Linguagens e Ambientes de Programação (2018/2019)

Teórica 08 (01/abr/2019)

Funções recursivas. Usando o método indutivo.

O método indutivo aplicado à resolução de problemas sobre: inteiros, listas, árvores binárias, árvores n-árias, ficheiros, strings.

Categorias de funções recursivas: funções de categoria 1, 2, 3 e 4.

Funções recursivas (indutivas)

Uma função recursiva bem definida faz sempre uma *análise de casos* sobre os seus parâmetros. Por exemplo, a função "fib" considera três casos relativamente ao seu parâmetro "n":

```
let rec fib n =  
  if n = 0 then 1  
  else if n = 1 then 1  
  else fib (n-1) + fib (n-2).  
;;
```

Existem sempre dois géneros de casos que uma função recursiva tem sempre de tratar: *casos base* e *casos gerais* (ou *casos indutivos*). Os **casos base** são aqueles que não conduzem, nem direta nem indiretamente, a chamadas recursivas da função; os **casos gerais** são aqueles que conduzem, direta ou indiretamente, a chamadas recursivas da função. A função "fib" trata dois casos base, $n=0$ e $n=1$, e trata um caso geral, $n \geq 2$.

Toda a função recursiva deve tratar pelo menos um caso base e um caso geral. Além disso todas as chamadas recursivas que estão associadas aos casos gerais devem corresponder **problemas mais simples** do que o problema que a função, no seu todo, resolve (*problema mais simples* significa *problema mais próximo de algum dos casos base*). Apenas este conjunto de condições garante terminação.

Para nós, **função indutiva** é apenas outro nome para uma função recursiva definida nos termos atrás descritos.

Usando o método indutivo

O **método indutivo**, introduzido na aula teórica 3, ajuda a programar os casos indutivos das funções recursivas (indutivas). Concretamente, dá alguma orientação sobre como fazer a *redução de problemas a problemas mais simples*.

Quando se usa o método indutivo, **devem-se ter em mente** apenas as propriedades lógicas do problema a resolver. **Não se devem ter em mente** quaisquer preocupações relacionadas com as propriedades operacionais da função que está a ser programada. Misturar do uso do método indutivo com preocupações de natureza operacional é uma receita para o desastre.

De qualquer forma, depois de programada a função, já não há problema em tentar compreendê-la operacionalmente. Pode mesmo valer a pena escrever uma avaliação da função para se ganhar confiança

nela. Exemplo: avaliação de (len [1;5;3;8]):

```
len [1;5;3;8] =
  1 + len [5;3;8] =
    1 + 1 + len [3;8] =
      1 + 1 + 1 + len [8] =
        1 + 1 + 1 + 1 + len [] =
          1 + 1 + 1 + 1 + 0 =
            4
```

Categorias de funções recursivas

Vamos agora estudar 4 categorias de funções recursivas, todas programadas usando o método indutivo: funções de categoria 1, de categoria 2, de categoria 3 e de categoria 4.

Na avaliação da qualidade duma função, um dos critérios que surge em primeiro lugar é o da **facilidade em compreender a função**. As funções de categoria 1 e 2 tendem a ser as mais simples de perceber, mas nem todos os problemas podem ser resolvidos diretamente usando apenas funções de categoria 1 e 2.

Funções de categoria 1

As **funções de categoria 1** são programadas usando o método indutivo e caracterizam-se pela seguinte propriedade:

- A estrutura da função baseia-se num dos esquemas rígidos predefinidos, indicados a seguir. Repare que em todos esses esquemas, os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.

Nesses esquemas, o PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR aparece sublinhado.

Orientação prática: Quando se tenta resolver um problema, convém começar por procurar construir uma função de categoria 1 já que estas são as mais fáceis de escrever e compreender, na maioria dos casos.

Inteiros

```
let rec f n =
  if n = 0 then ...
  else ... f(n-1) ...
;;
```

Listas

```
let rec f l =
  match l with
  [] -> ...
  | x::xs -> ... f xs ...
;;
```

Árvores binárias

```
let rec f t =
  match t with
  Nil -> ...
  | Node(x,l,r) -> ... f l ... f r ...
;;
```

Árvores n-árias

```

let rec f t =
  match t with
  | NNil -> ...
  | NNode(x,cs) -> ... lf cs ...
and lf tl =
  match tl with
  | [] -> ...
  | t::ts -> ... f t ... lf ts ...
;;

```

Ficheiros: leitura de sequência

```

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

```

Strings

```

let cut s = (String.get s 0, String.sub s 1 ((String.length s)-1)) ;;
let rec f s =
  if s = "" then ...
  else
    let (x,xs) = cut s in
    ... f xs ...
;;

```

Exemplos de funções de categoria 1: fact, len, append, rev, belongs, union, power, height, size, zeros, treeToList, balanced, subTrees, etc. (em suma, a maioria das funções estudadas até ao momento).

Mais exemplos de funções de categoria 1:

stringAsList: converte string em lista de caracteres

```

let cut s = (String.get s 0, String.sub s 1 ((String.length s)-1)) ;;
let rec stringAsList s =
  if s = "" then []
  else
    let (x,xs) = cut s in
    x::stringAsList xs
;;

```

sortList: ordena lista

```

let rec insOrd v l =
  match l with
  | [] -> [v]
  | x::xs ->
    if v <= x then v::x::xs
    else x::insOrd v xs
;;
let rec sortList l =
  match l with
  | [] -> []
  | x::xs ->
    insOrd x (sortList xs)
;;

```

Funções de categoria 2

As **funções de categoria 2** são programadas usando o método indutivo e caracterizam-se pelas duas seguintes propriedades:

- Os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.
- Fogem aos esquemas rígidos das funções de categoria 1.

Orientação prática: Quando se tenta escrever uma função de categoria 1, por vezes descobre-se que é necessário ou conveniente fazer alguns pequenos ajustamentos ao esquema rígido de base. Desta forma somos levados a inventar uma função de categoria 2. O PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR envolve um pouco de descoberta mas, em geral, é fácil de descobrir esse ponto de partida pois os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.

Exemplos de funções de categoria 2: `maxList`, `fall`, `fib`.

Mais exemplos de funções de categoria 2:

halfHalf: reparte os elementos numa lista por duas listas, alternadamente (esta versão é programada com base na ideia de processar os elementos da lista dois a dois)

```
let rec halfHalf l =
  match l with
  | [] -> ([],[])
  | [x] -> ([x],[])
  | x::y::xs ->
      let (us,vs) = halfHalf xs in
      (x::us, y::vs)
;;
```

addEvenPos: soma todos os elementos numa lista que estão em posições de índice par (0, 2, 4, ...)

```
let rec addEvenPos l =
  match l with
  | [] -> 0
  | [x] -> x
  | x::_:xs ->
      x + addEvenPos xs
;;
```

Funções de categoria 3

As **funções de categoria 3** são programadas usando o método indutivo e caracterizam-se pelas seguinte propriedade:

- A chamada recursiva envolve expressões que *não são* subpadrões dos padrões que representam os argumentos da função. As expressões que constituem os argumentos da chamada recursiva são *calculados*.

Orientação prática: As funções de categoria 3 surgem geralmente quando se tenta resolver um problema aplicando uma estratégia previamente pensada, em vez de se tentar encontrar uma solução baseada nos esquemas predefinidos que caracterizam as funções de categoria 1.

Precaução: Programar uma função de categoria 3 em vez de uma de categoria 1 ou 2 significa, quase sempre, **complicar desnecessariamente**. Além disso, as funções de categoria 3 são mais difíceis de perceber do que as funções de categoria 1 ou 2. Em todo o caso, ocasionalmente surge uma boa razão para se escrever uma função de categoria 3: por exemplo, o aumento da eficiência (se for caso disso, porque por vezes a função fica menos eficiente).

Exemplo de função de categoria 3:

quickSort: ordena lista eficientemente (reduz-se o tratamento de uma lista ao tratamento de duas secções dessa lista)

```
let rec partition p l =
  match l with
  | [] -> ([],[])
  | x::xs ->
    let (a,b) = partition p xs in
    if x <= p then (x::a,b) else (a, x::b)
;;
let rec quickSort l =
  match l with
  | [] -> []
  | x::xs ->
    let (us,vs) = partition x xs in
    (quickSort us) @ [x] @ (quickSort vs)
;;
```

minSort: ordena lista usando o algoritmo de seleção direta

```
let rec removeFromList v l =
  match l with
  | [] -> []
  | x::xs ->
    if x = v then xs
    else x::removeFromList v xs
;;
let rec minList l =
  match l with
  | [x] -> x
  | x::xs -> min x (minList xs)
;;
let rec minSort l =
  match l with
  | [] -> []
  | list ->
    let m = minList list in
    m::minSort (removeFromList m list)
;;
```

A função `quickSort` permite ganhar eficiência. Já a função `minSort` é muito mais complicada do que a função de categoria 1 `sortList` e não é mais eficiente.

Funções de categoria 4

Uma **função de categoria 4** é uma função não recursiva que se define à custa de funções auxiliares de categoria 1, 2, ou 3.

Orientação prática: Existem problemas que não podem ser resolvidos diretamente usando o método indutivo. Nestes casos surge a necessidade de escrever uma função de categoria 4. Quase sempre é preciso inventar um novo problema que já se consiga resolver usando o método indutivo e que ajude a resolver o problema original.

Exemplos de funções de categoria 4:

prime: determina de um inteiro é ou não primo

```
let rec hasDiv n a z =
  if a > z then false
  else (n mod a = 0) or (hasDiv n (a+1) z)
;;
let prime n =
  n > 1 && not (hasDiv n 2 (n-1))
;;
```

width: determina a largura duma árvore binária

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;

let rec maxList l =
  match l with
  [x] -> x
  | x::xs -> max x (maxList xs)
;;
let rec sumLists l1 l2 =
  match l1,l2 with
  [],l -> l
  | l,[] -> l
  | x::xs, y::ys -> (x+y)::sumLists xs ys
;;
let rec levels t =
  match t with
  Nil -> []
  | Node(x,l,r) -> 1::sumLists (levels l) (levels r)
;;
let width t =
  if t = Nil then 0
  else maxList (levels t) ;;
```

Os dois problemas anteriores não são indutivos. Porquê?

#100