Aluno: Carlos Victor Martins da Silva

Matrícula: 476516

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:

| declare Test  proc {Test X}  case X of f(a Y c) then  {Browse ´case´(1)}  else  {Browse ´case´(2)}  end  end  declare X Y {Test f(a Y d)}  % f(a Y1 d) f(a Y2 c) ? a == a && d == c ? = 'case'(2)  declare X Y {Test f(X Y d)}  % f(x1 y1 d) f(a Y2 c) ? x1 == a && c == d ? = 'case'(2) then block  declare X Y  {Browse f (X Y a) == f (c d b)} % Resultado: false  declare X Y in  if f(X Y d) == f(a Y c) then  {Browse ´case´(1)}  else  {Browse ´case´(2)}  end  end % Resultado: 'case'(2)  No quarto pedaço de código, 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:

| declare Max3 Max5  proc {SpecialMax Value ?SMax}  SMax = fun {$ X}  if X > Value then X else Value  end  end    {SpecialMax 3 Max3}  {SpecialMax 5 Max5}  {Browse [{Max3 4} {Max5 4}]}  end  Acredito que o resultado será [4 5], já que, a função Max irá repassar o valor passado como argumento independente se maior ou menor, então, SpecialMax irá executar e entregar os maiores valores passados como retorno. |
| --- |

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). |
| --- |