

Lista 01 - LIP

Aluno Gabriel Teixeira Queiroz Damasceno - 565118

Questão 1)

Free identifiers: a free identifier in a statement is an identifier that is not defined in that statement. It might be defined in an enclosing statement.

full language:

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

kernel language:

```
P = proc{$ X}
  local T in
    local Y in
      Y = 0
      T = X>Y
      if T then
        local J in
          J = X-1
          {P J}
        end
      end
    end
  end
end
```

Resposta: por definição, um identificador livre em um procedimento corresponde a um identificador que não está presente entre seus parâmetros nem é declarado localmente. Se olharmos para a segunda ocorrência de P, que está dentro do procedimento, vemos que esse identificador não é parâmetro de P, e também que P não é definido dentro da função, o que o torna um identificador livre. Vale dizer que P, isto é, o identificador do procedimento, deveria ser definido do lado de fora para que o programa possa funcionar. Assim, a primeira ocorrência de P seria um identificador ligado, enquanto a segunda corresponderia a um identificador livre (no escopo do proc) que seria mapeado pelo ambiente contextual do procedimento como

sendo um identificador que aponta para o próprio procedimento no qual aparece (categorizando-se como uma chamada recursiva).

Questão 2)

Static scope: The variable corresponding to an identifier occurrence is the one defined in the textually innermost declaration surrounding the occurrence in the source program.

Dynamic scope: The variable corresponding to an identifier occurrence is the one in the most-recent declaration seen during the execution leading up to the current statement.

Esse passo é necessário porque, dentro do procedimento `MulByN`, `N` é um identificador livre, visto que não foi definido no escopo da função (nem em seu corpo nem em seus parâmetros). Assim, quando `{MulByN A B}` é chamado, o ambiente na chamada é $\{A \rightarrow 10, B \rightarrow x1\}$, isto é, não possui o mapeamento de `N`, o que impossibilita que `N*X` seja calculado. Daí, devido ao closure, o que acontece é que o mapeamento `N->3`, definido fora do procedimento, é armazenado no ambiente contextual (CE) de `MulByN`, e é capturado no momento da chamada, sendo, então, adicionado ao ambiente, para que o identificador e seu valor permitam a devida a execução do procedimento. O ambiente contextual, vale dizer, é responsável por armazenar as variáveis externas que serão necessárias ao procedimento, como é o caso de `N`. Obs.: no ambiente da chamada, o mapeamento de `N` não existe; entretanto, no ambiente de definição do procedimento, existe.

Não. Devido ao escopo estático, `N` só está definido no escopo mais interno que envolve sua ocorrência, isto é, no momento em que a função também foi definida. Daí, seu mapeamento é salvo no CE, não importando se `N` foi definido ou não no escopo da chamada.

Exemplo 1:

```
declare MulByN N in
  N=3
  proc {MulByN X ?Y}
    Y=N*X
  end

local A B in
  A = 10
  {MulByN A B}
end
```

Exemplo 2:

```
declare MulByN N in
  N=3
  proc {MulByN X ?Y}
    Y=N*X
  end

local A B N in
  A = 10
  N = 5
  {MulByN A B}
end
```

Obs.: mesmo que N esteja presente no ambiente de chamada ($N \rightarrow 5$), ainda assim, para o procedimento MulByN, esse mapeamento é ignorado, visto que, como já foi dito, devido ao escopo estático, o mapeamento considerado encontra-se no ambiente da definição, que é capturado pelo CE do procedimento.

Questão 4)

item A)

```
if <C> then <s1> else <s2> end
    ≡
declare C
case C of true then <s1>
  [] false then <s2> end
```

Item B)

```

if {Arity X} == {Arity <pattern>} andthen
{Label X} == {Label <pattern>} then
    <s1> /* s1 é qualquer statement que utilize as
        features de X individualmente, X.1, X.2... */
else
    <s2>
end

```

Por exemplo, considere o seguinte programa:

```

local L in
    L=[5 6 7 8]
    case L of H|T then
        {Browse H} {Browse T}
    end
end

```

Escrevamos ele em termos do statement if:

```

local Pattern L H T in
    Pattern = H|T
    L = [5 6 7 8]
    if {Arity L} == {Arity Pattern} andthen
    {Label L} == {Label Pattern} then
        H = L.1
        T = L.2
        {Browse H} {Browse T}
    end
end

```

Questão 5)

{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)}.

A primeira chamada $\{ \text{Test } [b \ c \ a] \}$ entrará no **caso 4**, uma vez que $[b \ c \ a]$ possui formato Y|Z (cabeça “b” e cauda $[c \ a]$). A segunda chamada $\{ \text{Test } f(b(3)) \}$ entrará no **caso 5**, visto que X é um registro com um campo Y qualquer (neste caso, outro registro). A chamada $\{ \text{Test } f(a) \}$ se enquadra no **caso 2**, já que é um registro de label “f” e com um campo “a”. Já no caso da chamada $\{ \text{Test } f(a(3)) \}$, temos novamente um registro que possui um campo que também é um registro, enquadrando-se no **caso 5**. A chamada $\{ \text{Test } f(d) \}$, também por ser um registro com label “f” e um campo diferente de “a”, enquadra-se no **caso 5**. O teste $\{ \text{Test } [a \ b \ c] \}$ se encontra no **caso 1**, pois temos uma lista de cabeça “a”. Já o teste $\{ \text{Test } [c \ a \ b] \}$, como temos uma lista de cabeça diferente de “a”, então enquadra-se no **caso 4**. A chamada $\{ \text{Test } a|a \}$ entra no **caso 1**, visto que temos uma lista com cabeça “a”. Por fim, temos $\{ \text{Test } ' | ' (a \ b \ c) \}$, que não se enquadra a nenhum pattern, fazendo o programa entrar na instrução “else” e exibir o **caso 6**.

revisar: registros e listas

Questão 7)

$\{ \text{Browse } [\{ \text{Max3 } 4 \} \ \{ \text{Max5 } 4 \}] \}$

O procedimento SpecialMax funciona como um “construtor” de funções SMax. O SpecialMax recebe um valor (Value), correspondente ao primeiro parâmetro, e o usa para comparar com o parâmetro X de SMax, além de nomear a função utilizando o identificador recebido por seu segundo parâmetro. Daí, com $\{ \text{SpecialMax } 3 \ \text{Max3} \}$, há a criação de uma função de identificador Max3 e que contém o mapeamento Value->3 em seu ambiente contextual (CE). Análogo para $\{ \text{SpecialMax } 5 \ \text{Max5} \}$. Portanto, na chamada $\{ \text{Max3 } 4 \}$, a função Max3 recebe 4 como argumento e realiza a comparação $X > \text{Value}$. Como $X \rightarrow 4$, pelo argumento, está mapeado e $\text{Value} \rightarrow 3$, no CE de Max3, também está, a comparação $4 > 3$ é realizada. Por 4, ligado a X, ser maior do que 3, a comparação é verdadeira, o programa entra na instrução “if”, e $X = 4$ é retornado. O caso de Max5 é semelhante, visto que realiza a comparação $4 > 5$, e, como essa comparação é falsa, o programa executa a instrução “else” e retorna Value, que é ligado a 5 no CE de Max5. Portanto, o programa exibe $[4 \ 5]$.

Questão 8)

Item A)

Sim. A função AndThen recebe duas funções (programação de alta ordem) que retornam valores booleanos. Na instrução “if” do AndThen, temos a seguinte semântica: se BP1 for falso, então a função retornará falso (a primeira expressão já é falsa); se BP1 for verdadeiro, a função retornará o valor de BP2, isto é, verdadeiro se BP2 for verdadeiro (ambas as expressões verdadeiras) e falso se BP2 for falso (uma expressão verdadeira e outra falsa). Portanto, essa lógica reflete justamente a maneira como a abstração linguística andthen funciona: semelhante ao operador “and”, com a diferença de que, se a primeira expressão for falsa, o andthen já entende que `<expression>1 and <expression2>` é uma afirmação falsa, sem precisar verificar a segunda expressão.

Item B)

```
fun {OrElse BP1 BP2}
  if {BP1} then
    true
  else
    {BP2}
  end
end
```

Se a expressão de BP1 for verdadeira, a função OrElse já retornará verdadeiro sem verificar a segunda expressão (uma das expressões é verdadeira). Por outro lado, se BP1 for falso, então o programa entrará na instrução “else” e retornará o valor booleano de BP2, isto é, se BP2 for falso, o programa retornará falso (ambas as expressões são falsas); se BP2 for verdadeiro, o programa retornará verdadeiro (uma das expressões é verdadeira).

revisar: andthan e orelse (From kernel language to practical language)

Questão 9)

full language:

```
fun {Sum1 N}
  if N==0 then 0 else N+{Sum1 N-1} end
end
```

```
fun {Sum2 N S}
  if N==0 then S else {Sum2 N-1 N+S} end
end
```

kernel language:

```
local Sum1 in
  Sum1 = proc {$ N ?I}
    local T in
      local Z in
        Z = 0
        T = N == Z
        if T then
          I = Z
        else
          local J in
            local M in
              M = N-1
              J = {Sum1 M}
              I = N + J
            end
          end
        end
      end
    end
  end
end
end
end
end
end
```

```

local Sum2 in
  Sum2 = proc {$ N S ?R}
    local Z in
      local T in
        Z = 0
        T = N == Z
        if T then
          R = S
        else
          local M in
            local I in
              M = N-1
              I = N+S
              R = {Sum2 M I}
            end
          end
        end
      end
    end
  end
end
end
end

```

Portanto, por intermédio da kernel language, nota-se que Sum1 possui uma instrução a ser feita após a chamada recursiva, o que não a configura como chamada recursiva pela cauda. Já em Sum2, vemos que não há mais nenhuma operação a ser realizada após a chamada, portando Sum 2 é recursivo pela cauda.

Item B)

Procedimento Sum1:

$((\langle s \rangle, \emptyset) \emptyset) \Rightarrow$

$((\langle s1 \rangle, \{ \text{Sum1} \rightarrow s, X \rightarrow x \}), \{s, x\}) \Rightarrow$

Chamada da função

$((\{ \text{Sum1 } X \}, \{ \text{Sum1} \rightarrow s, X \rightarrow x \}), \{s = (\text{proc } \{ \$ N \} \langle s2 \rangle \text{ end}, \{ \text{Sum1} \}), x = 10\}) \Rightarrow$

$((\langle s2 \rangle, \{ N \rightarrow x, Z \rightarrow z, T \rightarrow t \}),$

$\{s = \text{proc} \dots, x = 10, z = 0, t = \text{false}\}) \Rightarrow$

$((\langle s3 \rangle, \{ J \rightarrow j, M \rightarrow m, I \rightarrow i \}), \{s = \text{proc} \dots, x = 10, z = 0,$
 $t = \text{false}, j, m, i\}) \Rightarrow$

$((\langle I = N + J \rangle, \{ J \rightarrow j, M \rightarrow m, I \rightarrow i \}), \{s = \text{proc} \dots,$
 $x = 10, z = 0, t = \text{false}, j = \{ \text{Sum1 } m \}, m = 9, i = 10 + j\}) \Rightarrow$

$((\langle I = N + \{ \text{Sum1 } M \} \rangle, \{ \text{Sum1} \rightarrow s, J \rightarrow j', M \rightarrow m',$
 $I \rightarrow i' \}), \{s = \text{proc} \dots, x = 10, x' = 9, z = 0, t = \text{false}, j = \{ \text{Sum1 } m \},$
 $j' = \{ \text{Sum1 } m' \}, m = 9, m' = 8, i = 10 + j, i' = 9 + j'\}) \Rightarrow$

$((\langle I = N + (N + \{ \text{Sum1 } M \}) \rangle, \{ \text{Sum1} \rightarrow s, J \rightarrow j'', M \rightarrow m'',$
 $I \rightarrow i'' \}), \{s = \text{proc} \dots, x = 10, x' = 9,$
 $x'' = 8, z = 0, t = \text{false}, j = \{ \text{Sum1 } m \}, j' = \{ \text{Sum1 } m' \}, j'' = \{ \text{Sum1 } m'' \},$
 $m = 9, m' = 8, m'' = 7, i = 10 + j, i' = 9 + j', i'' = 8 + j''\}) \Rightarrow$

(...)

Nota-se que, além de o tamanho do store aumentar, o tamanho da pilha também aumenta.

Obs.: essa execução de Sum1 na máquina abstrata provavelmente está errada, pois não sei como realizá-la em um procedimento que é recursivo e que não tem recursão pela cauda.

Procedimento Sum2:

Começando pela chamada

$((\langle \{ \text{Sum2 } X Y \} \rangle, \{ \text{Sum2} \rightarrow s, X \rightarrow x, Y \rightarrow y \}), \{s = (\text{proc } \{ \$ N S \} \langle s \rangle \text{ end}, \{ \text{Sum2} \rightarrow s \}), x = 10, y = 0\}) \Rightarrow$

$((\langle M = N - 1 \ I = N + s \ \{ \text{Sum2 } M I \} \rangle, \{ \text{Sum2} \rightarrow s, N \rightarrow x,$
 $S \rightarrow y, M \rightarrow m, I \rightarrow i \}), \{s = \text{proc} \dots, x = 10, y = 0, m = 9, i = 10\})$
 \Rightarrow

$((\langle \text{Sum2 } M \text{ I} \rangle, \{ \text{Sum2} \rightarrow s, M \rightarrow x', I \rightarrow y' \}),$
 $\{s = \text{proc} \dots, x = 10, x' = 9, y = 0, y' = 10\}) \Rightarrow$

$((\langle \text{Sum2 } M \text{ I} \rangle, \{ \text{Sum2} \rightarrow s, M \rightarrow x'', I \rightarrow y'' \}),$
 $\{s = \text{proc} \dots, x = 10, x' = 9, x'' = 8, y = 0, y' = 10, y'' = 19\}) \Rightarrow$

$((\langle \text{Sum2 } M \text{ I} \rangle, \{ \text{Sum2} \rightarrow s, M \rightarrow x''', I \rightarrow y''' \}),$
 $\{s = \text{proc} \dots, x = 10, x' = 9, x'' = 8, x''' = 7, y = 0, y' = 10, y'' = 19,$
 $y''' = 27\}) \Rightarrow$
 (\dots)

Nota-se que, apesar do store aumentar, o tamanho da pilha é constante.

Item C)

Na chamada $\{\text{Browse } \{\text{Sum1 } 100000000\}\}$, o programa dá um erro de excesso de memória devido ao stack overflow. Já no caso da chamada $\{\text{Sum2 } 100000000 \text{ } 0\}$, o programa é capaz de executar devido à otimização de chamada pela cauda, que permite que o tamanho da pilha tenha tamanho constante, e o resultado exibido foi

500000000500000000.

Questão 10)

kernel language com chamada pela cauda:

```
declare SMerge
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
          local T in
            local J in
              T = X=<Y
              if T then
                S = X|J
                {SMerge Xr Ys J}
              else
                S = Y|J
                {SMerge Xs Yr J}
              end
            end
          end
        end
      end
    end
  end
end
end
end
end
end
end
end
end
```

Como '|' (cons) não pede valor no momento que é utilizado, podendo manter um valor parcial, então tanto X|J quanto Y|J podem ser colocados antes da chamada recursiva, utilizando J como variável não ligada. Dessa forma, é possível utilizar a chamada pela cauda.

Revisar: tuplas, valor parcial, cons

Questão 11)

Para mostrar que as funções mutuamente recursivas IsOdd e IsEven executam com tamanho de pilha constante - isto é, possuem a otimização da chamada pela cauda - basta substituir a chamada interna de uma das funções pela sua respectiva definição, assim revelando a chamada pela cauda diretamente. Por exemplo:

```
fun {IsEven X}
  if X==0 then
    true
  else
    if X==0 then
      false
    else
      {IsEven X-1}
    end
  end
end
end
```

Aqui, a chamada {IsOdd X-1}, dentro de IsEven, foi trocada pela definição de IsOdd, que contém uma chamada de IsEven. Assim, não há mais recursão mútua, mas a chamada pela cauda é vista de forma direta, enquanto anteriormente estava implícita. Análogo para a função IsOdd.