Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Lectivo 201X/201X
Linguagens e Ambientes Programação – Solução Exame de Recurso

21 de Junho de 201X às 09:00

Exame com consulta com 2 horas e 45 minutos de duração + 15 minutos de tolerância

Nome: Num:

Notas: *Este enunciado é constituído por 6 grupos de perguntas. Responda no próprio enunciado, usando a frente e o verso.*
Nos problemas em OCaml mostre que sabe usar o método indutivo. Nos problemas de C, não use recursão.
Pode definir funções/métodos auxiliares sempre que precisar (muitas vezes é mesmo preciso).
Normalmente, respostas imperfeitas merecem alguma pontuação.
Fraude implica reprovação na cadeira.

1. [2 valores] Escolha múltipla (as respostas erradas não descontam). Indique as respostas mais correctas aqui:

A	B	C	D
a	c	a	a

A) Qual dos seguintes casos serve para exemplificar uma ligação semi-dinâmica, portanto efetuada parcialmente antes do programa começar a correr e parcialmente depois do programa começar a correr?

a) Variável local em C.
b) Variável global em C.
c) Função em C.
d) Tipo em C (definido usando typedef).

B)Numa linguagem com aninhamento de funções. Como o OCaml ou o GCC, para implementar a passagem duma função por parâmetro…

a) Passa-se um apontador para o código da função.
b) Passa-se um apontador para o código da função e para os valores dos seus argumentos.
c) Passa-se um apontador para o código da função e para o static link pré-calculado da mesma função.
d) Passa-se um apontador para o código da função e para o dinamic link pré-calculado da mesma função.

C) Relativamente a um programa em C errado, que corrompa a memória:

a) Pode acontecer que funcione bem numa plataforma computacional e mal noutras plataformas.
b) Se funciona bem numa plataforma computacional, então funcionará bem em todas as outras plataformas.
b) Se funciona mal numa plataforma computacional, então funcionará mal em todas as outras plataformas.
d) Se corrompe a memória, o programa nem sequer compila e portanto não faz sentido falar na sua execução.

D) Qual dos seguintes conjuntos é constituído exclusivamente por linguagens com sistema de tipos essencialmente estático?

a) Java, OCaml, C.
b) Java, OCaml, JavaScript.
c) Java, C, JavaScript.
d) OCaml, C, JavaScript.

2. [2 valores] Diga qual o tipo da seguinte função em OCaml:

```
let rec f a b = if f a a then f "ola" b elxe f b a ;;
```

Solução:

```
string -> string -> bool
```

3. [2 valores] Considere o seguinte programa escrito em GCC, uma variante do C que suporta aninhamento de funções:

```
void A() {
    int a = 10;
    void B() {
       int b = --a;
       void C() {
          int c = --a + b--;
          A();
       }
       C();
    }
    B();
}

int main(void) {
    A();
    return 0;
}
```

Mostre qual o estado da pilha de execução no momento em que a execução do programa empilha e inicializa o segundo registo de ativação da função B.

Use as convenções habituais das aulas: Para efeito da criação do registo de activação inicial, imagine que cada programa em GCC está embebido numa função sem argumentos chamada `start`. Depois trate todas as entidades globais do programa como sendo locais à função imaginária `start`. Assuma também que a primeira célula da pilha de execução é identificada como posição 00, a segunda célula como posição 01, etc.

Solução:

```
    b  9        --B--
    RET ?
    SL 18
22: DL 18
    a  10->9    --A--
    RET ?
    SL 0
18: DL 14
    c  17       --C--
    RET ?
    SL 10
14: DL 10
    b  9->8     --B--
    RET ?
    SL 6
10: DL 6
    a  10->9->8 --A--
    RET ?
    SL 0
 6: DL 3
    RET ?       --main--
    SL 0
 3: DL 0
    RET ?       --start--
    SL ?
 0: DL ?
```

4. Uma árvore genealógica é uma árvore binária de strings. O seu tipo define-se em OCaml da seguinte forma:

```
type gtree = Nil | Node of string * gtree * gtree ;;
```

Assume-te que não ocorrem strings vazias nestas árvores. Dado um nó duma árvore genealógica, convenciona-se que a sua subárvore esquerda contém os ascendentes do lado materno e a sua subárvore direita contém os ascendentes do lado paterno. Examine o seguinte exemplo de árvore genealógica:


```
let family = Node("Manuel",
                  Node("Maria",
                     Node("Matilde", Nil, Nil),
                     Nil),
               Node("Artur", Nil, Nil))
```

Trata-se da árvore genealógica do Manuel, cuja mãe se chama Maria e cuja avó do lado materno se chama Matilde. O pai do Manuel chama-se Artur. Nada mais se sabe sobre os ascendentes do Manuel. É normal uma árvore genealógica estar incompleta, refletindo o nível de conhecimento parcial existente.

a) [1 valor] Escreva em OCaml uma função

```
size: gtree -> int
```

para contar o número de nós duma árvore genealógica. Portanto resultado dá-nos uma medida do nível de informação genealógica associado a um indivíduo. Um exemplo: o tamanho da árvore do Manuel é **4**.

```
let rec size t =
```

Solução:

```
let rec size t =
    match t with
       Nil -> 0
    | Node(x,l,r) -> 1 + size l + size r
;;
```

b) [3 valores] Escreva em OCaml uma função

```
childOf: string -> string -> gtree -> string list
```

para descobrir numa `gtree` uma pessoa que tenha a mãe e o pai indicados. Se existir mais do que um indivíduo nessas mesmas condições, basta encontrar um deles. Se não existir nenhum, a resposta é a string vazia. Exemplo:

```
childOf "Maria" "Artur" family = "Manuel"     (* family é o exemplo que aparece atrás *)
```

Para escrever a sua função, pode usar o seguinte ponto de partida, mas não é obrigatório:

```
let rec childOf
```

Solução:

```
let rec childrenOf m f t =
    match t with
       Nil -> []
     | Node(x,Nil,Nil) -> []
     | Node(x,l,Nil) -> childrenOf m f l
     | Node(x,Nil,r) -> childrenOf m f r
     | Node(x, Node(a1,l1,r1), Node(a2,l2,r2)) ->
        (if a1 = m && a2 = f then [x] else [])
        @ childrenOf m f (Node(a1,l1,r1))
        @ childrenOf m f (Node(a2,l2,r2))
;;
```

c) [2 valores] Escreva em OCaml uma função

```
fathers: gtree -> string list
```

para descobrir numa gtree todos os pais que lá ocorrerem. Permite-se que apareçam repetições no resultado. Um exemplo:

```
fathers: Node("a", Node("m", Node("x", Nil, Nil), Node("y", Nil, Nil)),
                   Node("f", Node("x", Nil, Nil), Node("y", Nil, Nil))) = ["f"; "y";  "y"]
```

```
let rec fathers t =
```

Solução:

```
let rec fathers t =
    match t with
       Nil -> []
    | Node(x,l,Nil) -> fathers l
    | Node(x,l,Node(b,l1,r1)) -> b :: (fathers l @ fathers (Node(b,l1,r1)))
;;
```

Nome: Num:

5. Considere, em ANSI-C, o seguinte tipo que permite definir listas ligadas de inteiros. Cada nó da lista contém um valor inteiro e um apontador para o nó que se segue. O apontador `NULL` marca o final da lista.

```
typedef struct Node {
    int value ;
    struct Node *next ;
} Node, *List;
```

a) [1 valores] Escreva em C uma função para contar o número de valores positivos (estritamente maiores do que zero) que ocorrem numa lista l. Pretende-se uma solução iterativa, portanto, sem uso de recursão.

```
int countPositive(List l) {
```

Solução:

```
int countPositive(List l){
    int n = 0;
    for( ; l != NULL ; l = l->next )
        if( l->value > 0 )
            n++;
    return n;
}
```

b) [1.5 valores] Escreva em C uma função que permita obter os apontadores para todos os nós duma lista **l** que contenham elementos positivos. O resultado sai num array **a** de apontadores para nós que é passado para a função na altura da chamada. Esse array tem o tamanho indicado pelo argumento **n**. Se existir um excesso de valores positivos, só são guardados os **n** primeiros que forem encontrados. Se existirem valores a menos, as posições em excesso no array devem ser preenchidas com `NULL`. Pretende-se uma solução iterativa, portanto, sem uso de recursão.

```
void readPositive(List l, List a[], int n) {
```

Solução:

```
void readPositive(List l, List a[], int n) {
    int i = 0;
    for( ; i < n && l != NULL ; l = l->next ) {
        if( l->value > 0 ) {
            a[i] = l;
            i++;
        }
    for( ; i < n ; i++ )
        a[i] = NULL;
}
```

c) [1.5 valores] Escreva em C uma função semelhante à anterior, com a diferença que elimina da própria lista os nós cujos apontadores ficam guardados no array **a** de apontadores para nós que é passado para a função na altura da chamada. A função retorna a lista com os elementos que ficam. Repare que podem existir remoções no início, no meio e no final da lista. Pretende-se uma solução iterativa, portanto, sem uso de recursão. [Resolva nas costas desta folha.]

```
List extractPositive(List l, List a[], int n) {
```

Solução:

```
List extractPositive(List l, List a[], int n) {
    int k = 0;
    Node dummy ;  // técnica do apontador atrasado
    dummy.next = l ;
    l = &dummy ;
    while( l->next != NULL && k < n )
        if( l->next->value > 0 )  {
            a[k++] = l->next;
            l->next = l->next->next;
        }
        else
            l = l->next;
    for( ; k < n ; k++ )
        a[k] = NULL;
    return dummy.next;
}
```

6. [4 valores] **Vida**. O objetivo deste problema é definir em JavaScript um sistema de classes adequado à representação de espécies de seres vivos e de relações entre espécies. Esta pergunta é um pouco extensa, mas descobrirá que o problema se resolve escrevendo pouco código.

Vamos considerar **espécies** e **relações simbióticas**.

Espécies: Uma **espécie** é um tipo de ser vivo, que pode ser tanto animal como vegetal. Exemplos: **humano**, **lírio**, **cão**, **raposa**, **peixepalhaço**, **anémona**. Cada espécie é implementada usando uma classe com atributos que podem ser bastante variados. Contudo, existem dois atributos que são comuns a todas as espécies: ***name*** (o nome da espécie, uma string) e ***time*** (o tempo médio de vida dos membros dessa espécie em segundos, um inteiro).

Relações simbióticas: Existem diversos tipos de relações simbióticas: **mutualista** quando os dois participantes recolhem benefícios; **comensalistica** quando o primeiro participante beneficia e o segundo nem beneficia nem é prejudicado (geralmente o primeiro alimenta-se dos restos deixado pelo segundo); **parasítica** quando o primeiro beneficia e o segundo é prejudicado. Há outros tipos de relações simbióticas conhecidas e mais poderão vir a ser descobertos no futuro.

Cada **relação simbiótica** envolve duas componentes, que tanto podem ser espécies como podem ser outras relações simbióticas. Por exemplo: a relação **[peixepalhaço-anémona]** envolve só duas espécies; a relação **[humano-cão]** também só envolve duas espécies; mas a relação **[[humano-cão]-raposa]** já envolve uma relação simbiótica (caçador + cão) e uma espécie (raposa). Cada tipo de relação simbiótica é implementado usando uma classe com atributos que podem ser bastante variados. Mas existem dois atributos que são comuns a todas as relações simbióticas: ***l*** (o primeiro participante na relação) e ***r*** (o segundo participante na relação).

Problema

O objetivo deste problema é a definição dum sistema de classes bem fatorizado e extensível, adequado à representação de aspetos da **vida**. Escreva código compacto, bem fatorizado e extensível. No futuro vamos querer acrescentar novas espécies e novas variedades de relações simbióticas, e o programa tem de estar preparado para isso.

Programe apenas as classes abstratas do sistema. Se escrever classes concretas para exemplificar, deixe-as ficar vazias. As funções que todas as classes devem suportar são as seguintes quatro:

```
constructor(…) – Construtor com argumentos que dependem de cada classe.

getName() – No caso duma espécie, é o nome específico destas, por exemplo "human".
Caso duma relação simbiótica, é a concatenação dos nomes dos elementos envolvidos,
separados por "-" e rodeados por parêntesis retos, por exemplo "[[human-dog]-fox]".

getTime() – No caso duma espécie, é o tempo específico dela. No caso duma relação
simbiótica, é o mínimo de todos os tempos envolvidos.

winWin() - No caso duma espécie, o resultado é true. No caso duma relação simbiótica, o
resultado só é true só se todas as relações envolvidas forem mutualistas. Defina este
método nos sítios certos para poupar na escrita e favorecer a reutilização de código.
Ou seja, fatorize bem.
```

Recomendamos que use as seguintes classes, já identificadas. Se preferir, pode ignorar o que se oferece e fazer diferente.
class Life { // abstrato

Solução:
```
Aula teórica 25
```