Lista de Exercícios de LP

Prof. Carlos Bazilio

carlosbazilio@id.uff.br

Tópicos

1 Esquemas de Tradução

1.1 Considere o seguinte programa em C:

// Arquivo programa.c
#include “bib.h”
void main() { … }
 	
// Arquivo bib.h
#include “bib2.h”
#include “bib3.h”
int f (int *) { … };
	// Arquivo bib2.h
	int g () { … };
	
	// Arquivo bib3.h
	void h (void) { … };

a) Como fica o programa após a fase de pré-processamento?

b) Imagine que no arquivo bib.h o programador cometeu um erro de digitação e digitou 2 vezes o include ao arquivo bib2.h. Que erro acontecerá? Esse erro acontecerá em que momento: pré-processamento, compilação, linkedição ou execução?

2 Expressões

2.1 Considere o seguinte programa em C:

```
int f (int *i) {
        *i = *i + 5;
        return 4;
}
void main() {
        int x = 3;
        x = x + f(&x);
        printf("%d\n", x);
}
```

Qual o valor impresso por x quando?

a) Operandos são avaliados da esquerda para a direita.

b) Operandos são avaliados da direita para a esquerda.

2.2 Um aluno ouviu falar da linguagem Ruby e ficou curioso em saber como esta funciona. Leu que os operadores “or”, “||” e “|” realizam um OU entre 2 expressões booleanas. Querendo saber se havia diferença, ele abriu o prompt da linguagem e executou a seguinte sequência de comandos:

```
MaquinaTeste-Prompt:~ alunocomp$ irb
> true or condicao
=> true
> true || condicao
=> true
> true | condicao
NameError: undefined local variable or method `condicao' for main:Object
      from (irb):2
```

(Obs.: irb é a chamada do interpretador Ruby, “>” indica as entradas do aluno, enquanto que “=>” indica a resposta do interpretador).

Com estes testes, ele entendeu o operador “|”, mas não conseguiu distinguir os operadores “or” e “||”. Daí, realizou outros testes:

```
> a = false or true
=> true
> a
=> false
> b = false || true
=> true
> b
=> true
```

O que este aluno deduziu com respeito aos operadores?

2.3 Considere o código abaixo:

```
typedef struct ponto { int x; int y; int z } ponto;
main() {
        int i;
        ponto figura[4];
        for (i=0; v[i]!=NULL; i++) {
                v[i](figura);
        }
}
```

Apresente, em C, uma declaração para v. Não é necessário repetir o código da main().

2.4 Suponha a expressão abaixo em C. Descreva que tipo pode ter cada um dos identificadores que aparecem nesta expressão.

```
g(x++, h(g(10)), f()->v).k[y(v)] = {50, “abc”};
```

3 Macros

3.1 Macros são um recurso de programação que permite o reuso de expressões e comandos que ocorrem com frequência. Por exemplo, suponha uma macro em C para troca de valores entre variáveis:

```
#define troca(A, B) {int tmp; tmp = A; A = B; B = tmp;}
```

O compilador resolve esta definição fazendo uma expansão de código, da mesma forma que ocorre com os includes. Compare esta abordagem com a implementação de uma função que realize esta troca. Qual a vantagem de se usar uma macro? Qual seria uma possível limitação em relação a definição como função?

4 Escopo

4.1 Considere o código abaixo que implementa a função fatorial em Javascript.

```
1: y = 5;
2:
3: function fat(x) {
4:         base = 1;
5:         function fat_aux(x1, acum) {
6:                 if (x1 === 0)
7:                         return acum;
8:                 else
9:                         return fat_aux(x1 - 1, acum*x1);
10:        }
11:        return fat_aux(x, base);
12: }
13:
14: console.log(fat(y));
```

a) Considerando que Javascript implementa escopo estático, em que linhas do código acima é visível cada uma das variáveis (incluindo os nomes de funções)? Por exemplo, uma hipotética variável *xpto* é visível entre as linhas *i0* e *i5.*
b) Há diferença em relação a tratar o escopo de forma dinâmica neste caso? Justifique.

4.2 Considere o seguinte programa em C++ (C + OO):

```
#include <stdio.h>
  int x=0;
  void p(int,int);
  void main(){
    int x = 1;
    p(x,x);
  }
  void p(int y, int z){
    x = x+1;
    y = y+1;
    z = z+1;
    printf("%d\n",x+y+z);
  }
```

a) Qual é o valor impresso? Desenhe o estado da pilha de ativação no momento em que a impressão ocorre, indicando as variáveis visíveis em cada registro de ativação.
b) Observando a pilha informada no item a), caso a manipulação da variável x em p acessasse a declaração de x na main(), diríamos que o escopo de C++ é dinâmico. Informe o valor impresso nesse novo tipo de escopo.

c) Suponha que nós modifiquemos a declaração de p para *p(int &y, int &z){...}*, ou seja, fazendo com que os seus parâmetros sejam passados por referência. Qual é o valor impresso neste caso? (Considere escopo estático, como no item a) – tipo de escopo que estamos acostumados, o qual depende da estrutura do código)

4.3 Considere o seguinte fragmento de código em C:

```
{
    int a, b, c;
    …
    {
        int d, e;
        …
        {
            int f;
            …
        }
    }
    …
    {
        int g, h, i;
        …
    }
}
```

Supondo que cada variável inteira ocupa 4 bytes, quanto de espaço total é requerido pelas variáveis neste código?

5 BNF

5.1 Considere o trecho abaixo extraído da documentação da linguagem LUA:

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar syntax:

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else block] end
```

a) Especifique sentenças nesta metalinguagem que descrevam funções em C, ou seja, especifique como é o formato geral de uma função em C. Na parte de comandos, considere apenas os de atribuição e o comando *return.*
b) Suponha que C permita que uma função retorne mais que um valor. Altere a gramática de forma a sugerir uma nova sintaxe para essa característica.
c) Implemente uma função nesta nova sintaxe para o cálculo de raízes de uma expressão do 2o. grau: (-b +/- sqrt (b*b – 4*a*c)) / 2*a, onde sqrt é a função raiz quadrada.

5.2 Considere o programa Python abaixo (esquerda) que imprime se um número é primo e está intencionalmente com alguns erros. Sua suposta saída está à direita. A função range() gera um intervalo de valores, baseado nos parâmetros passados. Por exemplo, a chamada range (2,10) gera o intervalo [2, 3, 4, 5, 6, 7, 8, 9]. Os outros comandos (break, print, if e for) tem comportamento similar aos equivalentes em C.

```
for n in range(2, 10):
   for x range(2, n):
      if n % x == 0
         print(n, 'e divisivel por', x, ' => ', n/x)
         break
   else:
   print(n, 'e primo!')
```

```
(2, 'e primo!')
(3, 'e primo!')
(4, 'e divisivel por', 2, ' => ', 2)
(5, 'e primo!')
(6, 'e divisivel por', 2, ' => ', 3)
(7, 'e primo!')
(8, 'e divisivel por', 2, ' => ', 4)
(9, 'e divisivel por', 3, ' => ', 3)
```

Para compreender precisamente a sintaxe do código listado, abaixo segue um extrato da gramática de Python (https://docs.python.org/3/reference/grammar.html). Observe que, diferente de muitas outras linguagens, indentações (INDENT e DEDENT) e quebras de linha (NEWLINE) fazem parte da sintaxe da linguagem. Ou seja, uma indentação esquecida, por exemplo, causa erro no código.

```
programa: (NEWLINE | stmt)*
stmt: if_stmt | while_stmt | for_stmt | simple_stmt | funcdef
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
simple_stmt: print_stmt | break_stmt
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [argslist] ')'
```

```
argslist: NAME (',' NAME)*
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
```

a) Aponte os erros sintáticos que existem no código fornecido.
b) O comando else acima está vinculado ao if ou ao for? Analise o código considerando o algoritmo para identificação de números primos.
c) À partir da regra funcdef, declare uma função primos, a qual recebe um valor e executa o código fornecido para um número qualquer, e não apenas até o valor 10.

6 Passagem de Parâmetros

6.1 Suponha uma linguagem que não possua o comando *while*. Um programador inquieto utiliza os recursos da linguagem e elabora o código abaixo para implementar o *while*, o qual aparenta estar correto:

```
void fwhile (bool condição, commands corpo) {
        if (condição) {
            corpo;
            fwhile (condição, corpo);
        }
}
```

Após implementá-lo, ele testa com o código abaixo, que sintaticamente está correto nesta linguagem, mas executa indefinidamente ("entra em *looping*").

```
int x = 0;
fwhile (x < 10, {x = x + 1});
```

O que pode estar acontecendo? Como podemos contornar este problema?

6.2 Considere o código abaixo numa linguagem que suporta funções com diferentes modos de passagem de parâmetros. Que valores são impressos na função foo e na main quando:

```
function foo(int a, int b, int c)
  begin
    a := b + c;
    b := c + 1;
    print a, b, c;
  end

function main
  begin
    int i := 5;
    int j := 10;
    ink k := 15;
    foo(i, j, j + k);
    print i, j, k;
  end
```

a) Todos os parâmetros são passados por valor.
b) **a** e **b** por referência e **c** por valor.
c) **a** e **b** por valor-resultado e **c** por valor.
d) Todos os parâmetros são passados por nome.

6.3 Considere o seguinte programa em C:

```
int f(int v[]) {
        int v2 [] = {10, 20, 30};
```

```
        v = v2;
}

void g(int v[]) {
        v[0] = 10;
}

int main()
{
        int vetor [] = {1, 2, 3};
        f(vetor);
        printf("Depois de f: v[0] = %d\n", vetor[0]);
        g(vetor);
        printf("Depois de g: v[0] = %d\n", vetor[0]);
        return 0;
}
```

Sua execução tem a seguinte saída:

```
Depois de f: v[0] = 1
Depois de g: v[0] = 10
```

Aparentemente a atribuição em f() não fez efeito. O que ocorreu? Que tipo de passagem de parâmetros está sendo realizada? Justifique sua resposta.

6.4 C# possui passagens de parâmetro por valor, por referência (ref) e por resultado (out). Nesta última, o valor inicial do parâmetro passado pela função chamadora é descartado na função chamada. Além disso, na função chamada, o programador é proibido de tentar usar o valor inicial da variável passada. Considere o código à seguir para cálculo do fatorial em C#:

```
void fat (int x, out int ac) {
        if (x > 0) {
                ac = ac * x;
                fat (x-1, out ac);
        }
}
```

a) Esse código está correto, considerando as restrições listadas acima? Por quê?
b) Passar o parâmetro ac por referência (trocar out por ref) trará o mesmo resultado? Por quê?
c) Do ponto de vista de uso da função, compare o retorno de um valor por uma função (comando return) da passagem de parâmetro por resultado (out em C#).

6.5 Considere a declaração de uma rotina com 2 parâmetros (x e y) do tipo inteiro passados pelo modo valor-resultado:

procedure p(valor-resultado x, y: integer) <corpo>

Se nós programamos numa linguagem que não implementa chamada por valor-resultado, mas apenas chamada por referência, como podemos definir uma rotina p' que tenha o mesmo efeito que esta acima? Obs.: Comandos como atribuições, condicionais e repetições existem na linguagem para implementar p'.

6.6 Uma aplicação interessante da passagem de parâmetros por nome é o mecanismo de Jensen (*Jensen's Device*). Suponha a rotina de somatório de propósito geral em Algol abaixo:

```
1 real procedure Soma (Expr, Indice, LimiteInf, LimiteSup); value LimiteInf, LimiteSup;
2       real Expr, integer Indice, LimiteInf, LimiteSup;
3       begin
4               real Temp;
5               Temp := 0;
6               for (Indice := LimiteInf step 1 until LimiteSup do
7                       Temp := Temp + Expr;
8               Soma := Temp;
9       end Soma;
```

Neste programa, *Expr* e *Indice* são passados por nome, enquanto que LimiteInf e LimiteSup são por valor. A chamada Soma(A[i], i, 1, 25) realiza a soma dos primeiros 25 elementos de A, enquanto que a chamada Soma (A[i]*B[i], i, 1, 25) realiza a soma dos produtos dos primeiros 25 elementos correspondentes dos vetores A e B.

a) Altere a chamada para calcular a soma dos quadrados dos primeiros 100 valores inteiros?
b) Apresente uma rotina similar para o cálculo do maior elemento num conjunto de valores.

6.7 Considere a seguinte sequência de declarações numa linguagem imperativa qualquer:

```
z : integer;
procedure p (x:integer)
    < corpo >;
```

Escreva uma sequência de 1 ou mais comandos para <corpo> de forma que o código abaixo

```
z := 1; p(z); write(z)
```

terá diferentes saídas (comando write) para quando *z* é passado para *p* por valor, por referência e por valor-resultado.

7 Chamada de Função

7.1 Considere o código abaixo que implementa a busca binária:

```
1. int binary_search(int *data, int toFind, int start, int end)
2. {
3.   // Get the midpoint.
4.   int mid = start + (end - start)/2;   //Integer division
5.   if (start > end) return -1;          //Stop condition.
6.   else if (data[mid] == toFind) return mid;      //Found?
7.   else if (data[mid] > toFind)        //Data is greater than toFind, search lower half
8.       return binary_search(data, toFind, start, mid-1);
9.   else                                //Data is less than toFind, search upper half
10.      return binary_search(data, toFind, mid+1, end);
11. }
12. int * vetor = { 1, 2, 4, 9, 16, 25, 36 };
13. binary_search (vetor, 16, 0, 6);
```

a) Ilustre a pilha de ativação, para a chamada fornecida (linha 13), quando a função alcança o caso base da recursão (linha 6). O registro deve conter os valores dos parâmetros e o endereço de retorno.
b) Considerando que inteiros ocupam 4 bytes e endereços de memória ocupam 8 bytes, qual o tamanho total de memória alocado ao chegar ao caso base?
c) Esta recursão é final? Justifique sua resposta.

7.2 Considere o código abaixo em C. Este simula o comportamento de uma corotina? Justifique sua resposta.

```
void corotina (int modo) {
        // Variáveis locais ...
        if (modo == 0) {
                // Comandos da Corotina 0
                corotina(1);
        }
        else {
                // Comandos da Corotina 1
                corotina(0);
        }
}
```

7.3 Seja a função em C dada abaixo:

```
int gcd(int x, int y) {
        if ((x < 1) || (y < 1))
                return 0;
        else
                if (x == y)
                        return x;
                else
                        if (x < y)
                                return gcd(x, y-x);
                        else
                                return gcd(x-y, y);
}
```

a) Esta possui recursão em cauda? Se sim, por quê? Se não, altere a função para que ela apresente esta característica.
b) Gere uma versão iterativa (sem recursão).
c) Compare o tamanho de memória de dados exigido (tamanho da pilha) entre a versão do enunciado e a iterativa para a chamada gcd(4,6). Suponha que cada inteiro (tipo *int*) ocupa 4 bytes.

7.4 Considere o programa abaixo, implementado em Lua, o qual manipula uma corotina:

```
function fib()
    anterior = 0
    corrente = 1
    while (true) do
            coroutine.yield(corrente) -- Interrompe a execução da corotina,
            -- devolvendo o valor passado por parâmetro
            corrente = anterior + corrente
            anterior = corrente - anterior
    end
end

function principal()
    co = coroutine.create(fib) -- Cria uma corotina com a função fib
    print ("Digite um número para cálculo da série de Fibonacci")
    num = io.read("*n") -- Lê um número
    cont = 1
    while (cont <= num) do
            print (cont .. "o. = ", coroutine.resume(co)) -- .. Concatena strings
            -- coroutine.resume desvia a execução para a corotina
            cont = cont+1
    end
end

principal() -- Início do programa
```

a) Como deve funcionar a pilha de ativação para que este programa execute corretamente?

b) Supondo que usuário digitou o valor 5 na entrada, desenhe o estado da pilha na 5ª chamada de coroutine.resume na função principal.

8 Coleta Automática de Lixo

8.1 Considere o código abaixo:

```
int f (int x) {
        int *a = NULL;
        int *b = (int *) malloc (sizeof (int));
        a = b;
        *b = x
        return *a;
}
```

a) O código acima gera lixo de memória? Se não gera, altere para que gere. Justifique sua resposta.
b) Considerando a existência de um coletor de lixo por contagem de referências, quantas instruções são executadas neste trecho de código para a chamada *"z = f(10)"?* Considere atribuições e comandos de return como 1 instrução, malloc's e if's como 2 instruções. Explique como chegou em tal quantidade.

8.2 Suponha o trecho de código de um programa abaixo, onde new() aloca um área de memória e size_memory() retorna a área de memória total alocada pelo programa:

```
print(size_memory()) // 100
x = new(1000)
print(size_memory()) // 1100
x = null
print(size_memory()) // 1100
```

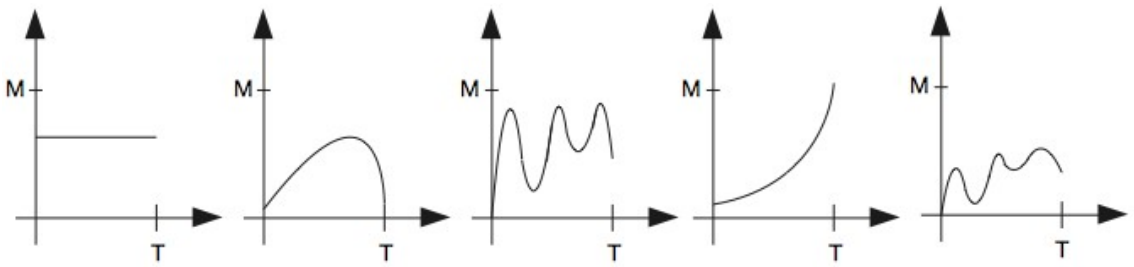
Podemos afirmar que esta linguagem não implementa coleta de lixo (justifique sua resposta)? Que alterações no trecho de código e saída garantiriam a implementação dos 2 métodos de coleta vistos (responda em separado para cada método)?

8.3 "A coleta de lixo utilizando a técnica de *Contagem de Referências* não se dá muito bem com estruturas de dados circulares, como listas circulares." Esta afirmação procede? Justifique sua resposta desenhando alguma figura que descreva a situação. Podemos afirmar o mesmo com o uso da técnica *Mark-and-Sweep*? Novamente, justifique sua resposta.

8.4 Analise os gráficos (tempo x memória) abaixo. Nos gráficos, M indica o tamanho de alocação máximo do heap, enquanto que T é o tempo que o programa leva executando. Na sua resposta, fale de 1o, 2o, …, 5o gráfico para se referir a cada imagem.

a) Indique qual melhor caracteriza a execução de um programa com coleta de lixo por contagem de referência e qual melhor caracteriza com o mark-and-sweep.

b) Por que os outros 3 gráficos foram descartados na análise? Explique sucintamente.


M
T
M
T
M
T
M
T
M
T

9 Tipos

9.1 Considere o código que manipula vetores de números e estruturas:

```
typedef struct produto {
	int codigo;
	char nome[20];
	double preco;
} tproduto;

int soma(int vet[4]) {
	int i, soma = 0;
	for (i=0; i<4; i++)
		soma = soma + vet[i];
	return soma;
}

double somaPrecos(tproduto vet[2]) {
	int i;
	double soma = 0;
	for (i=0; i<2; i++)
		soma = soma + vet[i].preco;
	return soma;
}

main() {
	int valores[] = {1, 2, 3, 4};
	printf("%d\n", soma(valores));
	tproduto produtos[] = {{1, "arroz", 5.5}, {2, "feijao", 8}};
	printf("%3.2f\n", somaPrecos(produtos));
}
```

As funções *soma* e *somaPrecos* são similares, pois recebem um produto e calculam a soma de seus conteúdos de forma adequada. Considere que queiramos criar uma versão genérica deste código, ou seja, uma única função para realizar a soma. Implemente as funções *alocaValor*, *retornaValor*, *alocaProduto*, *retornaPreco* e a própria *somaValores,* genérica, necessárias para que o código abaixo funcione.

```
main() {
	void *valores2[] = {alocaValor(1), alocaValor(2), alocaValor(3), alocaValor(4)};
	printf("%3.2f\n", somaValores(valores2, 4, retornaValor));
	void *produtos2[] = {alocaProduto(1, "arroz", 5.5), alocaProduto(2, "feijao", 8)};
	printf("%3.2f\n", somaValores(produtos2, 2, retornaPreco));
}
```

9.2 Considere o código abaixo que contém a definição de uma lista genérica para a manipulação de figuras 2D num plano. Inicialmente foi definido apenas o tipo específico para manipular círculos e seu método de desenho, o qual apenas exibe as características destes objetos. Sabendo disso, faça:

a) Defina um tipo que represente uma reta (2 pontos), assim como seu método de desenho, de forma que objetos deste tipo possam ser adicionados à lista e o programa continue funcionando.
b) Adicione uma reta que conecte os pontos (0,0) e (3,4).
c) Crie uma função que translade as figuras em função de uma coordenada (x,y), ou seja, some x e y às coordenadas de cada ponto das figuras.

```
typedef struct lista_gen {
		void* info; // Informação do nó
		void (* desenha)(void *); // Campo que representa uma função
		struct lista_gen* prox;
} ListaGen;
typedef struct circulo { // Definição do tipo círculo
		float raio; float x; float y;
} tCirculo;
ListaGen* insere(ListaGen *l, void* p, void des(void *)) {
		ListaGen* n = (ListaGen *) malloc(sizeof(ListaGen));
		n->info = p; n->desenha = des; n->prox = l;
		return n;
}
void desenhaCirculo (void *p) { // Método de desenho do círculo
		tCirculo *c = (tCirculo *)p;
		printf("Raio: %f na posição (x, y)\n", c->raio, c->x, c->y);
}
void desenha(ListaGen *l) { // Método que desenha os objetos da lista
		ListaGen *laux;
		for (laux = l; laux != NULL; laux=laux->prox) {
			laux->desenha(laux->info);	}
}
main() {
		ListaGen* l = NULL;
		tCirculo *circulo1 = (tCirculo *)malloc(sizeof(tCirculo));
		circulo1->raio = 3.5; circulo1->x = 0; circulo1->y = 0;
		l = insere(l, circulo1, desenhaCirculo);
		desenha(l);
}
```