas

`List.exists` `: ('a -> bool) -> 'a list -> bool` (* Testa se pelo menos um dos valores duma lista satisfaz um dado predicado. *)

`List.filter : ('a -> bool) -> 'a list -> 'a list` (* Seleciona os elementos duma lista que satisfazem um dado predicado. *)

`List.partition : ('a -> bool) -> 'a list -> 'a list * 'a list` (* Separa os elementos que satisfazem um predicado dos elementos que não o satisfazem *)

Função de teste de pertença a uma lista ("membership")

`List.mem : 'a -> 'a list -> bool` (* Testa se um valor pertence a uma lista. *)

Exemplo de utilização: Quantificação universal

Verificar se todos os elementos duma lista são pares:

```
let allEven l =
    List.for_all (fun x -> x mod 2 = 0) l
;;
```

Testar se todos os elementos duma lista são menores do que os elementos de outra lista:

```
let allLess l1 l2 =
    List.for_all (fun x ->
          List.for_all (fun y -> x < y) l2) l1
;;
```

#100