Universidade Federal de Minas Gerais
Ciência da Computação

Linguagens de Programação - Haniel Barbosa

Lista de Exercícios 2

Respostas

1 Linguagem de expressões

1. Utilizando o ADT `expr` visto em aula:

   `datatype expr = IConst of int | Plus of expr * expr | Minus of expr * expr;`

   Estenda essa linguagem com os seguintes operadores:

   - Multiplicação. `(Multi)` ⇒ multiplica dois valores.
   - Divisão. `(Div)` ⇒ divisão inteira de dois valores. Divisão por 0 deve retornar 0.
   - Maior valor. `(Max)` ⇒ retorna o maior de dois valores.
   - Menor Valor. `(Min)` ⇒ retorna o menor de dois valores.
   - Igual. `(Eq)` ⇒ retorna 1 se os valores são iguais, 0 caso contrário.
   - Maior que. `(Gt)` ⇒ retorna 1 se o primeiro valor é estritamente maior que o segundo, e 0 caso contrário.

   Estenda também a função `eval : expr -> int` para conseguir avaliar expressões que utilizem esses operadores.

   input: val e1 = Max(IConst 3, Plus(IConst 2, IConst 3));
   output: val it = 5 : int

   input: val e2 = Div(Multi(IConst 5, IConst 4), Minus(IConst 4, IConst 4));
   output: val it = 0 : int

   Solution:

```
1 datatype expr = IConst of int | Plus of expr * expr | Minus of
     expr * expr | Multi of expr * expr | Div of expr * expr |
     Max of expr * expr | Min of expr * expr | Eq of expr * expr
     | Gt of expr * expr;
2
3
4 fun eval (IConst i) = i
5   | eval (Plus(e1, e2)) = (eval e1) + (eval e2)
6   | eval (Minus(e1, e2)) = (eval e1) - (eval e2)
7   | eval (Multi (e1, e2)) = (eval e1) * (eval e2)
8   | eval (Div (e1, e2)) =
```

```
    let val r1 = eval(e1) val r2 = eval(e2)
      in
        if (r1 = 0) orelse (r2 = 0) then
          0
        else
          (eval e1) div (eval e2)
      end
  | eval (Max (e1,e2)) =
      if (eval e1) > (eval e2) then (eval e1) else (eval e2)
  | eval (Min (e1,e2)) =
      if (eval e1) < (eval e2) then (eval e1) else (eval e2)
  | eval (Eq (e1,e2)) =
      if (eval e1) = (eval e2) then 1 else 0
  | eval (Gt (e1,e2)) =
      if (eval e1) > (eval e2) then 1 else 0;
```

2. Escreva uma linguagem que calcule a área de objetos quadrados, retangulares, e circulares. Você deve definir o ADT `area`. Os nomes dos construtores de `area` devem seguir o seguinte padrão:

   `datatype area = RConst of real | AQuadrado of area | ACirculo ...`

   Defina também a função `eval : area -> real` para realizar a interpretação dessas expressões. **IMPORTANTE:** as medidas desses objetos deveram ser do tipo `real`.

   input: val e = ACirculo(RConst 2.0);
   output: val it = 12.56: real;

   Solution:

```
datatype area  = RConst of real | AQuadrado of area |
   ARetangulo of area * area | ACirculo of area;

fun eval (RConst a) = a
  | eval (AQuadrado(a)) = eval(a) * eval(a)
  | eval (ARetangulo(a1,a2)) = eval(a1) * eval(a2)
  | eval (ACirculo (a)) = 3.14 * eval(a) * eval(a);
```

3. Utilizando as mesmas convenções da questão anterior, defina um ADT `perimetro` e sua função `eval : perimetro -> real`. Inclua também suporte para calcular perímetro de triângulos (`PTriangulo`).

   input: val p = PQuadrado(RConst 4.0);
   output: val it = 16.0: real;

   Solution:

```
datatype perimetro  = RConst of real | PQuadrado of perimetro |
    PRetangulo of perimetro * perimetro | PCirculo of perimetro
    | PTriangulo of perimetro * perimetro * perimetro;

```

```
fun eval (RConst p) = p
  | eval (PQuadrado(l)) = 4.0 * eval(l)
  | eval (PRetangulo(b,h)) = 2.0 * eval(b) * eval(h)
  | eval (PCirculo (r)) = 3.14 * 2.0  * eval(r)
  | eval (PTriangulo (l1,l2,l3)) = (eval  l1)+ (eval  l2) + (
     eval l3);
```

4. Compiladores frequentemente aplicam otimizações com intuito de deixar o código gerado mais eficiente. Um exemplo comum é a simplificação de expressões, utilizando propriedades aritméticas e/ou Booleanas. Dados os ADT's abaixo:

   ```
   datatype UnOp = Not;
   ```

   ```
   datatype BinOp = Add | Sub | Mul | Gt | Eq | Or;
   ```

   ```
   datatype Sexpr = IConst of int | Op1 of UnOp * Sexpr | Op2 of BinOp * Sexpr
   * Sexpr;
   ```

   Escreva uma função `simplify : Sexpr -> Sexpr` que seja capaz de simplificar expressões `Sexpr` de acordo com as regras de simplificação listadas abaixo ($\vee$ simboliza disjunção lógica e $\neg$ simboliza negação lógica):

$$0 + e \to e$$
$$e + 0 \to e$$
$$e - 0 \to e$$
$$1 * e \to e$$
$$e * 1 \to e$$
$$0 * e \to 0$$
$$e * 0 \to 0$$
$$e - e \to 0$$
$$e \vee e \to e$$
$$\neg(\neg e) \to e$$

   A sua função deve ser capaz de simplificar, por exemplo, as expressões $x + 0$, $1 * x$, e $(1 + 0) * (x + 0)$ para somente x. O retorno deve ser uma expressão que não possa ser mais simplificada.

   DICAS:

   - Não se esqueça dos casos em que não é possível mais simplificar.
   - Passos recursivos podem ser necessários para produzir expressões que não são simplificáveis.
   - "You ain't never had a friend like pattern matching."

//(1+0)*(9+0) → 9
input: Op2(Mul, Op2(Add, IConst 1, IConst 0), Op2(Add, IConst 9, IConst 0));
output: val it = IConst 9: Sexpr;

//(1+0)*((10 ∨ 12) +0) → (10 ∨ 12)
input: Op2 (Mul, Op2 (Add, IConst 1, IConst 0), Op2 (Add, Op2 (Or, IConst 10, IConst 12), IConst 0)): Sexpr;
output: val it = Op2 (Or, IConst 10, IConst 12): Sexpr;

Solution:

```
datatype UnOp = Not;
datatype BinOp = Add | Sub | Mul | Or | Eq | Gt;

datatype Sexpr = IConst of int | Op1 of UnOp * Sexpr | Op2 of
   BinOp * Sexpr * Sexpr;

fun simplify (Op2(Add, IConst 0, e))   = simplify e
    | simplify (Op2(Add, e, IConst 0)) = simplify e
    | simplify (Op2(Mul, IConst 1, e)) = simplify e
    | simplify (Op2(Mul, e, IConst 1)) = simplify e
    | simplify (Op2(Mul, _, IConst 0)) = IConst 0
    | simplify (Op2(Mul, IConst 0, _)) = IConst 0
    | simplify (Op2(Sub, e1, e2)) = if (simplify e1 = simplify
       e2) then IConst 0 else
      if (simplify e2 = IConst 0) then simplify e1 else (Op2(
         Sub, simplify e1, simplify e2))
    | simplify (Op2(Or, e1, e2)) = if (simplify e1 = simplify
       e2) then (simplify e1)
      else (Op2(Or, simplify e1, simplify e2))
    | simplify (Op1(Not, Op1(Not, e1))) = simplify e1
    | simplify e =
      case e of
        (Op1(oper, e1)) =>
          let
            val f1 = simplify e1
          in
            if (f1 = e1) then e else (simplify(Op1(oper, f1)))
          end
        | (Op2(oper, e1, e2)) =>
          let
            val f1 = (simplify e1) val f2 = (simplify e2)
          in
            if ((f1 = e1) andalso (f2 = e2)) then e
            else (simplify(Op2(oper, f1, f2)))
          end
    | _ => e;
```

```
33
34 val x = Op2(Or, IConst 10, IConst 12);
35 val p1 = Op2(Add, IConst 1, IConst 0);
36 val p2 = Op2(Add, x, IConst 0);
37 val i = Op2(Mul, p1, p2);
38 simplify i;
```

2 Semântica formal

As questões a seguir são retiradas do livro *Semantics with Applications: An Appetizer*, Capítulos 1 e 2. Portanto a leitura é necessária para o entendimento das questões. Os dois primeiros capítulos encontram-se disponíveis no Moodle.

1. Suponha que o valor inicial da variável x é n e o valor inicial de y é m. Escreva um programa em **WHILE** que atribui a Z o valor de n^m.

   Solution:

   x:=n;z:=1;while ¬ (m=0) do (z:=z*x;m:=m-1)

2. Considere a função de interpretação de expressões Booleanas de **WHILE**, $\mathcal{B}$ (Tabela 1.2), e suponha que $s\ x = 3$. Determine $\mathcal{B}[\![\neg(x = 1)]\!]$.

   Solution:

   $$\begin{aligned}\beta[\![\neg(x=1)]\!]s \ ?&= \beta[\![(x=1)]\!]s\\ ?&= A[\![x]\!]s = A[\![1]\!]s\\ ?&= sx = N[\![1]\!]\\ ?&= 3 = 1\\ &= \texttt{ff}\\ &\beta[\![\neg(\texttt{ff})]\!] \Rightarrow \texttt{tt}\end{aligned}$$

3. Defina uma substituição para expressões Booleanas de **WHILE**: $b[y \mapsto a_0]$ deve ser a expressão Booleana correspondente a b exceto que todas as ocorrências da variável y são substituídas pela expressão aritmética a_0.

   Solution:

   $\texttt{true}\ [y \mapsto a_0] = \texttt{tt}$
   $\texttt{false}\ [y \mapsto a_0] = \texttt{ff}$
   $(a_1 = a_2)[y \mapsto a_0] = a_1[y \mapsto a_0] = a_2[y \mapsto a_0]$
   $(a_1 \leq a_2)[y \mapsto a_0] = a_1[y \mapsto a_0] \leq a_2[y \mapsto a_0]$
   $(\neg a_1)[y \mapsto a_0] = \neg(b[y \mapsto a_0])$
   $(a_1 \wedge a_2)[y \mapsto a_0] = \mathrm{b}(a_1[y \mapsto a_0]) \wedge \mathrm{b}(a_2[y \mapsto a_0])$

4. Dado o programa

   $$\mathsf{z := 0;\ while\ y} \leq \mathsf{x\ do\ (z := z+1;\ x := x\text{-}y)}$$

construa uma árvore de derivação para este programa quando executado em um estado em que x tem valor **17** e y tem valor **5**.
T_0:

$$\dfrac{\dfrac{T_1}{\langle bodyC,\ s_{0,5,17}\rangle \to s_{1,5,12}} \quad \dfrac{T_2}{\langle bodyC,\ s_{1,5,12}\rangle \to s_{2,5,7}} \quad \dfrac{T_3}{\langle bodyC,\ s_{2,5,7}\rangle \to s_{3,5,2}} \quad \dfrac{}{\langle \text{while y} \leq \text{x do } bodyC,\ s_{3,5,2}\rangle \to s_{3,5,2}}\ while^{ff}_{ns}}{\langle \text{while y} \leq \text{x do } bodyC,\ s_{2,5,12}\rangle \to s_{3,5,2}}\ while^{tt}_{ns}}{\langle \text{while y} \leq \text{x do } bodyC,\ s_{1,5,17}\rangle \to s_{3,5,2}}\ while^{tt}_{ns}}{\langle \text{z := 0; while y} \leq \text{x do (z := z+1; x := x-y)},\ s_{0,5,17}\rangle \to s_{3,5,2}}$$

$$\dfrac{\dfrac{}{\langle \text{z := 0},\ s_{5,17}\rangle \to s_{0,5,17}}\ ass_{ns} \qquad T_0}{\langle \text{z := 0; while y} \leq \text{x do (z := z+1; x := x-y)},\ s_{5,17}\rangle \to s_{3,5,2}}\ comp_{ns}$$

em que $s_{a,b,c} = s[z \mapsto a][y \mapsto b][x \mapsto c]$, $bodyC = (\text{z := z+1; x := x-y})$

5. Considere os seguintes programas:

   - while ¬(x=1) do (y:=y*x; x:=x-1)
   - while 1 ≤ x do (y:=y*x; x:=x-1)
   - while true do skip

   Para cada um deles determine se eles *sempre* terminam ou se sempre entram em um laço infinito. Tente embasar suas repostas usando os axiomas e regras da Tabela 2.1.

   - while ¬(x=1) do (y:=y*x; x:=x-1)
     Não é possível determinar se esse programa sempre termina ou sempre entra em loop. A regra aplicada a while vai depender do valor de x no estado s. E o comando x:=x-1 produz um novo estado s' onde x pode possuir um valor que faça com que a condição do comando while seja falsa, fazendo com o que o programa sempre termine. Contudo, por exemplo, dado um estado inicial s em que o valor de x é menor que 1, o estado s' produzido pela atribuição de x nunca construirá um valor de x que seja igual a 1, levando o programa a sempre entrar num laço infinito.
   - while 1 ≤ x do (y:=y*x; x:=x-1)
     Esse programa sempre termina. Dado um estado s onde x possui um valor maior que 1, o programa automaticamente termina. Dado um estado onde x não é maior que 1, o comando de atribuição x:x-1 produzirá um novo estado s' onde o valor de x sempre é menor do que no estado anterior s, o que fatalmente fará com que a avaliação da condição do comando while seja falsa. Nesse momento, a regra aplicada será while`ff`, produzindo o estado final do programa.
   - while true do skip
     Esse programa sempre entra num laço infinito. A condição do comando while é sempre verdadeira, logo sempre aplicaremos a regra while`tt`. Note que o comando executado por while, skip, por definição não muda o estado s, logo a condição do comando while nunca é alterada. Sendo que a aplicação da regra while`tt` ocorreria de forma infinita.

6. Considere a seguinte AST e interpretador parcial para linguagem **WHILE**:

```
type Num = int;
type Var = string;

datatype Aexpr =
         N of Num
         | V of Var
         | Plus of Aexpr * Aexpr
         | Mult of Aexpr * Aexpr
         | Minus of Aexpr * Aexpr;

datatype Bexpr =
         True
       | False
       | Eq of Aexpr * Aexpr
       | Leq of Aexpr * Aexpr
       | Not of Bexpr
       | And of Bexpr * Bexpr;

datatype Stm =
         Assign of Var * Aexpr
       | Skip
       | Comp of Stm * Stm
       | If of Bexpr * Stm * Stm
       | While of Bexpr * Stm;

fun evalN n : Num = n

exception FreeVar;
fun lookup [] id = raise FreeVar
  | lookup ((k:string, v)::l) id = if id = k then v else lookup
      l id;

fun evalA (N n) _ = evalN n
  | evalA (V x) s = lookup s x
  | evalA (Plus(e1, e2)) s = (evalA e1 s) + (evalA e2 s)
  | evalA (Mult(e1, e2)) s = (evalA e1 s) * (evalA e2 s)
  | evalA (Minus(e1, e2)) s = (evalA e1 s) - (evalA e2 s);

fun evalB True _ = true
  | evalB False _ = false
  | evalB (Eq(a1, a2)) s = (evalA a1 s) = (evalA a2 s)
  | evalB (Leq(a1, a2)) s = (evalA a1 s) <= (evalA a2 s)
  | evalB (Not b) s = not (evalB b s)
  | evalB (And(b1, b2)) s = (evalB b1 s) andalso (evalB b2 s);

fun evalStm (stm : Stm) (s : (string * int) list) : (string *
```

```
   int) list =
46  case stm of
47      (Assign(x, a)) => (x, evalA a s)::s
48    |  Skip => s
49    | (Comp(stm1, stm2)) => evalStm stm2 (evalStm stm1 s)
50    | (If(b, stm1, stm2)) =>
51      if (evalB b s) then evalStm stm1 s else evalStm stm2 s
52    (* | While(b, stm) =>  ... *)
53    | _ => raise Match;
```

(a) Estenda o interpretador com o tratamento do comando while, seguindo a semântica da Tabela 2.1.

```
| (While(b, stm1)) =>
        if (evalB b s) then
          let
             val newS = evalStm stm1 s
          in
            evalStm stm newS
          end
        else s
```

(b) Estenda a linguagem com o comando

$$\text{repeat } S \text{ until } b$$

e defina a relação $\rightarrow$ para ele. (A semântica de repeat não deve utilizar o operador while da linguagem).

$$\frac{\langle S, s\rangle \rightarrow s' \ , \ \langle \text{ repeat S until b } \rangle s' \rightarrow s'' \quad \text{if } \beta[\![b]\!]s' = \texttt{ff}}{\langle \text{ repeat S until b } \rangle s \rightarrow s''}$$

$$\frac{\langle S, s\rangle \rightarrow s' \qquad \text{if } \beta[\![b]\!]s' = \texttt{tt}}{\langle \text{ repeat S until b } \rangle s \rightarrow s'}$$

(c) Estenda o interpretador acima para o comando repeat.

```
datatype Stm =
         Assign of Var * Aexpr
       | Skip
       | Comp of Stm * Stm
       | If of Bexpr * Stm * Stm
       | While of Bexpr * Stm
       | Repeat of Stm * Bexpr;

       ...


| (Repeat(stm1, b)) =>
        let
```

```
            val newS = evalStm stm1 s
            val evB = evalB b newS
        in
          if evB then newS else evalStm stm newS
        end
```

(d) Demonstre que repeat S until b e S ; if b then skip else (repeat S until b) são semanticamente equivalentes.

Solution: Considere que b seja uma condição falsa. Para S if b then skip else (repeat S until b) são geradas ao final duas derivações que correspondem à execução de S no estado s gerando o estado s', e ao resultado do if no estado s' que é a expressão repeat S until b no estado s' (Veja a primeira regra de derivação abaixo). Para repeat S until b serão geradas duas derivações também, que também são a execução de S no estado s e a expressão repeat S until b no estado s' (Veja a segunda regra de derivação abaixo). Note que ambos os programas se derivam nas mesmas expressões aplicadas aos mesmos estados, elas são equivalentes.

$$\frac{\langle S, s\rangle \rightarrow s' \ , \ \langle \text{ repeat S until b, s' } \rangle \rightarrow s'' \qquad \text{if } \beta[\![b]\!]s' = \texttt{ff}}{\langle \text{ S; if b then skip else (repeat S until b), s } \rangle \rightarrow s''}$$

$$\frac{\langle S, s\rangle \rightarrow s', \ \langle \text{ repeat S until b, s' } \rangle \rightarrow s'' \qquad \text{if } \beta[\![b]\!]s' = \texttt{tt}}{\langle \text{ repeat S until b, s } \rangle \rightarrow s'}$$

Considere que b seja uma condição verdadeira. Para repeat S until b será gerada uma derivação e terá como resultado a execução de S no estado s, que resulta em s'. Para S ; if b then skip else (repeat S until b) são geradas no final duas derivações, que correspondem à execução de S no estado s gerando o estado s', e um skip aplicado a s' que resulta no próprio s'. Note que ambos os programas se derivam nas mesmas expressões aplicadas aos mesmos estados, elas são equivalentes.

$$\frac{\langle \text{ S,s } \rangle \rightarrow s' \qquad \text{if } \beta[\![b]\!]s' = \texttt{ff}}{\langle \text{ repeat S until b,s } \rangle \rightarrow s'}$$

$$\frac{\langle S, s\rangle \rightarrow s', \ \langle \text{ if b then skip else (repeat S until b), s' } \rangle \rightarrow s' \qquad \text{if } \beta[\![b]\!]s' = \texttt{tt}}{\langle \text{ S; if b then skip else (repeat S until b), s } \rangle s \rightarrow s'}$$

$$\frac{\langle \text{ skip, s' } \rangle \rightarrow \text{s'}}{}$$

Logo, como os dois programas geram os mesmos resultados com as mesmas entradas, eles são semanticamente equivalentes.

Binding, escopo

1. Considere o seguinte programa em uma linguagem de programação genérica:

```
func p {
    x: integer;
```

```
3
4       func q{
5           x := x + 1;
6       }
7
8       func r{
9           x: integer;
10          x := 1;
11          q;
12          write(x);
13      }
14
15      x := 2;
16      r;
17  }
```

Qual é a saída desse programa (linha 12) ao executar p:

(a) Caso essa linguagem possua escopo estático?

Nesse caso, a saída seria 1. Pois a função q estaria incrementando a variável x definida no escopo de p. Logo, o valor de x em r se manteria o mesmo.

(b) Caso esta linguagem possua escopo dinâmico?

Nesse caso, a saída seria 2. Pois a função q estaria incrementando a variável x definida no escopo de r, a função que chama q, em outras palavras, seu *callee*.

2. Considere o programa abaixo, escrito em SML:

```
1  fun g x =
2    let
3      val inc = 1
4      fun f y = y + inc
5      fun h z =
6        let
7          val inc = 2
8        in
9          f z
10       end
11   in
12     h x
13   end
```

(a) Enumere cada bloco que esse programa contém. Exemplo: Escopo de g = bloco 1, etc. Bloco 1 = Escopo de g. Bloco 2 = Escopo de let (linha 2). Bloco 3 = Escopo de f. Bloco 4 = Escopo de h. Bloco 5 = escopo de let (linha 6).

(b) Quais são os nomes definidos nesse programa? g, x, inc(linha 3), f, y, h, z, inc((linha 7))

(c) Para cada definição, descreva seu escopo em termos dos números de blocos que você definiu no item a. g→ Bloco 1, x→ Bloco 1, inc(linha 3) → Bloco 2, f → Bloco 2, y → Bloco 3, h→ Bloco 2, z→ Bloco 4, inc(linha 7)→ Bloco 5.

(d) Tente responder sem executar o programa. Qual é o valor de `g 5`? Qual seria esse valor se SML possuísse escopo dinâmico? Explique o motivo de esses valores serem diferentes.

A saída do programa é 6. Caso SML possuísse escopo dinâmico, esse valor seria 7. Caso o escopo fosse dinâmico, o valor de `inc` na função f seria amarrado à definição de `inc` dentro de h, pois h chama f. No caso de escopo estático, como não há definição de `inc` no corpo de f, essa definição é amarrada ao valor definido no bloco do operador `let` de `g`, pois é o bloco que contém a definição de f.

3 Binding, escopo

1. Considere o seguinte programa em uma linguagem de programação genérica:

```
1  func p {
2    x: integer;
3
4    func q{
5        x := x + 1;
6    }
7
8    func r{
9        x: integer;
10       x := 1;
11       q;
12       write(x);
13   }
14
15   x := 2;
16   r;
17 }
```

Qual é a saída desse programa (linha 12) ao executar p:

(a) Caso essa linguagem possua escopo estático?

Nesse caso, a saída seria 1. Pois a função q estaria incrementando a variável x definida no escopo de p. Logo, o valor de x em r se manteria o mesmo.

(b) Caso esta linguagem possua escopo dinâmico?

Nesse caso, a saída seria 2. Pois a função q estaria incrementando a variável x definida no escopo de r, a função que chama q, em outras palavras, seu *callee*.

2. Considere o programa abaixo, escrito em SML:

```
1 fun g x =
```

```
2  let
3    val inc = 1
4    fun f y = y + inc
5    fun h z =
6      let
7        val inc = 2
8      in
9        f z
10     end
11 in
12   h x
13 end
```

(a) Enumere cada bloco que esse programa contém. Exemplo: Escopo de g = bloco 1, etc. Bloco 1 = Escopo de g. Bloco 2 = Escopo de let (linha 2). Bloco 3 = Escopo de f. Bloco 4 = Escopo de h. Bloco 5 = escopo de let (linha 6).

(b) Quais são os nomes definidos nesse programa? g, x, inc(linha 3), f, y, h, z, inc((linha 7))

(c) Para cada definição, descreva seu escopo em termos dos números de blocos que você definiu no item a. g→ Bloco 1, x→ Bloco 1, inc(linha 3) → Bloco 2, f → Bloco 2, y → Bloco 3, h→ Bloco 2, z→ Bloco 4, inc(linha 7)→ Bloco 5.

(d) Tente responder sem executar o programa. Qual é o valor de `g 5`? Qual seria esse valor se SML possuísse escopo dinâmico? Explique o motivo de esses valores serem diferentes.

A saída do programa é 6. Caso SML possuísse escopo dinâmico, esse valor seria 7. Caso o escopo fosse dinâmico, o valor de `inc` na função f seria amarrado à definição de `inc` dentro de h, pois h chama f. No caso de escopo estático, como não há definição de `inc` no corpo de f, essa definição é amarrada ao valor definido no bloco do operador `let` de `g`, pois é o bloco que contém a definição de f.

3. Escreva uma função principal count_main: `int -> int` list que receba um número inteiro como entrada e conte de 1 em 1 até chegar neste número. Escreva uma segunda função count: `int -> int` list que receba o primeiro número que deva começar a contagem e faça qualquer regra necessária dentro desta função. A função count deve funcionar apenas dentro do escopo da função count_main.

//sendo 1 o primeiro número
input: count_main(5);
output: val it = [1, 2, 3, 4, 5] : int list

Solução:

```
1  fun count (from : int, to : int) =
2      if from = to
```

```
3       then to::[]
4           else from :: count(from + 1, to);
5
6   fun count_main (x : int) =
7       let
8           fun count (y : int) =
9           if y = x
10          then x::[]
11              else y :: count(y + 1);
12      in count(1)
13  end;
```

4. Escreva uma função pow: `int -> int` que receba um número e retorne a seu resultado elevado a 2, utilize uma outra função calculePow: `int -> int` para fazer o cálculo, ela deve estar dentro do escopo da função pow.

   input: pow(3);
   output: val it = 9 : int

   Solução:

```
1   fun calculatorPow (x: int) =
2       let
3           fun pow(x : int, y: int) =
4           if y = 0
5           then 1
6               else x * pow(x, y - 1)
7       in pow (x, 2)
8   end;
```

5. A seguinte função bad_max que confere o maior número de uma lista e o retorna:

```
1  fun bad_max(xs: int list) =
2  if null xs
3  then 0
4  else if null (tl xs)
5  then hd xs
6  else if hd xs > bad_max(tl xs)
7  then hd xs
8  else bad_max(tl xs);
```

   (a) Explique porque essa função bad_max apesar de funcionar não é a forma mais correta de ser implementada.

   Se testarmos com valores mais altos para a função auxiliar countup podemos perceber uma demora no resultado. Isso acontece para cada comparação na linha 18

que iremos efetuar.
A justificativa anterior já seria uma resposta válida
Isso acontece a cada comparação que iremos efetuar ao chamar a recursão, porque por exemplo iremos chamar ela duas vezes para conferir o número 25, se tivessemos o countup indo do 1 ao 25 por exemplo, depois teríamos mais duas vezes para chamar a comparação do número 24 e assim por diante, e sempre dobramos o número de recursão a cada nível ou seja, ela está crescendo de forma exponencial.

(b) Escreva uma função good_max: `int list -> int` que corrija o problema da função bad_max dada anteriormente:

Solução:

```
1  fun good_max(xs : int list) =
2      if null xs
3      then 0
4          else if null (tl xs)
5          then hd xs
6          else
7              let val tail_anser = good_max(tl xs)
8              in
9                  if hd xs > tail_anser
10                 then hd xs
11                     else tail_anser
12             end;
```

6. Escreva uma função split: `'a list -> 'a list * 'a list` que receba uma lista de números e divida essa lista em outras duas listas. Se a lista contiver o número ímpar o número restante deve ficar à esquerda.

input: split([1, 2, 3, 8, 4, 5])
output: val it = ([1, 3, 4], [2, 8, 5])

Solução:

```
1  fun split(nil) = (nil, nil)
2      | split([a]) = ([a], nil)
3      | split(a::b::rest) =
4      let
5          val (L, R) = split(rest)
6      in
7          (a::L, b::R)
8      end;
```

7. Dada a função expr: `unit -> int` substitua os valores para a expressão da linha 5, e informe qual o valor que será impresso.

```
1  fun expr () =
2  let
```

```
3   val x = 1
4 in
5   (* x = 2, x + 1 + y = x + 2, y + 1 *)
6 end;
```

Solução:

```
1   fun expr () =
2       let
3           val x = 1
4       in
5           (let val x = 2 in x + 1 end) + (let val y = x+2 in y
                + 1 end)
6       end;
```