AULAS TEÓRICO-PRÁTICAS DE COMPILADORES

2º semestre de 2002/2003

AULA Nº 3 (3 horas)

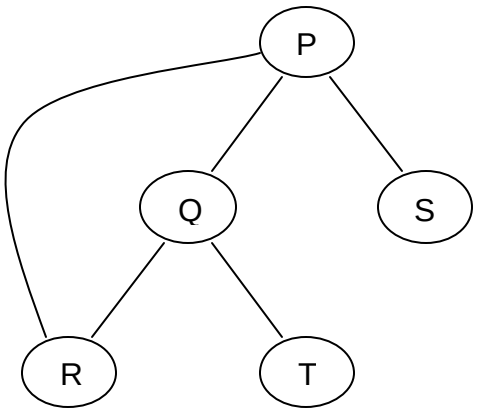
1 Algoritmo Iterativo para Determinar Funções Recursivas

Esta aula pretende representar uma classe de algoritmos muito utilizada em algumas fases de um compilador. Estes algoritmos designam-se por iterativos, pois vão alcançando a solução final por iteracções. O problema apresentado tem a ver com a determinação se um conjunto de chamadas a funções existentes num dado programa tem chamadas que possam definir recursividade.

Para determinar se um programa tem recursividade é normalmente criado um grafo chamado de *Call Graph*[1] (grafo de chamadas) que representa a estrutura de chamadas do programa. Vamos supor por exemplo os pedaços de código seguintes (as reticências indicam enunciados no programa que não contêm chamadas a instruções):

```
Void P() {... Q(); … S(); …}
Void Q() {...R(); … T(); …}
Void R() {…P(); …}
Void S() {…}
Void T() {...}
```

Cada nó do *Call Graph* representa a chamada a uma função ou procedimento, e cada laço entre dois nós representa que a função de onde sai a seta chama a função para onde a seta aponta. O *Call Graph* que representa o programa é (podería ser construído com base na análise do texto que define o programa):



Vamos supor que o compilador pretendia determinar quais são as funções recursivas.

1.1 Transitive Closure

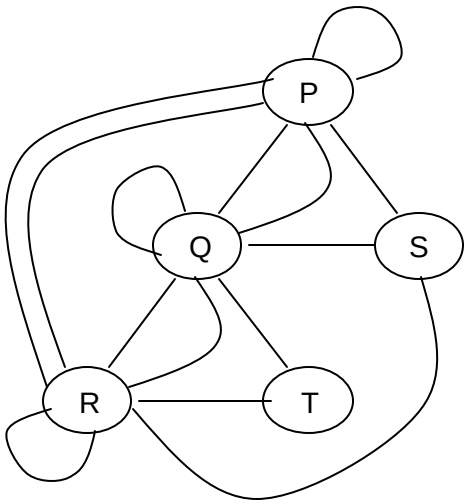
Uma das regras que se pode aplicar ao grafo é a da transitividade:

[1] O *Call Graph* é utilizado na compilação em outros contextos que não apenas o de recursividade.

Se existe uma seta do nó A para o nó B e do nó B para o nó C então vamos fazer com que exista também uma seta entre o nó A e o nó C.

Como os enunciados 'A chama A directa ou indirectamente' e 'A é recursiva' são equivalentes a aplicação da regra anterior permite determinar automaticamente as funções recursivas num determinado programa.

Após aplicação da regra da transitividade teríamos o grafo seguinte:



Que mostra claramente que as funções P, Q, e R são funções recursivas.

1.2 Algoritmo Iterativo para Transitive Closure

O seguinte algoritmo iterativo pode ser utilizado para aplicar a propriedade transitiva a um *Call Graph*:

```
SET flag of Something changed TO TRUE;
WHILE something changed {
     SET flag of Something changed TO FALSE;
     FOR EACH node A IN Graph {
         FOR EACH node B IN Descendants of Node A {
             FOR EACH node C IN Descendants of Node C {
                 IF there is no arrow from Node A to Node C {
                    Add an arrow from Node A to Node C;
                     SET flag Something changed TO TRUE;
                 }
             }
         }
     }
}
```

1.3 Exercício 1

Com base no código Java (ver a secção seguinte) formado pelas classes **Graph** e **TransitiveClosure** , implemente o método (nome designado a funções que pertencem a uma classe) da classe **Graph** designado por **transitiveClosure** que deve implementar o algoritmo apresentado anteriormente.

a) Compile as classes java fazendo: **javac *.java**

b) Execute o programa fazendo: **java TransitiveClosure**

1.4 Código Java

No código Java o *Call Graph* original é representado por um array, designado por Edges, de 5×5 elementos booleanos. O grafo original poderá ser representado pela seguinte matriz, em que os nós 0, 1, 2, 3, e 4 representam as funções P, Q, R, S, e T, respectivamente:

Edges[i][j]		J				
		0	1	2	3	4
I	0	False	**true**	false	**true**	false
	1	False	False	**true**	False	**true**
	2	**true**	False	False	False	False
	3	False	False	False	False	False
	4	False	False	False	False	False

Por exemplo, o valor **true** no elemento edges[0][1] indica que existe um laço entre o nó 0 e o nó 1 com sentido de 0 para 1.

1.4.1 Classe Graph.java

```
/**
   This is the class of the calling graph structure.
   It includes the transitive closure algorithm.

    the lines below tells the javadoc tool who is the author of this class and which is the
version (try: javadoc graph.java and then view the generated html files)

  @author João M. P. Cardoso
  @version v0.1
*/

// import from the API the class to use System.out.println
import java.io.*;

public class Graph {

   // array which represents the edges between nodes
   boolean[][] edges;

   /**
    add an edge between two nodes

    @param source represents the source of the directed edge
    @param sink represents the destination node of the directed edges
   */
   public void addEdge(int source, int sink) {
      edges[source][sink] = true;
   }
```

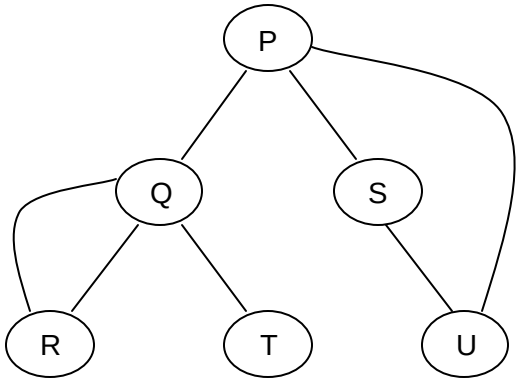
```
    /**
     Print the edges between nodes.
     */
    public void print() {
        System.out.println("Graph edges: ");

        int N = edges.length;

        for(int i=0; i<N; i++)
            for(int j=0; j<N; j++)
                if(edges[i][j])
                    System.out.println(i+" -> "+j);
    }

    /**
     Constructor of the graph.
     The line below defines the argument numNodes to be used by the javadoc tool
     @param numNodes the number of nodes in the graph
    */
    public Graph(int numNodes) {
        // create the array with size = numNodes x numNodes
        edges = new boolean[numNodes][numNodes];

        // initiate all the elements equal to false: none edges
        for(int i=0; i<numNodes; i++)
            for(int j=0; j<numNodes; j++)
                edges[i][j] = false;
    }

    /**
      Show the functions ofthe calling graph which are recursive.
    */
    public void printRecursiveFunctions() {
        int numNodes = this.edges.length;

        System.out.print("Recursive Functions in nodes: {");
        for(int k=0; k<numNodes; k++)
            // if there is an edge with source=sink then
            // the node represents a recursive function
            if(edges[k][k])
                System.out.print(" "+k);

        System.out.println(" }");
    }

    /**
     Algorithm to perform the transitive closure on the original graph.
    */
    public void transitiveClosure() {

        // your code must be here!!!

    }

}
```

1.4.2 Classe TransitiveClosure.java

```
/**
  Class where the calling graph is created and the transitive closure is applied.
*/
public class TransitiveClosure {

   public static void main(String args[]) {
        // create a calling graph with 5 nodes
        Graph callGraph = new Graph(5);

        // add the 5 edges
        callGraph.addEdge(0, 1);
        callGraph.addEdge(1, 2);
        callGraph.addEdge(2, 0);
        callGraph.addEdge(0, 3);
        callGraph.addEdge(1, 4);

        // print the original graph
        callGraph.print();

        // call the transitive closure algorithm over the callGraph
        callGraph.transitiveClosure();

        // print the graph after applying the transitive closure algorithm
        callGraph.print();

        // show the nodes which represent recursive functions
        callGraph.printRecursiveFunctions();
    }
}
```

2 Trabalho para Casa

Modifique o método `main` da classe `TransitiveClosure` de modo a determinar as funções recursivas do seguinte *Call Graph*:



3 Créditos

A aula é baseada no exemplo do *Call Graph* apresentado no livro:

Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen, *Modern Compiler Design*, John Wiley & Sons, Ltd, 2000.