Universidade do Porto
Faculdade de Engenharia
FEUP

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Eng.ª de Software

ALGUNS EXERCÍCIOS DE EXAMES ANTERIORES

1. Qual o valor das expressões, em VDM, que se seguem:

a. dom {mk_(1,2).#1 |-> 3, mk_(2,3).#2 |-> 4}

b. [[5,6],[3,1,1],[5]] ++ {2 |-> [5,5],3 |-> [8]}

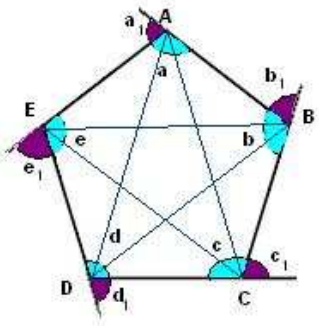
c. {mk_(x,y) | x in set elems ([1,2,2,1] ^ [2]), y in set inds [0,1] & x<=y}

d. {x|->y | x in set dom ({1|->2,2|->3} :> {3}), y in set rng {1|->4} & y = x*2}

e. conc ([[1,2],[2],[3,2]] ++ {1|->[3]})

f. {1|->2,2|->1,4|->4} munion ({1|->1,2|->2} ++ {1|->2,2|->1,3|->1})

2. Polígonos são figuras geométricas planas limitas por linhas poligonais fechadas, por exemplo, um hexágono é um polígono de seis lados. Um polígono é denominado regular se todos os seus lados e todos os seus ângulos forem congruentes.



— **Lados**: Cada um dos segmentos de recta que une vértices consecutivos: $\overline{AB}$, $\overline{BC}$, $\overline{CD}$, $\overline{DE}$, $\overline{EA}$.

— **Vértices**: Ponto de encontro de dois lados consecutivos: A, B, C, D, E.

— **Diagonais**: Segmentos que unem dois vértices não consecutivos: $\overline{AC}$, $\overline{AD}$, $\overline{BD}$, $\overline{BE}$, $\overline{CE}$

— **Ângulos internos**: Ângulos formados por dois lados consecutivos: $\hat{a}$, $\hat{b}$, $\hat{c}$, $\hat{d}$, $\hat{e}$

— **Ângulos externos**: Ângulos formados por um lado e pelo prolongamento do lado a ele consecutivo: $\hat{a}_1, \hat{b}_1, \hat{c}_1, \hat{d}_1, \hat{e}_1$.

Fig. 1: Exemplos de um polígono regular (pentágono)

Considere a seguinte representação de um polígono em VDM++ (Figura 2)

```
class Poligono
types
      public ponto:: x: nat1
                     y: nat1;
  instance variables
      public lados : seq of ponto;
      public angulos: map ponto to nat1;
end Poligono
```

Fig. 2: Representação de um grafo em VDM++

a. Escreva uma função, em VDM++, que determine se um dado polígono tem os ângulos internos todos iguais.

```
public angulosiguais: Poligono ==> Bool
angulosiguais(p) ==
return card (rng p.angulos) = 1;
```

b. Escreva o corpo e pré-condição de uma função (*Remove*), em VDM++, que remova um vértice de um polígono mantendo-o fechado (fig 3). (Nota: Não actualize a informação relativa aos ângulos).

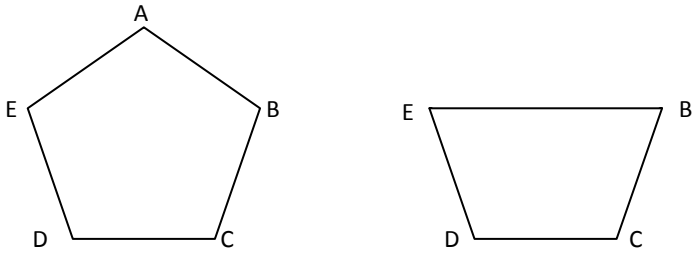


Fig. 3: Exemplo de remoção de um vértice (A) de um pentágono

```
public Remove: Poligono * ponto ==> Poligono
Remove (pol, p) ==
(
  let l1 ^ [p] ^ l2 = lados in
      ( lados := l1^l2;
        angulos := {p} <-: angulos ;
      )
);
```

c. Escreva uma função (*Diagonais*), em VDM++, que retorne o conjunto das diagonais (Fig. 1) de um polígono. Para isso, considere a definição de um tipo adicional (*segrecta*) e complete a especificação em baixo (Fig. 4).

```
class Poligono
  types
    public segrecta :: point
                  point;
    operations
      public Diagonais: () ==> set of segrecta
      // escreva "aqui" o corpo da função
end Poligono
```

Fig. 4: Exemplo de remoção de um vértice (A) de um pentágono

```
public Diagonais () ==> set of segrecta
Diagonais () ==
 { mk_segrecta(a,b) | a in set elems lados, b in set elems lados &
      not exists i in set inds lados & lados(i,...,i+1) = [a,b] and
      lados(i,...,i+1) = [b,a] and
      [lados(1)]^[lados(len lados)] = [x,y] and
      [lados(1)]^[lados(len lados)] = [y,x]
 }
```

3. Uma árvore é um conjunto de segmentos de recta ligados pelos seus extremos e sem ciclos fechados (figs 5 e 6).

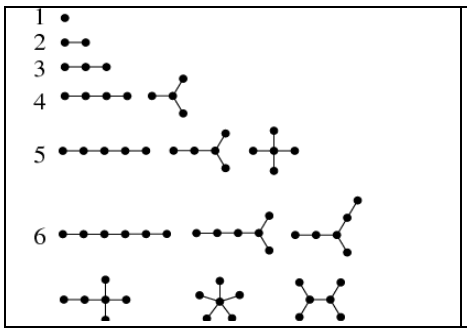


Figura 5: Exemplos de árvores

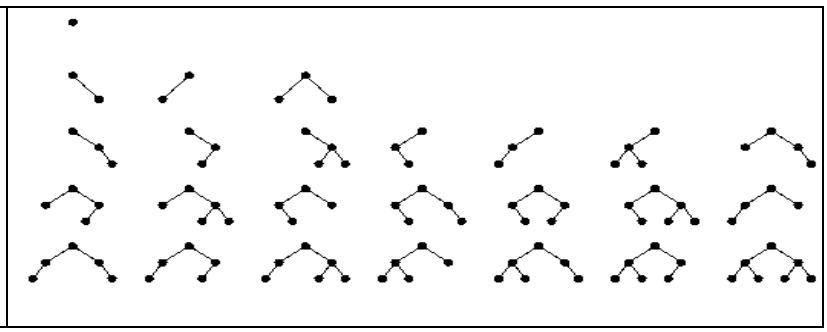
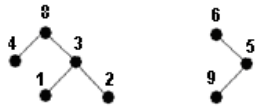


Figura 6: Exemplos de árvores binárias



a) b)

Figura 7: a árvore b) é sub-árvore da árvore a)

```
class Tree
   instance variables
     public root : nat1;
     public edge : map nat1 to nat1;
end Tree
```

Figura 8: Representação de uma árvore (*Tree*) em VDM++

a. Especifique uma função, em VDM++, que retorne o conjunto das folhas de uma árvore (representada em VDM++ pela figura 8).

```
public folhas() ==> set of nat1
folhas() ==
      return (dom edge \ rng edge) union (rng edge \ dom edge)
```

b. Especifique uma função *Insertedge*, em VDM++, que insira uma aresta numa árvore.

c. Se a árvore de input da função *Insertedge* (da alínea anterior) for binária (Figura 6), após executar essa mesma função, a árvore continua binária? Justifique. Em caso negativo, escreva a pré-condição necessária e suficiente para garantir que a árvore resultante da inserção continua binária.

d. Assuma que existe uma função *Distance(f: nat1, a:Tree)* que retorna a distância da folha *f* à raiz da árvore *a*. Especifique uma função, em VDM++, que determine a altura de uma árvore (representada em VDM++ pela fig 8). A altura de uma árvore é a maior distância existente entre uma folha e a raiz (*root*) da árvore.

4. Árvores com raiz balanceadas são grafos conexos sem ciclos (acíclicos) onde as folhas estão a uma distância *d* ou *d+1* da raiz. Uma árvore pode ser representada em VDM++ por:

```
class Tree
types
public edge:: no1: nat1
              no2: nat1;
instance variables
      public Root : nat1:=0;
      public Edges : set of edge := {};

--inv1 Uma árvore com n nós tem n-1 arestas (grafo acíclico).
--inv2 A raíz da árvore (Root) tem que ser um nó da árvore.
--inv3 A árvore tem que ser conexa.
--inv4 A árvore é balanceada (folhas a uma distância d ou d+1 da raíz)
--inv5 entre dois nós arbitrários da árvore existe um único caminho.

end Tree
```

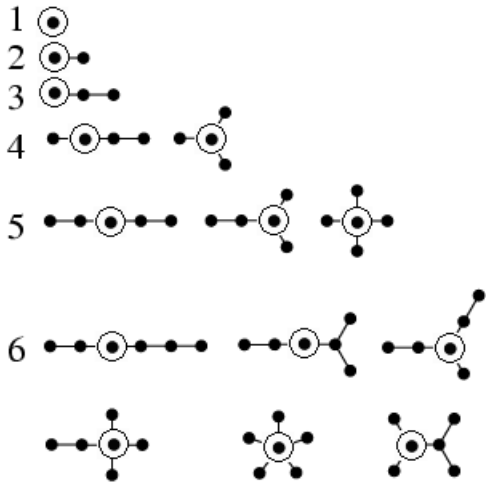


Fig. 9: Árvores balanceadas com 1, 2, ..., 6 nós

a. Formalize os invariantes *inv2* e *inv4*. Considere que existe uma função *dist(nat1)* que calcula a distância de um ponto à raiz da árvore

b. Escreva uma função (*Insert*) que acrescente uma aresta a uma árvore. (nota: considere a situação em que a raiz da árvore poderá ter que ser alterada para que a árvore permaneça balanceada)

```
public Insert: edge ==> Tree
Insert(e) ==.
```

c. Determine se uma dada árvore (representada pela classe *Tree* em VDM++) é uma árvore binária. Numa árvore binária, cada nó tem no máximo dois filhos.

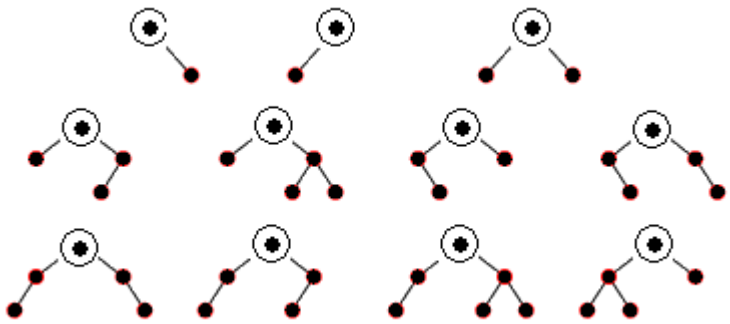


Fig. 10: Exemplos de árvores binárias

d. Escreva uma função (*Mutation*) que transforme uma árvore (representada pela classe *Tree*) no conjunto de todos os caminhos da raiz às folhas (representada pela classe VDM++, *TreePaths*, descrita de seguida).

```
class TreePaths
  type path: seq of nat1;

  instance variables
    public paths: set path;

end TreePaths
```

5. Um grafo dirigido conexo G (Fig. 21a) pode ser representado pelo par (V, A), em que

V – é um conjunto de vértices ou nós e

A – é um conjunto de arestas com um vértice de origem e um conjunto de vértices destino.

Em VDM++, um grafo dirigido conexo pode ser formalizado por:

```
class Graph

instance variables
      public V : set of nat1:={};
      public A : map nat1 to set of nat1:= {|->};

--inv1 Uma aresta só pode ligar vértices do grafo a que pertence.

--inv2 Um grafo tem que ser conexo, i.e., não podem existir dois vértices sem
  --caminho de ligação entre eles (independentemente do sentido das arestas).

end Graph
```

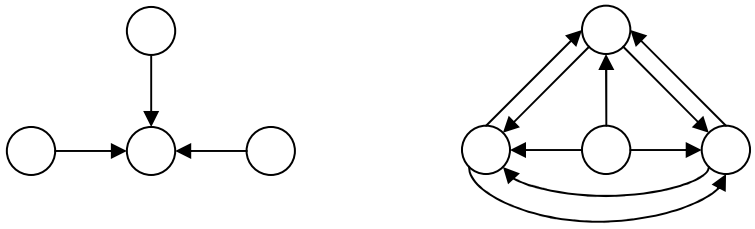


Fig. 21: a) Grafo dirigido conexo. b) Grafo complementar

a. Formalize os invariantes *inv1e inv2*.

```
inv1 not exists e in set dom A union dunion rng A & e not in set V;


inv2 forall e1,e2 in set V & Path(e1,e2);


public Path: nat1 * nat1 ==> bool
Path(n1,n2) ==
  return exists s: seq of nat1 &
s(1) = n1 and s(len s)=n2 and
      forall i,j in set inds s & j=i+1 and
 ( ( s(i) in set dom A and s(j) in set A(s(i)) ) or
         ( s(j) in set dom A and s(i) in set A(s(j)) ) );
```

b. Formalize a função *CompGraph* que determina o grafo H (Fig. 21b) complementar de um grafo G. Um grafo H complementar de G tem o mesmo conjunto de vértices de G e o conjunto de arestas não presentes em G. Formalize também a pós-condição desta função da forma mais abstracta possível.

```
      public CompGraph: () ==> Graph
      CompGraph() ==
      (  dcl cg : Graph := new Graph();
         cg.V := V;
         cg.A := {x |-> y | x: nat1, y : set of nat1 &
            x in set V and y subset V and
            (x not in set dom A and y = V\{x}) or
                  (x in set dom A and y = V\{x}\A(x)) } ;
         return cg;
      )
      // outra solução
      CompGraph() ==
      (
            dcl g:Graph := new Graph();
            g.V := V;
            g.A := {x|->y | x in set V, y = V \ {x} \ {A(x)}
            return g;
      )
      post RESULT.V = V and
        forall x in set V &
          if x in set dom self.A then (
             (A(x) = V\{x} and x not in set dom RESULT.A) or
             (A(x) union RESULT.A(x) = V\{x} and A(x) inter RESULT.A(x) = {})
```

```
                                        )
        else RESULT.A(x) = V\{x};
```

c. Usando a função *CompGraph* definida em cima formalize uma função *SubGraph* que determina se um grafo H é subgrafo de G.

```
    public SubGraph: Graph ==> bool
    SubGraph(g) ==
    (dcl cg : Graph := CompGraph();
     return g.V subset self.V and
     not exists x in set dom g.A &
        x in set dom cg.A => g.A(x) inter cg.A(x) <> {}  ;
    );
    // outra solução sem usar o grafo complementar
    SubGraph(g) ==
        g.V subset V &
        forall x in set dom g.A & g.A(x) subset A(x);
```

d. Formalize uma função *TransitiveClosure* que determina o conjunto dos vértices que é possível atingir (com *n* passos) a partir de um dado vértice inicial (considere o sentido das arestas).

```
    public TransitiveClosure: nat1 * nat1 ==> set of nat1
    TransitiveClosure(v,n) ==
      return {e | e in set V & exists s: seq of nat1 &
  s(1) = v and s(len s) = e and len s <= n and
      forall x,y in set inds s & y = x+1 and
        s(x) in set dom A and y in set A(s(x))}
      pre v in set V;
```

e. Considere a especificação de uma função InsertEdge que adiciona uma aresta a um grafo dirigido:

```
public InsertEdge: nat1 * nat1 ==> Graph
InsertEdge(n1, n2) ==
      (
        dcl V_old: set of nat1 := V;
        V:= V union {n1,n2};
        A := A ++ if n1 in set V_old then {n1 |-> A(n1) union {n2}}
                  else {n1 |-> {n2}};
        return self;
      )
```

f. "A especificação apresentada não é consistente". Comente a afirmação.

```
Não é consistente porque não garante que o grafo final seja conexo. Com esta
função é possível adicionar uma nova aresta a ligar dois vértices não existentes
anteriormente no grafo o que resulta num grafo não conexo.
```

g. Formalize uma pré-condição para a função *InsertEdge* que garanta a consistência do modelo.

```
pre n1 in set V or n2 in set V;
```

6. Um "cut-vertice" de um grafo conexo é um nó do grafo cuja remoção torna o grafo desconexo. Um grafo sem "cut-vertices" é um grafo bi-conexo (Figura 12).

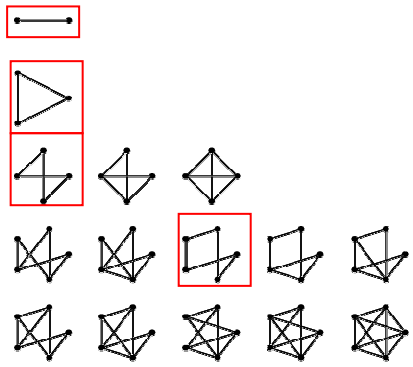


Figura 12: Exemplos de grafos bi-conexos

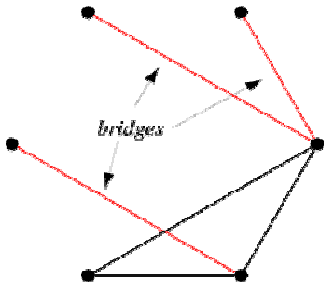


Figura 13: Exemplo de grafo com pontes (*bridges*)

```
class Graph
   instance variables
      public edge : map nat1 to nat1;
end Graph
```

Figura 14: Representação de um grafo em VDM++

a. Especifique uma função, em VDM++, que retorne o nº de vértices de um grafo (especificado pela classe *Graph* na fig. 14) é bi-conexo.

b. Especifique o corpo da função *removearesta*, em VDM++, que remova uma aresta de um grafo conexo.

c. Uma **ponte** (*bridge*) de um grafo conexo é uma aresta (*edge*) do grafo cuja remoção torna o grafo não conexo (Figura 13). Especifique uma função pontes, em VDM++, que devolva todas as pontes de um grafo. Escreva também, em VDM++, a pós-condição desta função.

Resolução

```
class Graph
   instance variables
      public edge : map nat1 to nat1;

   operations
      public vertices : map nat1 to nat1 ==> int
      vertices (e) ==
            return card (dom e union rng e);

      public removedot : map nat1 to nat1 * nat1 ==> map nat1 to nat1
      removedot(e,p) == return ({p}<-:e):->{p};
      -- outra forma de resolver
      -- return {x|->y | x in set dom e, y in set rng e & x<>p and y<>p
      --                 and y=e(x)};

      --Resolução de conexão (não executável)
      public connected : map nat1 to nat1 ==> bool
      connected(e) == return
        forall x,y in set (dom e union rng e) &
          exists s: seq of nat1 &
          (s(1)=x and s(len s)=y and
            (forall i,j in set inds s &
              s(i) in set dom e and e(s(i))=s(j) or
              s(j) in set dom e and e(s(j))=s(i)));

      public biconnected : map nat1 to nat1 ==> bool
      biconnected (e) == return
        connected(e) and
        forall x in set (dom e union rng e) &
          connected(removedot(e,x));

      public removearesta : Graph * nat1 * nat1 ==> map nat1 to nat1
            removearesta (g,y,w) ==
              return {y} <-: g.edge
      pre y in set dom g.edge and g.edge(y) = w;
```

```
      public pontes : Graph ==> map nat1 to nat1
      pontes (g) ==
        return {x|->y | x in set dom g.edge, y in set rng g.edge &
                g.edge(x)=y and not connected(removearesta(g,x,y))}
      pre connected(g.edge)
      post forall x in set dom RESULT &
       not connected(removearesta(g,x,g.edge(x)));

end Graph
```