Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Letivo 2017/2018
Linguagens e Ambientes Programação – Teste 2 – Parte ❶

09 de junho de 2018 às 09:00

Teste com consulta com 1 hora e 40 minutos de duração

Nome: Num:

Notas: *Este enunciado é constituído por 4 grupos de perguntas. Responda no próprio enunciado, usando a frente e o verso, mas sempre no mesmo caderno em que aparece a pergunta. Normalmente, respostas imperfeitas merecem alguma pontuação. Fraude implica reprovação na cadeira.*

1. **Listas compactadas em ANSI-C**. Vamos usar o tipo `List` definido abaixo para representar listas ligadas de inteiros sem qualquer ordenação particular - os valores são guardados pela ordem em que forem inseridos. Cada nó duma lista é constituído um valor inteiro, um contador e um apontador para o nó seguinte. O apontador NULL marca o final da lista e também serve para representar listas vazias. A função **`newNode`** cria nós inicializados. O contador de cada nó será sempre ≥ 1 e este facto não precisa de ser validado.

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

```
List newNode(int value, int count, List next) {
    List n = malloc(sizeof(Node));
    if( n == NULL ) return NULL;
    n->value = value;
    n->count = count;
    n->next = next;
    return n; }
```

As nossas listas estão otimizadas para poupar nós sempre que aparecerem valores consecutivos iguais. Nós consecutivos com o mesmo valor são colapsados num único nó e usa-se o contador para indicar o número de ocorrências consecutivas desse valor. Por exemplo, a lista [1;1;1;1;4;4;4] será representada usando apenas dois nós em vez de sete nós.

Programe soluções iterativas, portanto sem usar recursão. A complexidade das soluções deve ser linear. Sempre que possível, percorra o argumento principal apenas uma vez. As soluções mais simples serão mais valorizadas.

a) [2 valores] Escreva uma função para testar se uma lista está bem compactada. Para uma lista estar bem compactada, dois nós consecutivos precisam sempre de ter valores diferentes. A lista vazia considera-se bem compactada.

```
bool isCompacted(List l) {
```

b) [2 valores] Escreva uma função para compactar uma lista que poderá estar incompletamente compactada ou não estar compactada de todo. O objetivo é fundir num único nó toda e qualquer sequência de nós consecutivos com valor repetido. A ordem dos valores tem de ser respeitada.

```
List pack(List l) {
```

c) [2 valores] Escreva uma função para descompatar completamente uma lista sem perder informação. O objetivo é que, na lista resultante, o contador de todos os nós seja 1. A ordem dos valores tem de ser respeitada. **[Resolva nas costas.]**

```
List unpack(List l) {
```

2. **Minilinguagem imperativa em JavaScript.** Vamos escrever em JavaScript um interpretador para uma linguagem imperativa muito pequena e limitada. Contudo, o interpretador deverá ser desenhado para facilitar a adição futura de novas construções linguísticas. Estrategicamente, vamos introduzir determinadas classes abstratas a pensar no futuro e vamos preocupar-nos que a implementação esteja bem fatorizada. Este problema tem algumas semelhanças com o último exemplo da teórica 20.

A minilinguagem tem a seguinte curta lista de ingredientes:

- Variáveis globais que são criadas automaticamente nas atribuições.
- Expressões que consistem apenas em constantes inteiras, valores de variáveis e somas (adições).
- As instruções básicas são: atribuição e print.
- Existem uma instrução if para tomar decisões e uma instrução bloco {} que guarda uma sequência de instruções.

Não há declarações, não há ciclos, não há funções, não há booleanos, etc. Eis um exemplo dum programa completo, constituído por um bloco com quatro instruções:

```
{ a = 1;
  b = a+1;
  if( b ) {c = 10; d=a;} else c = 20;
  print(c+a);
}
```

Para o interpretador funcionar, precisamos primeiro de representar o programa usando uma árvore de objetos, que designaremos de **forma interna do programa**. Eis um pedaço de código JavaScript que constrói a forma interna do programa anterior:

```
var program =
  new Program(
    new Block([
        new Assignment(new Var("a"), new Constant(1)),
        new Assignment(new Var("b"), new Add(new VarValue("a"), new Constant(1))),
        new If(new VarValue("b"),
                  new Block([new Assignment(new Var("c"), new Constant(10)),
                                 new Assignment(new Var("d"), new VarValue("a"))]),
                  new Assignment(new Var("c"), new Constant(20))
          ),
        new Print(new Add(new VarValue("c"), new VarValue("a")))
    ])
  );
```

Esta árvore é constituída por 26 nós (a palavra new aparece 26 vezes).

O seu objetivo será implementar as classes da forma interna da minilinguagem. Pedimos-lhe para escrever essa implementação ao longo de várias alíneas. As classes já estão todas identificadas e você terá de programar o interior de cada uma delas - concretamente o construtor (que inicializa os atributos) mais os métodos.

a) **Variáveis**. Para representar as variáveis globais durante a execução dum programa da minilinguagem, precisamos dum dicionário que associe um valor inteiro ao nome de cada variável. Lembramos que em JavaScript um objeto já é um dicionário. Vamos definir o dicionário das variáveis usando a seguinte variável global do JavaScript:

```
var vars = {}
```

No código da implementação, para saber por exemplo qual o valor corrente da variável "a" escreve-se `vars["a"]` e para atribuir o valor 5 à variável "a" escreve-se `vars["a"]=5`. Isto foi explicado no início da teórica 20.

Para não complicar, só vamos considerar programas bem comportados que não usam variáveis indefinidas.

O dicionário das variáveis já ficou definido e você não precisa de fazer nada nesta alínea. Neste momento, já temos uma linha do programa escrita.

b) [2 valores] **Expressões**. Na minilinguagem, as expressões podem aparecer em três locais: do lado direito das atribuições, como argumentos de instruções print e nas condições dos ifs. Na condição dum if, uma expressão inteira com valor diferente de zero é tratada como **true** e com valor zero é tratada como **false**.

As classes que implementam as expressões da minilinguagem devem suportar os seguintes métodos:

`size()` - conta o número de nós da expressão. O resultado é inteiro.
`value()` - avalia a expressão. O resultado é inteiro.
`isTrue()` - avalia a expressão e determina se o resultado é ≠ zero. Resultado booleano.
`isFalse()` - avalia a expressão e determina se o resultado é zero. Resultado booleano.

Por exemplo, a expressão "1+1" tem tamanho 3, tem valor 2, e este valor 2 equivale ao booleano **true**.

A classe Exp é abstrata e as restantes são concretas. Complete os construtores e escreva os métodos pedidos. Fatorize o código dentro do que for possível.

```
class Exp { // abstract
    constructor() { /* vazio */ }









}
class Constant extends Exp {
    constructor(c) {









}
class Add extends Exp {
    constructor(l, r) {









}
class VarValue extends Exp {
    constructor(name) {









}
```

c) [1 valor] **L-expressões**. Uma l-expressão é uma expressão especial que pode aparecer do lado esquerdo do sinal de atribuição. Na minilinguagem, a única forma prevista de l-expressão é o nome duma variável. Mas, no futuro, vamos querer adicionar outras formas de l-expressões, por exemplo componentes de arrays, campos de registos, etc.

As classes que implementam as l-expressões da minilinguagem devem suportar os seguintes métodos:

size() - conta o número de nós da l-expressão. O resultado é inteiro.
assign(v) - atribui um valor à l-expressão. Não tem resultado.

Por exemplo, relativamente à instrução "a = 1+1", a l-expressão "a" tem tamanho 1 e o método assign pode ser usado para meter o valor 2 na variável "a".

A classe LExp é abstrata e a outra é concreta. Complete os construtores e escreva os métodos pedidos. Fatorize o código dentro do que for possível.

```
class LExp { // abstract
    constructor() { /* vazio */ }









}

class Var extends LExp  {
    constructor(name) {









}
```

d) [3 valores] **Instruções**. Na minilinguagem há 5 formas de instruções: atribuição, print, if, bloco e programa. Já todas foram exemplificadas exceto a última. Um programa guarda apenas uma instrução.

As classes que implementam as instruções da minilinguagem devem suportar os seguintes métodos:

size() - conta o número de nós da instrução, incluindo também o número de nós das expressões e das l-expressões que ocorrem internamente. O resultado é inteiro.
run() - executa a instrução. Por exemplo, no caso dum if, avalia primeiro a condição e, consoante o resultado, aplica depois o método run ao ramo then ou ao ramo else. Não se dão mais explicações e você precisa de ter alguma noção de como funcionam as instruções numa linguagem imperativa. Não tem resultado.
extension() - calcula o número de máximo de instruções básicas (print e atribuição) da instrução que poderão ser ativadas durante a execução. Esse valor é calculado analisando a instrução, mas sem a executar. Estamos sempre interessados no pior caso. Por exemplo num if, em que há dois ramos à escolha, escolhemos o ramo que produz o resultado maior, independentemente do valor da condição. O resultado é inteiro.
big() - produz **true** se o resultado de size() for maior do que 100 e **false** no caso contrário.

Por exemplo, o programa apresentado no início desta grupo tem tamanho 26, a sua execução produz algum output gerado pelos prints, o valor de extensão é 5, e o programa não é considerado "big".

A classe Instruction é abstrata e as restantes são concretas. Complete os construtores e escreva os métodos pedidos. Fatorize o código dentro do que for possível. A classe `Program` já foi toda escrita.

```
class Instruction { // abstract
    constructor() { /* vazio */ }









}
```

```
class Block extends Instruction  {
    constructor(seq) {  // o argumento é um vetor de instruções









}
class Assignment extends Instruction  {
    constructor(l, r) {









}
class If extends Instruction  {
    constructor(c, l, r) {









}
class Print extends Instruction  {  // para imprimir, usar "console.log".
    constructor(exp) {









}
class Program extends Instruction  {  // já está completamente feito
    constructor(inst) { super(); this.inst = inst; }
    size() { return 1 + this.inst.size(); }
    run() { vars = {}; this.inst.run(); }
    extension() { return this.inst.extension(); }
}
```

Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Letivo 2017/2018
Linguagens e Ambientes Programação – Teste 2 – Parte ❷

09 de junho de 2018 às 09:00

Teste com consulta com 1 hora e 40 minutos de duração

Nome: Num:

3. [4 valores] Considere o seguinte programa escrito em GCC, uma variante do C que suporta aninhamento de funções:

```
#include <stdio.h>
int SUM(int a[], int n) {
    int NEXT(int a[], int n) {
      a[n] = n * 10;
      return SUM(a+1, n-1);
    }
    if( n == 0 )
      return 0;
    else
      return a[0] + NEXT(a, n);
}
int main(void) {
    int a[4] = {7, 7, 7, 7};
    int b = SUM(a, 4);
    return 0;
}
```

Mostre qual o estado da pilha de execução **no momento em que acabou de ser empilhado o 6° registo de ativação**. Não se esqueça dos registos de ativação das funções `start` e `main`. Este programa poderá escrever fora dos limites do array - isso é deliberado; não é um erro no enunciado.

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.

11	23	35
10	22	34
09	21	33
08	20	32
07	19	31
06	18	30
05	17	29
04	16	28
03	15	27
02	14	26
01	13	25
00	12	24

4. [4 valores] Escolha múltipla. As respostas erradas não descontam. Indique as respostas aqui:

A	B	C

A) Regra de escopo dinâmico comparada com a regra de escopo estático:

a) A regra de escopo dinâmico é sempre usada nas linguagens dinamicamente tipificadas, como o JavaScript.

b) A regra de escopo estático implica uma implementação mais simples de funções aninhadas do que a outra regra.

c) A regra de escopo dinâmico ajuda uma linguagem a ter tipificacão estática.

d) A regra de escopo estático ajuda uma linguagem a ter tipificacão estática. Já a regra de escopo dinâmico prejudica a tipificação estática.

B) O static link do registo de ativação duma função FUN aponta para

a) O registo de ativação corrente.

b) O registo de ativação mais recente da função dentro da qual FUN está definida.

c) A zona das variáveis globais do programa.

d) O registo de ativação da função que chamou FUN.

C) Um ficheiro XML:

a) É um ficheiro binário que guarda a serialização duma árvore n-ária.

b) É um ficheiro de texto contendo marcas que ajudam a definir uma estrutura de árvore n-ária nos dados.

c) É um ficheiro de texto que usa as convenções sintáticas dos objetos JavaScript para definir uma estrutura de árvore n-ária nos dados.

d) É um ficheiro de texto onde podemos colocar tudo o que desejarmos sem preocupação com a forma sintática do conteúdo.