UFMG

Linguagens de Programação - Lista de Exercícios 4

Arthur Pontes Nader - 2019022294
Belo Horizonte - MG - Brasil

Passagem de parâmetros

1)

(a) O programa produz como resultado o valor 5. A variável x dentro da função receberá o valor de x global, que é 0, somado de uma unidade, o que resulta no valor 1. Já y e z receberão o valor 1 como cópia do parâmetro passado. Assim, como em seguida se incrementa y e z em uma unidade, ambos terão valor 2. Dessa forma, x + y + z = 1 + 2 + 2 = 5.

(b) No caso em que os valores são passados por referência, o valor impresso será 7. Isso ocorre pois dentro da função tanto y como z apontam para a mesma posição de memória. Dessa forma, novamente a variável x dentro da função receberá o valor global de x acrescido de uma unidade, resultando em 1. Já o incremento de y, fará com que o local de memória que inicialmente tinha valor 1 passe para o valor 2. Em seguida, z incrementa o mesmo local de memória que foi utilizado anteriormente, fazendo com que nele o valor armazenado passe de 2 para 3. Portanto, x + y + z = 1 + 3 + 3 = 7.

2)

(a) Após o pré-processamento, tem-se o seguinte programa:

```
int main (int argc , char ** argv){
    printf (" sum = %d\n", (argc + argv [0][0]);
}
```

(b) A captura de variável é um problema que ocorre quando uma variável local presente na expansão de macros possui o mesmo nome de um dos parâmetros reais. Nesse caso, a computação a ser realizada ocorrerá considerando o valor armazenado na variável local, o que pode levar o programa a produzir resultados inesperados.

(c)

```
#include <stdio.h>
#define run(r1, r2) r1 > r2? (r1) : (r2)

int a = 0, b = 0;
int incA() {return ++a;}
int incB() {return ++b;}

int main() {
    printf("a = %d, b = %d\n", a, b);
    run(incA(), incB());
    printf("a = %d, b = %d\n", a, b);
}
```

O programa acima exemplifica a múltipla avaliação de parâmetros, em que a função incB() é executada duas vezes dentro da expansão de macros. Isso pode ser percebido pois os seguintes resultados são obtidos como saída do programa:

```
a = 0, b = 0
a = 1, b = 2
```

3)

(a) Z possuirá valor igual a 30 após a chamada da função.

(b) O programa lê um valor inteiro do usuário e o guarda na variável “a”. Em seguida, realiza o somatório de $1/(i + a)^2$ com i variando de 1 até 100.

4)

(a) Após a chamada do método m1.swap1(m2), tanto o valor de m1.i quanto o de m2.i será 4.

(b) Já após a chamada do método m1.swap2(m2), m1.i e m2.i também possuirão o mesmo valor, só que agora esse valor será 3.

(c) Assim como na letra a, a chamada de m1.swap3(m2.i) fará com que ambos m1.i e m2.i possuam 4 como valor.

(d) Para tipos primitivos, Java adota a passagem de parâmetros por valor.

(e) Já para objetos, Java também adota a passagem por valor da referência para o objeto.

Programação Lógica

1)

(a)

```
firstCousin(X,Y) :- parent(Z,X), parent(W,Y), not(X=Y), not(Z=W), parent(G,Z),
    parent(G,W).
```

(b)

```
descendant(X,Y) :- parent(Y,X).
descendant(X,Y) :- parent(Z,X), descendant(Z,Y).
```

2)

```
third([First|[Second|[Third|_]]], Third)
```

3)

```
dupList([],[]).
dupList([Element|Rest1], [Element|[Element|Rest2]]) :- dupList(Rest1, Rest2).
```

4)

```
isIn(Element,[Element|_]).
isIn(Element,[_|Tail]) :- isIn(Element, Tail).

allAreIn([],_).
allAreIn([Head|Tail], List) :- isIn(Head, List), allAreIn(Tail, List).

isEqual([],[]).
isEqual(List1, List2) :- allAreIn(List1,List2), allAreIn(List2,List1).
```

5)

```
isIn(Element,[Element|_]).
isIn(Element,[_|Tail]) :- isIn(Element, Tail).

isDifference([], _, []).
isDifference([Head|Tail1],List2,Result):- isIn(Head, List2), !,
isDifference(Tail1,List2,Result).
isDifference([Head|Tail], List2, [Head|Result]):- isDifference(Tail,List2,Result).
```

6) Fatos e Regras:

```
append([], B, B)
append([Head|TailA], B, [Head|TailC]) :- append(TailA, B, TailC).

Query: append(X, Y, [1, 2])
~Query: not(append(X, Y, [1, 2]))

not(append(X, Y, [1,2]))    append([Head|TailA], B, [Head|TailC])
---------------------------------------------------------------------------
```

Res {X = [1|TailA], Y = B, [1|[2]] = [Head|TailC]}

not(append(TailA, Y, [2])) append([Head|TailA], B, [Head|TailC])

Res {TailA = [2|TailA], Y = B, [2|[]] = [Head|TailC]}

not(append(TailA, Y, [])) append([Head|TailA], B, [Head|TailC])

Res {TailA = [], Y = B, [] = [Head| TailC]}

not(append([], B, B)) append([], B, B)

False

7) maxList([OneElement], OneElement).
maxList([Head|Tail], Max):- maxList(Tail, LastMax), (Head > LastMax -> Max = Head; Max = LastMax).

8) createList(List, List, 0) :- !.
createList(List, Rest, Max) :- New is Max-1, createList(List, [Max|Rest], New).

exchange(Element,[Element|Rest],Rest).
exchange(Element,[First|Rest],[First|FirstRest]) :-
exchange(Element,Rest,FirstRest).

generatePermutation([],[]).
generatePermutation([Head|Tail],NewPermut) :- generatePermutation(Tail,Aux),
exchange(Head,NewPermut,Aux).

absolutValue(Num1,Num2) :- Num1 < 0 -> Num2 is -Num1; Num2 = Num1.

checkerConfiguration(_,[],_).
checkerConfiguration(Value,[Head|Rest],ActualDist) :- Sub is Value - Head,
absolutValue(Sub, Abs), Abs =\= ActualDist,
NewDistance is ActualDist + 1, checkerConfiguration(Value,Rest, NewDistance).

isValid([]).
isValid([Q|Rest]) :- isValid(Rest), checkerConfiguration(Q,Rest,1).

nqueens(N,Q) :- createList(List,[],N), generatePermutation(List, Q), isValid(Q).