Linguagens e Ambientes de Programação (2018/2019)

Teórica 03 (13/mar/2019)

O método indutivo (redução de problemas a problemas mais simples).
Muitas funções sobre listas programadas usando o método indutivo.
Nomes locais. Funções mutuamente recursivas.

Método indutivo

Revisitando a a recursividade

A programação de funções recursivas não é novidade pois já foi abordada em cadeiras anteriores, por exemplo POO e AED. Contudo, o estudo duma linguagem funcional é uma excelente oportunidade para revisitar o tema, com os seguintes objetivos: aprofundar a análise do mecanismo; estudar diversos padrões de utilização; passar a encarar a recursividade como uma técnica que ajuda a descobrir soluções para certos problemas complicados.

Numa linguagem funcional, a recursividade está na base da programação de todas as atividades repetitivas, já que não existe o mecanismo da iteração, típico das linguagens de base imperativa.

Além disso, nós queremos usar a recursividade duma forma que enfatize as propriedades lógicas do problema a resolver em vez de questões operacionais. O objetivo é programar a mais alto nível, a um nível mais humano, evitando ter de pensar como uma máquina.

Redução de problemas a problemas mais simples

Considere as seguintes duas funções recursivas, escritas em OCaml:

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

let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

Analisando estas duas funções recursivas verificamos que quando lidam com o caso não-trivial - o chamado **caso geral** - ambas efetuam a **redução do problema original a uma instância mais simples do mesmo problema**. Concretamente, a função `len` reduz o problema `len (x::xs)` ao problema mais simples `len xs`, e a função `fact` reduz o problema `fact n` ao problema mais simples `fact (n-1)`.

Além de efetuarem *redução de problemas* no *caso geral*, as duas funções lidam também, separadamente, com o caso trivial de cada problema - o chamado **caso base**. Concretamente, a função `len` trata separadamente o caso base `len []` e a função `fact` trata separadamente o caso base `fact 0`.

Repare que quando uma destas duas funções é aplicada a um argumento, inicia-se uma sucessão de *reduções a problemas mais simples* que só termina quando o caso base é alcançado. Por exemplo, a sequência de *reduções* produzida pela aplicação `len [7;6;5;4]` é a seguinte:

```
len [7;6;5;4]              <-- problema inicial
= 1 + len [6;5;4]          <-- problema um pouco mais simples
= 1 + 1 + len [5;4]        <-- problema ainda mais simples
= 1 + 1 + 1 + len [4]      <-- problema ainda mais simples
= 1 + 1 + 1 + 1 + len []   <-- caso base
= 1 + 1 + 1 + 1 + 0
= 4
```

As funções recursivas `len` e `fact` são representativas do que é programar usando o núcleo funcional do OCaml. Em OCaml programa-se essencialmente *reduzindo problemas a problemas mais simples*.

Técnica do método indutivo

O método indutivo (também conhecido como técnica da divisão e conquista) serve para ajudar a programar funções recursivas que efetuam *reduções de problemas a problemas mais simples*.

O método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA (para começar a raciocinar) que o resultado S de aplicar a função a argumentos *mais simples do que os iniciais* (por exemplo a cauda da lista-argumento) já se encontra disponível.
- O problema para resolver é o seguinte: COMO SE PASSA DE S PARA G?

Exemplo de utilização do método indutivo

Problema: Programar uma função `len` que determine o comprimento de listas.

Resolução: Para começar, a função `len` recebe uma lista, a qual terá de ser processada usando emparelhamento de padrões. Aplicando o método indutivo ao caso geral `G=len (x::xs)`, vamos começar por assumir que já temos o problema resolvido para o caso mais simples `S=len xs`.

Desde já podemos ir escrevendo a seguinte estrutura incompleta:

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

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de `G=len (x::xs)` a partir do valor de `S=len xs`?* Mas este problema é simples! De facto, basta adicionar uma unidade a S para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

```
let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

NOTA1: Neste exemplo, reduzimos o problema `G=len (x::xs)` ao problema `S=len xs`. Mas esta não é a única redução possível... Existem muitas outras... Voltaremos a este assunto noutra aula.

NOTA2: A função `len` é tão simples que faz o método indutivo parecer óbvio e desinteressante. No entanto, este método de resolver problemas tem imensas virtualidades:

- Ajuda-nos a organizar o pensamento quando queremos resolver problemas mais complicados, e.g. funções `power` e `sort`.
- Simplifica a resolução de problemas. Repare que já não temos de resolver "o problema completo": passamos a ter de resolver "apenas" o problema da passagem de S para G, o que costuma ser "muito mais fácil".
- Ajuda-nos a descobrir soluções simples e compactas.

Mais funções sobre listas programadas usando o método indutivo

Estude-as todas de forma muito atenta. Depois tente programá-las todas "sem espreitar".

Comprimento de lista:

```
len : 'a list -> int

let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

Soma dos elementos de lista:

```
sum : int list -> int

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

Concatenação de listas:

```
append : 'a list -> 'a list -> 'a list

let rec append l1 l2 =
    match l1 with
        [] -> l2
      | x::xs -> x::append xs l2
;;
```

Acrescentar elemento no final de lista:

```
putAtEnd : 'a -> 'a list -> 'a list

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

Inversão de lista:

```
rev : 'a list -> 'a list

let rec rev l =
    match l with
        [] -> []
      | x::xs -> append (rev xs) [x]
;;
```

Máximo duma lista (tratamento implícito do erro):

```
maxList : 'a list -> 'a

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

Máximo duma lista (tratamento explícito do erro):

```
maxList : 'a list -> 'a

let rec maxList l =  (* pre: l <> [] *)
    match l with
        [] -> raise (Arg.Bad "maxList: lista vazia")
      | [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Aplicação de função a todos os elementos de lista:

```
map : ('a -> 'b) -> 'a list -> 'b list

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

Seleção dos elementos duma lista que verificam uma dada propriedade:

```
filter : ('a -> bool) -> 'a list -> 'a list

let rec filter b l =
    match l with
        [] -> []
      | x::xs ->
            if b x then x::filter b xs
            else filter b xs
;;
```

Concatenação de todas as listas contidas numa lista de listas:

```
flatten : 'a list list -> 'a list

let rec flatten ll =
    match ll with
        [] -> []
      | l::ls ->
            l @ flatten  ls
;;
```

Concatenação de todos os resultados gerados por uma função de mapping que produz listas:

```
flatMap : ('a -> 'b list) -> 'a list -> 'b list

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

Ordenação de listas:

```
sortList : 'a list -> 'a list

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)
;;
```

Nota: Repare que não tinhamos pensado em nenhum algoritmo de ordenação em particular e que o método indutivo nos levou a inventar um algoritmo de forma não deliberada.

Fusão de listas ordenadas sem repetições (aqui dá jeito usar matching duplo):

```
fusion : 'a list -> 'a list -> 'a list

let rec fusion l1 l2 =
    match l1, l2 with
        _, [] -> l1
      | [], _ -> l2
      | x::xs, y::ys ->
          if x < y then x::fusion xs l2
          else if y < x then y::fusion l1 ys
          else x::fusion xs ys
;;
```

Exemplos de avaliações

```
sum [1;2;3]
= 1 + sum [2;3]
= 1 + 2 + sum [3]
= 1 + 2 + 3 + sum []
= 1 + 2 + 3 + 0
= 6

append [1;2;0] [3;4]
= 1::append [2;0] [3;4]
= 1::2::append [0] [3;4]
= 1::2::0::append [] [3;4]
= 1::2::0::[3;4]
= [1;2;0;3;4]

rev [1;2;3]
= (rev [2;3]) @ [1]
= (rev [3]) @ [2] @ [1]
= (rev []) @ [3] @ [2] @ [1]
= [] @ [3] @ [2] @ [1]
= [3;2;1]
```

A função @

A função `append` está predefinida em OCaml, sendo denotada pelo operador binário `@`. Assim, podemos escrever:

```
[1;2;3] @ [4;5;6] = append [1;2;3] [4;5;6] = [1;2;3;4;5;6]
```

No entanto, a função `@` é pouco eficiente e deve ser usada com critério. Por exemplo, para acrescentar um zero à cabeça duma lista `list` é má ideia escrever `[0]@list`; escreve-se sempre `0::list`. No entanto, para acrescentar um zero ao final duma lista `list` não temos alternativa e escreve-se mesmo `list@[0]`.

Nomes locais

Agora, um novo assunto...

Construção let-in

A construção **let-in**:

```
let n = exp in
    exp1
```

associa o nome `n` ao valor da expressão `exp` no contexto da expressão `exp1`. O nome `n` passa a definir uma **constante local** à expressão `exp1`.

O nome `n` também pode ser parametrizado, passando a denotar uma **função local**, assim

```
let n x = exp in
    exp1
```

Esta forma é equivalente a

```
let n = (fun x -> exp) in
    exp1
```

A palavra reservada **and** pode ser usada para definir simultaneamente vários nomes no mesmo `let-in`. Por exemplo:

```
let a = 1 and b = 2 in
    a + b ;;
```

ou

```
let f x = x + 1 and g x = x + 2 in
    f (g x) ;;
```

Exercício: Procure no manual da linguagem a forma sintática geral da construção **let-in**.

Vamos ver de seguida que a construção **let-in** permite:

1. Aumentar a legibilidade de certas funções;
2. Aumentar a eficiência de certas funções;
3. Definir funções mutuamente recursivas.

1. Legibilidade

Considere o seguinte problema: Escreva em OCaml uma função chamada **parque** que calcule o preço a pagar no parque de estacionamento subterrâneo dos Restauradores, em Lisboa. A função `parque` recebe 2 pares ordenados de inteiros como argumentos: hora e minuto de entrada, hora e minuto de saída. Os tempos de permanência são arredondados para horas completas e sempre para cima. Assuma que o carro nunca está no parque mais do que 24:00. Em Março de 1999, a tabela de preços era a seguinte:

```
1ª hora        120 cêntimos
2ª hora        140 cêntimos
```

```
3ª hora        150 cêntimos
seguintes      155 cêntimos
```

Apresentam-se a seguir duas versões equivalentes da função `parque`, a segunda programada usando **let-in**. A escolha entre elas é, em grande medida, uma questão de gosto, mas pode argumentar-se que a segunda versão, apesar de mais longa, é mais fácil de perceber porque torna explícita a ordem de avaliação das subexpressões dentro duma expressão que é demasiado grande e complexa.

```
let parque (he, me) (hs, ms) =
    pagar (convHoras (permanencia (convMinutos he me) (convMinutos hs ms)))
;;

let parque (he, me) (hs, ms) =
    let te = convMinutos he me in
        let ts = convMinutos hs ms in
            let perm = permanencia te ts in
                let h = convHoras perm in
                    pagar h
;;
```

Exercício: As funções auxiliares não foram ainda programadas. Tente escrevê-las.

2. Eficiência

A construção **let-in** também pode ser usada para aumentar a eficiência de certas funções. Por exemplo a segunda função é mais eficiente do que a primeira (porquê?):

```
let f g x =
    g x + g x * g x
;;

let f g x =
    let r = g x in
        r + r * r
;;
```

3. Funções mutuamente recursivas

A forma geral da construção **let-in** inclui a possibilidade de utilização simultânea de **rec** e **and**. Isso permite definir 2 ou mais **funções mutuamente recursivas**. Eis um exemplo curioso com duas funções:

```
    let rec even n = if n = 0 then true else odd (n-1)
         and odd n = if n = 0 then false else even (n-1) in
             exp
;;
```

O que fazem estas funções?

Tradução do let-in

A construção **let-in**

```
let n = exp in
    exp1
```

é internamente traduzida para a seguinte representação:

```
(fun n -> exp1) exp
```

As duas formas são equivalentes (medite um pouco nisso!), mas a primeira é mais fácil de perceber.

Tradução do let global

A já nossa conhecida construção **let-global**, que permite definir nomes globais, é internamente traduzida para a construção **let-in** - os nomes globais são habilidosamente traduzidos para nomes locais.

Por exemplo, a seguinte sequência de definições e expressões:

```
# let f x = x + 1 ;;
# let g x = x + 2 ;;
# f (g x) ;;
```

é internamente traduzida para uma sequência de **let** aninhados:

```
let f x = x + 1 in
    let g x = x + 2 in
        f (g x) ;;
```

#140