Aluno: Carlos Victor Martins da Silva

Matrícula: 476516

Link para o arquivo: [PasteBin](https://pastebin.com/bnKKgSiB) / [Google Drive](https://drive.google.com/file/d/1kOmC2GA6pDZOQmIy42ONOiPMH1eS1LzG/view?usp=sharing)

| **Questões feitas** | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 11 |
| X | X | X | a. X | X | X | X | a. X | X | X |
| b. X | b. X |

1. Consider the following statement:

proc {P X}

if X>0 then {P X-1} end

end

Is the identifier occurrence of P in the procedure body free or bound? Justify your answer.

\* **Hint:** this is easy to answer if you first translate to kernel syntax.

Resposta:

| Ao declararmos esse procedimento, o compilador do ambiente Mozart irá transformar esse código com açúcares sintéticos (syntax sugars) para o sua versão feita na linguagem núcleo (kernel language):  P = proc {$ X}  if X>0 then {P X-1} end  end  Observando a ordem com que os comandos são chamados e a organização do escopo, é possível notar que o símbolo “P” dentro do procedimento é limitado ao escopo do procedimento. |
| --- |

2. Section 2.4 explains how a procedure call is executed. Consider the following procedure MulByN:

declare MulByN N in

N=3

proc {MulByN X ?Y}

Y=N\*X

end

together with the call {MulByN A B}. Assume that the environment at the call contains {A → 10, B → x1}. When the procedure body is executed, the mapping N → 3 is added to the environment.

Why is this a necessary step? In particular, would not N → 3 already exist somewhere in the environment at the call? Would not this be enough to ensure that the identifier N already maps to 3? Give an example where N does not exist in the environment at the call. Then give a second example where N does exist there, but is bound to a different value than 3.

Resposta:

| É necessário declarar o valor de N no corpo do procedimento para que essa variável esteja presente e seja constante a cada chamada desse procedimento independente do contexto. No exemplo de código abaixo é possível ver que o valor de N se restringe ao escopo local que define a função e executa os comandos Browse que mostra os valores para testes.  declare A = 10  declare B  local MulByn in    local N in  N = 3  proc {MulByn X ?Y}  Y = N \* X  end  end  {MulByn A B}  {Browse B} % Resultado: 30  {Browse N} % Resultado: 3  end  Se uma variável N é declarada fora do bloco de código “local N in … end”, o valor ao executar o procedimento o valor de N presente dentro do escopo local seria respeitado na multiplicação feita mas o valor de N seria diferente ao executar a função “Browse” já que não houve uma definição do mesma variável na pilha de variáveis locais. No exemplo de código abaixo é visível esse comportamento.  declare A = 10  declare B  declare N = 9  local MulByn in    % N presente  local N A B in  N = 3  proc {MulByn X ?Y}  Y = N \* X  end    A = 10  {MulByn A B}  {Browse B} % Resultado: 30  {Browse N} % Resultado: 3  end  % N presente fora da declaração do procedimento  {MulByn A B}  {Browse B} % Resultado: 30  {Browse N} % Resultado: 9  % N presente declarado dentro de um escopo local isolado  % e sobrescrevendo o valor presente na declaração de N  local N in  N = 100  {MulByn A B}  {Browse B} % Resultado: 30  {Browse N} % Resultado: 100  end  end |
| --- |

3. If a function body has an if statement with a missing else case, then an exception is raised if the if condition is false. Explain why this behavior is correct. This situation does not occur for procedures. Explain why not.

Resposta:

| Por conta da sintaxe da linguagem Oz, o fim de uma função deve sempre terminar com uma expressão que será passada como resultado. Um caso exemplo seria quando há a declaração da expressão ‘if … then … else’, o parser irá interpretar essa declaração como uma expressão e não declarar um erro mesmo se o ‘else’ não possuir nenhuma expressão declarada em seu corpo, interpretando-se que o ‘if … then … else’ seja apenas uma expressão. Agora, durante a interpretação do código, caso o ‘if’ falhe (retorne false), o interpretador tentará executar o ‘else’ que não possui comandos e nem limitação de fim fazendo com que seja lançado um erro. |
| --- |

4. This exercise explores the relationship between the **if** statement and the case statement.

(a) Define the **if** statement in terms of the **case** statement. This shows that the conditional does not add any expressiveness over pattern matching. It could have been added as a linguistic abstraction.

Resposta:

| if X then <s>¹ else <s>² end ::= case X  of true then  <s>¹  [] else  <s>²  end  Como uma abstração de linguagem, o mais correto seria colocar um try … catch X … end para garantir que o programa não acabe sofrendo com erros de interpretação vindos da conversão para linguagem núcleo dessas abstrações linguísticas. |
| --- |

(b) Define the **case** statement in terms of the **if** statement, using the operations Label, Arity, and ´.´ (feature selection).

Resposta:

| case <x> of <label>(<feat>¹:<x>¹ ... <feat>n:<x>n) then <s>¹ else <s>² end ::=  if {Label <x>} == <label> then  if {Arity <x>} == [<feat>¹ ... <feat>n] then  local <x>¹ = <x>.<feat>¹ ... <x>n = <x>.<feat>n in <s>¹ end  else <s>² end  else <s>² end |
| --- |

5. This exercise tests your understanding of the full case statement. Given the following procedure:

proc {Test X}

case X of

a|Z then {Browse ´case´(1)}

[] f(a) then {Browse ´case´(2)}

[] Y|Z andthen Y==Z then {Browse ´case´(3)}

[] Y|Z then {Browse ´case´(4)}

[] f(Y) then {Browse ´case´(5)}

else {Browse ´case´(6)}

end

end

Without executing any code, predict what will happen when you feed {Test [b c a]}, {Test f(b(3))}, {Test f(a)}, {Test f(a(3))}, {Test f(d)}, {Test [a b c]}, {Test [c a b]}, {Test a|a}, and {Test ´|´(a b c)}.

Use the kernel translation and the semantics if necessary to make the predictions. After making the predictions, check your understanding by running the examples in Mozart.

Resposta:

| Código núcleo do algoritmo:  declare Test  proc {Test X}  case X of '|'(a Z) then  {Browse 'case'(1)}  else  case X of f(a) then  {Browse 'case'(2)}  else  case X of '|'(Y Z) then  if Y==Z then  {Browse 'case'(3)}  else  case X of '|'(Y Z) then  {Browse 'case'(4)}  else  case X of f(Y) then  {Browse 'case'(5)}  else  {Browse 'case'(6)}  end  end  end  else  case X of '|'(Y Z) then  {Browse 'case'(4)}  else  case X of f(Y) then  {Browse 'case'(5)}  else  {Browse 'case'(6)}  end  end  end  end  end  end  Seguindo o algoritmo acima, os resultados são:  {Test [b c a]} % Resultado: 4  {Test f(b(3))} % Resultado: 5  {Test f(a)} % Resultado: 2  {Test f(a(3))} % Resultado: 5  {Test f(d)} % Resultado: 5  {Test [a b c]} % Resultado: 1  {Test [c a b]} % Resultado: 4  {Test a|a} % Resultado: 1  {Test ´|´(a b c)} % Resultado: 6 |
| --- |

6. Given the following procedure:

proc {Test X}

case X of

f(a Y c) then {Browse ´case´(1)}

else {Browse ´case´(2)} end

end

Without executing any code, predict what will happen when you feed: declare X Y {Test f(X b Y)}

Same for:

declare X Y {Test f(a Y d)}

Same for:

declare X Y {Test f(X Y d)}

Use the kernel translation and the semantics if necessary to make the predictions. After making the predictions, check your understanding by running the examples in Mozart. Now run the following example:

declare X Y

if f(X Y d) == f(a Y c) then

{Browse ´case´(1)}

else

{Browse ´case´(2)}

end

Does this give the same result or a different result than the previous example? Explain the result

Resposta:

| Respostas para os primeiros 3 itens estão presentes no arquivo de código.  Já no quarto item, se houver uma variável não definida, será pulado para a próxima linha e ela será checada, mas com o uso do ‘case’, a checagem se a variável não está definida bloqueia a sequência de execução sem fazer a checagem da próxima. |
| --- |

7. Given the following code:

declare Max3 Max5

proc {SpecialMax Value ?SMax}

fun {SMax X}

if X>Value then X else Value

end

end

{SpecialMax 3 Max3}

{SpecialMax 5 Max5}

end

Without executing any code, predict what will happen when you feed:

{Browse [{Max3 4} {Max5 4}]}

Check your understanding by running this example in Mozart.

Resposta:

| O resultado é [4 5]. (código no arquivo) |
| --- |

8. This exercise explores the relationship between linguistic abstractions and higher-order programming.

(a) Define the function AndThen as follows:

fun {AndThen BP1 BP2}

if {BP1} then {BP2}

else false end

end

Does the following call:

{AndThen fun {$} <expression>¹ end

fun {$} <expression>² end}

give the same result as <expression>¹ andthen <expression>² ? Does it avoid the evaluation of <expression>² in the same situations?

Resposta:

| <expression>¹ andthen <expression>² ::= if <expression>¹ then ::= fun {AndThen BP1 BP2 }  <expression>² if {BP1} then  else {BP2}  false else  end false  end  O resultado é o mesmo, se a primeira condição sempre for falsa a segunda expressão não será chamada, ou seja, se a primeira condição falhar a segunda não irá ser chamada para avaliação. |
| --- |

(b) Write a function OrElse that is to orelse as AndThen is to andthen. Explain its behavior.

Resposta:

| <expression>¹ orelse <expression>² ::= if <expression>¹ then true else <expression>² end  fun {OrElse BP1 BP2}  if {BP1} then  true  else  {BP2}  end  end  A função OrElse que implementa a lógica de orelse faz a checagem da primeira expressão e se for verdadeira automaticamente retorna verdadeiro já que não é necessário determinar o resultado da segunda expressão para que o resultado seja positivo, no entanto, caso a primeira expressão dê falso, a segunda expressão é checada e seu resultado é retornado a função, sendo falso ou positivo. |
| --- |

10. Consider the following function SMerge that merges two sorted lists:

fun {SMerge Xs Ys}

case Xs#Ys

of nil#Ys then Ys

[] Xs#nil then Xs

[] (X|Xr)#(Y|Yr) then

if X=<Y then

X|{SMerge Xr Ys}

else

Y|{SMerge Xs Yr}

end

end

end

Expand SMerge into the kernel syntax. Note that X#Y is a tuple of two arguments that can also be written ´#´(X Y). The resulting procedure should be tail recursive, if the rules of Section 2.5.2 are followed correctly

Resposta:

| declare SMerge R  SMerge = proc {$ Xs Ys ?S}  case Xs of nil then S = Ys  else  case Ys of nil then S = Xs  else  case Xs of X|Xr then  case Ys of Y|Yr then  if X=<Y then  local Mid in  S = X|Mid  {SMerge Xr Ys Mid}  end  else  local Mid in  S = Y|Mid  {SMerge Xs Yr Mid}  end  end  end  end  end  end  end |
| --- |

11. Last call optimization is important for much more than just recursive calls. Consider the following mutually recursive definition of the functions IsOdd and IsEven:

fun {IsEven X}

if X==0 then

true

else

{IsOdd X-1}

end

end

fun {IsOdd X}

if X==0 then

false

else

{IsEven X-1}

end

end

We say that these functions are mutually recursive since each function calls the other. Mutual recursion can be generalized to any number of functions. A set of functions is mutually recursive if they can be put in a sequence such that each function calls the next and the last calls the first. For this exercise, show that the calls {IsOdd N} and {IsEven N} execute with constant stack size for all nonnegative N. In general, if each function in a mutually-recursive set has just one function call in its body, and this function call is a last call, then all functions in the set will execute with their stack size bounded by a constant.

Resposta:

| ([({IsEven X}, {X -> x1})], {x1 = N1}) =>([({IsOdd X}, {X -> x2})], {x2 = N2})  Observando a recorrência dupla é notável que a pilha e armazenamento utilizados pela recorrência seguem uma crescente constante (1 a cada passo dado) aumentando até o valor de X - 1, e o mesmo, para o memória utilizada (seguindo uma regra de memória por função vezes o nível da recursão). |
| --- |