

Teoria da Computação - Programação OCaml

Ficha de exercícios

Simão Melo de Sousa

Ano lectivo 2005/2006

Em parte, estes exercícios baseam-se nas fichas práticas retiradas do *site* OCaml Hump

1 Aritmética

Exercício 1 (Fibonacci) *O objectivo deste exercício é escrever um programa Ocaml que permita calcular para um dado $n \in \mathbb{N}$ o valor de $Fib(n)$, tendo em conta que :*

$$\begin{cases} Fib(0) = 1 \\ Fib(1) = 1 \\ Fib(n + 2) = Fib(n) + Fib(n + 1) \end{cases}$$

1. *Numa primeira abordagem, escreva um programa que peça o valor de n via menu. O calculo deverá ser feito de forma recursiva;*
2. *modifique o seu programa de forma a que o valor de n seja dado pela linha de comando;*
3. *modifique o seu programa de forma a que o valor de n seja extraído de um ficheiro chamado `dados.txt`;*
4. *dê uma versão iterativa do calculo de $Fib(n)$;*
5. *dê uma versão recursiva terminal do calculo de $Fib(n)$;*

□

Exercício 2 (Factorial)

1. *Repita o exercício anterior mas desta vez com a definição da função factorial;*
2. *Utilize a biblioteca `Num` para permitir o cálculo de grande factoriais.*

□

Exercício 3 (Euclides) *Escreva um programa OCaml que, usando o algoritmo de Euclides, permita calcular o maior divisor comum de dois números. Relembra-se que o método é o seguinte:*

$$\begin{cases} MCD(x, y) = MCD(y, \text{modulo}(x, y)) \\ MCD(x, 0) = x \end{cases}$$

□

Exercício 4 (Números primos) *Escreva uma função OCaml recursiva, recursiva terminal e uma função OCaml iterativa para testar se um dado inteiro é primo ou não. Relembra-se que um inteiro $n > 1$ é primo se, e só se, é divisível por ele próprio e por 1.*

□

Exercício 5 (Funções mutuamente recursivas) *Sejam $\text{par}(n)$ a função que devolve true se n for par e false no caso contrário e $\text{impar}(n)$ a função que devolve true se n for ímpar e false no caso contrário.*

1. Defina as funções par e impar de forma mutuamente recursiva;
2. dê definições alternativas a estas duas funções sem utilizar a recursividade mútua.

□

Exercício 6 (Binômio de Pascal) *Para $(n, p) \in \mathbb{N}^2$ tal que $(0 \leq p \leq n)$, a função C_p^n pode ser calculada usando a definição recursiva seguinte:*

$$\begin{cases} C_n^0 = 1 \\ C_n^n = 1 \\ C_p^n = C_p^{n-1} + C_{p-1}^{n-1} \quad \text{se } (0 < p < n) \end{cases}$$

Escreva a função recursiva OCaml correspondente.

□

Exercício 7 (Números de Catalan) *Uma famosa sequência de números inteiros conhecida por Números de Catalan é definida por:*

$$\text{Catalan}(n) = \begin{cases} 1 & \text{se } n = 0 \wedge n = 1 \\ \sum_{(p,q) \text{ tais que } p+q=n-1} \text{Catalan}(p) * \text{Catalan}(q) & \text{se } n > 1. \end{cases}$$

1. Defina uma função OCaml que calcula $\text{Catalan}(n)$ para um n inteiro. Tente, em particular, utilizar funções de ordem superior;
2. analise o comportamento computacional de tal função. Em particular, dado um par $(q, p) \in \mathbb{N}^2$, quantas vezes é calculado o produto $\text{Catalan}(p) * \text{Catalan}(q)$?
3. melhore a sua função de acordo com a análise feita no ponto anterior.

□

2 Manipulação de listas

Exercício 8 (Método de Horner) Podemos representar um polinómio P de grau n por uma lista p de reais em o i -ésimo elemento da lista representa o coeficiente associado à potência de grau i . Escreva um programa que peça o valor de n (com a restrição que $n \geq 0$), inicialize P e que dado um x , calcule $P(x)$ usando o método de Horner, i.e.

$$P_n(x) = (\cdots((p_n x + p_{n-1})x + p_{n-2})x + \cdots + p_1)x + a_0$$

□

Exercício 9 Defina as seguintes funções sobre listas:

1. `val pelo_menos_dois : 'a list → 'a list`

2. `val exactamente_dois : 'a list → 'a list`

A primeira função devolve a lista dos elementos que aparecem pelo menos duas vezes na lista em parâmetro. A segunda função devolve a lista dos elementos que aparecem exactamente duas vezes.

□

Exercício 10 (Conjuntos) Considere que se codifique a noção de conjuntos como listas sem repetição dos elementos.

```
1 type 'a conjunto = 'a list
2 and 'a familia = 'a conjunto list
```

Implemente então as seguintes funções:

```
1 val elemento : 'a → 'a conjunto → bool
2 val conjunto_vazio : 'a conjunto
3 val subconjunto : 'a conjunto → 'a conjunto → bool
4 val igual : 'a conjunto → 'a conjunto → bool
5 val vazio : 'a conjunto → bool
6 val subconjunto_proprio : 'a conjunto → 'a conjunto → bool
7 val normalizar : 'a conjunto → 'a conjunto
8 val forma_normal : 'a conjunto → bool
9 val uniao : 'a conjunto → 'a conjunto → 'a conjunto
10 val interseccao : 'a conjunto → 'a conjunto → 'a conjunto
11 val diferenca : 'a conjunto → 'a conjunto → 'a conjunto
12 val diferenca_simetrica : 'a conjunto → 'a conjunto → 'a conjunto
13 val complemento : 'a conjunto → 'a conjunto → 'a conjunto
14 val partes : 'a conjunto → 'a conjunto conjunto
15 val produto_cartesiano : 'a conjunto → 'b conjunto → ('a * 'b) conjunto
16 val interseccao_fam : 'a familia → 'a conjunto
17 val uniao_fam : 'a familia → 'a conjunto
18 val diferenca_fam : 'a familia → 'a conjunto
19 val diferenca_simetrica_fam : 'a familia → 'a conjunto
20 val particao : 'a familia → 'a conjunto → bool
21 val aplica_operacao : ('a → 'b) → 'a conjunto → 'b conjunto
22 val aplica_operador_n : int → ('a → 'a) → 'a → 'a
23 val aplica_operador : ('a conjunto → 'a conjunto) → 'a conjunto → 'a conjunto
24 val calcula_ponto_fixo : ('a conjunto → 'a conjunto) → 'a conjunto
25 val fechado : ('a conjunto → 'a conjunto) → 'a conjunto → bool
```

□

Exercício 11 (Relações) *Considere que se codifique a noção de relação binária da seguinte forma:*

```
1 type ('a, 'b) relacao_bin = ('a * 'b) conjunto
2 and 'a relacao = ('a, 'a) relacao_bin
```

Implemente então as seguintes funções:

```
1 val dominio : ('a, 'b) relacao_bin → 'a conjunto
2 val codominio : ('a, 'b) relacao_bin → 'b conjunto
3 val reflexiva : 'a relacao → bool
4 val irreflexiva : 'a relacao → bool
5 val simetrica : 'a relacao → bool
6 val antisimetrica : 'a relacao → bool
7 val transitiva : 'a relacao → bool
8 val composicao :
9   ('a, 'b) relacao_bin → ('b, 'c) relacao_bin → ('a, 'c) relacao_bin
10 val relacao_igualdade : 'a conjunto → 'a relacao
11 val potencia : int → 'a relacao → ('a, 'a) relacao_bin
12 val fecho_transitivo : 'a relacao → 'a relacao
13 val fecho_reflexivo : 'a relacao → 'a relacao
14 val fecho_simetrico : 'a relacao → 'a relacao
15 val inverso : 'a relacao → 'a relacao
16 val ordem_parcial : 'a relacao → bool
17 val mais_fraca : 'a relacao → bool
18 val mais_forte : 'a relacao → bool
19 val quasi_ordem : 'a relacao → bool
20 val cadeia : 'a relacao → bool
21 val ordem_linear : 'a relacao → bool
22 val equivalencia : 'a relacao → bool
```

□

3 Estruturas, Vectors, Referências e Programação imperativa

Exercício 12 (Horner - bis) *Apresente uma resolução do método de Horner usando vectores.* □

Exercício 13 *Defina a função de ordenação por bubble-sort sobre vectores* □

Exercício 14 (Lista ligadas)

1. *Defina em OCaml o tipo das listas ligadas do paradigma imperativo;*
2. *defina as funções habituais sobre este tipo de dado.*

□

4 Manipulação de strings, Input/Output e Ficheiros

Exercício 15 Defina as seguintes funções OCaml:

1. `val reverse : string → string` (devolve a inversa da string em parâmetro)
2. `val lowercase : string → string` (devolve uma cópia do seu argumento em que todas as letras maiúsculas foram transformadas em minúsculas)
3. `val capicua : string → bool` (testa se uma string é uma capicua)
4. `val strstr : string → string → int` (verifica se o primeiro parâmetro está contido no segundo)
5. `val compare : string → string → int` (compare a $b = \begin{cases} -1 & \text{se } a < b \\ 0 & \text{se } a = b \\ 1 & \text{se } a > b \end{cases}$)

□

Exercício 16 Escreva em OCaml um clone do programa `wc` (word count) do sistema UNIX. Este programa deverá aceitar as opções `-l -c -w` do programa original □

5 Tipos indutivos, Filtragem, Algoritmos e Estruturas de dados

Exercício 17 (calculadora de expressões aritméticas) Considere a pequena linguagem aritmética seguinte:

```
1 <expr> ::= let <identificador> = <expr> (*variável global*)
2         | let <identificador> = <expr> in <expr> (*variável local*)
3         | <inteiro>
4         | <identificador>
5         | <expr> + <expr>
6         | <expr> * <expr>
7         | <expr> - <expr>
8         | <expr> / <expr>
```

Uma calculadora para esta linguagem terá de interpretar qualquer string contendo uma expressão aritmética num valor inteiro. Para tal terá de transformar a string num valor de uma estrutura de dados que retrata esta linguagem aritmética. Seja `expr` este tipo de dados. Esta fase da interpretação é conhecida por parsing. A operação contrária que transforma um valor do tipo `expr` numa string é chamada de pretty printing. O cálculo propriamente dito é realizado por recursão sobre valores do tipo `expr`.

1. Defina o tipo `expr` das expressões aritméticas;
2. defina a função `pretty_print_expr` que transforma um valor do tipo `expr` em string;

3. defina a função `val eval_expr: expr → (identificador*int) → (identificador*int) → int` que calcule o valor da expressão aritmética em parâmetro. Os dois últimos argumentos arquivam o ambiente global (onde são arquivados as variáveis globais e os seus valores) e o ambiente local (onde são arquivados as variáveis locais e os seus valores). A função `eval_expr` deverá ser capaz de detectar divisões por zero e utilização de variáveis não inicializadas;
4. assumindo que exista uma função `parsing: string → expr` que realize a fase de parsing a partir de uma `string`, escreva um programa que lê, da entrada standard e linha a linha, expressões aritméticas e calcule os respectivos valores. O programa deverá ter em conta a adequada gestão dos ambiente de variáveis globais e locais.

□

Exercício 18 (Avaliação de Fórmulas Lógicas Proposicionais) Consideramos aqui a avaliação de fórmulas da lógica proposicional. Na lógica das proposições, uma fórmula é constituída pelas conectivas $\wedge, \vee, \neg, \Rightarrow$ e \Leftrightarrow , pelas constantes \top e \perp e por variáveis proposicionais.

1. Defina o tipo das fórmulas da lógica proposicional;
2. utilizando o método das tabelas de verdade, escreva a função OCaml `val eval : prop → bool` que verifica se o seu parâmetro é uma tautologia (ou não).

□

Exercício 19 (Árvores Binárias)

1. Defina o tipo das árvores binárias polimórficas.
2. Defina as funções `tamanho`, `altura`, `pertence`, `percurso_largura` e `percurso_profundidade`
3. Defina as funções `map` e `fold_left` sobre árvores binárias
4. Reescreva as funções do segundo ponto usando o `map` e o `fold`

□

Exercício 20 (Árvores Binárias de Pesquisa) Repita o exercício anterior, mas desta vez com as árvores binárias de pesquisa (ABP). Defina igualmente as funções `add` e `remove` □

Exercício 21 (Árvores Binárias de Pesquisa Equilibradas) Uma árvore binárias de pesquisa equilibradas (ABPE) é uma ABP em que para qualquer nodo a , a altura da sua sub-árvore direita difere da altura da sua sub-árvore esquerda no máximo de um valor. Redefina o tipo e as funções do exercício por forma a implementar as ABPE □

Exercício 22 (árvore n -árias)

1. Generalize a noção de árvore binária para árvore n -árias;

2. adapte as funções do exercício 19

□

Exercício 23 (Grafos e Algoritmos) *Um grafo dirigido com label nos vértices pode ser representado pelo tipo seguinte:*

```
1 type ('a,'b) digrafo = ('a*'b*'a) list
```

onde 'a é o tipo dos nodos e 'b é o tipo dos vértices. Assim (1,'a'',2) significa que um vértice com a informação ''a'' sai do nodo 1 para o nodo 2.

Defina as funções seguintes:

1. *conexo a b ar* que verifica se o nodo a está na mesma componente conexa que b na árvore ar;
2. *mais_curto a b ar* que devolve o caminho mais curto (em termos de nodos envolvidos) entre a e b na árvore ar;
3. *mais_leve a b ar* que devolve o caminho mais leve (onde a informação dos vértices tem o papel de peso) entre a e b na árvore ar;
4. *Considere agora grafos não dirigidos. A semelhança do tipo digrafo, defina o tipo destes grafos;*
5. *defina a função `spanning_tree a ar` que calcule a árvore de cobertura de raiz a da componente conexa contendo o nodo a.*

□