Linguagens e Ambientes de Programação

Exame de Época Normal 2008/2009 – Proposta de Resolução

1. **E** Em OCaml, os acessos a variáveis intermédias são verificados estaticamente.

   M O compilador determina o *offset* dentro do registo de activação onde a variável reside, mas o sítio exacto pode encontrar-se mais abaixo ou mais acima na pilha e só pode ser verificado dinamicamente.

   E O compilador detecta o erro ainda antes de executar o programa.

   D No caso geral, este tipo de erros ocorre em tempo de execução.

   D A generalidade das linguagens de *scripting* dispõe de sistemas de tipos dinâmicos.

   M O sistema de tipos em Java é essencialmente estático, mas também possui uma componente dinâmica que engloba a operação `instanceof`.
   A resposta **E** também seria aceite.

   T O JavaScript, por exemplo, é implementado através de um interpretador, mas para o qual também existem compiladores.

   D Só se sabe qual das variáveis está a ser acedida em tempo de execução, e como tal a atribuição só pode ser validada dinamicamente.

   E Ao usar o Eclipse com o *plugin* para JavaScript podemos facilmente observar que este tipo de erro é detectado ainda antes de correr o programa.

   D Estas conversões são efectuadas em tempo de execução.

2.

1. A função `f` recebe dois argumentos.

```
f : __ -> __ -> __
```

2. Ambos os argumentos são funções:
   - A função `a` recebe `b` como argumento;
   - A função `b` recebe o resultado de `a` sobre `b`.

```
f : (__ -> __) -> (__ -> __) -> __
```

3. Admitindo que os primeiros parêntesis representam a função `a` e os segundos a função `b`.
   - A função `a` recebe como argumento a função `b`, pelo que se completa primeiro a função `b`;
   - A função `b` recebe o resultado de `a` e actua sobre o mesmo, i.e. recebe `'a` e devolve `'b`.

```
f : ((__ -> __) -> __) -> ('a -> 'b) -> __
```

4. Como já foi dito anteriormente, a função `a` recebe `b` como argumento e devolve `'a`, para que a função `b` receba `'a` como argumento.

```
f : (('a -> 'b) -> 'a) -> ('a -> 'b) -> __
```

5. Para terminar, é necessário inferir o resultado da função `f`, que será o resultado de `b`, neste caso `'b`.

Assim sendo:

```
f : (('a -> 'b) -> 'a) -> ('a -> 'b) -> 'b
```

3.

a)
```
let rec sufix s r =
    match r with
        | [] -> []
        | (st,t)::xs ->
            if st = s then (st,t)::xs
            else sufix s xs
;;
```

b)
```
let rec prefix s r =
    match r with
        | [] -> []
        | (st,t)::xs ->
            if st = s then [(st,t)]
            else let res = prefix s xs in
                if res = [] then []
                else (st,t)::res
;;
```

Solução alternativa:

```
let rec prefix s r =
    List.rev (sufix s (List.rev r))
;;
```

c)
```
let rec direct d a t =
    match t with
        | [] -> []
        | x::xs ->
            let currPath =
                   prefix d (sufix a xs) in
                let altPath = direct d a xs in
                    if currPath = [] then altPath
                    else if altPath = [] then currPath
                    else if List.length currPath <
                              List.length altPath then currPath
                    else altPath
;;
```

d) Para descobrir a melhor solução, geram-se e exploram-se em paralelo todos os melhores caminhos possíveis. Quando se muda de comboio e se continua a viagem no novo comboio, alguns dos caminhos alternativos crescem; outros são removidos porque não têm continuação.

Funções auxiliares:

```
let rec flatMap f l =
    List.flatten (List.map f l)
;;

let rec last l =
    List.hd (List.rev l)
;;
```

A função `next` calcula todos os melhores caminhos a partir de `a` até todas as estações de destino possíveis, sem mudar de comboio. (Poderíamos ainda eliminar as repetições no resultado para tornar a solução mais eficiente.)

```
let next a ll =
    let all = flatMap (fun l -> List.map fst l) ll in
        let routes =
          List.map (fun s -> direct a s ll) all in
            List.filter (fun l -> List.length l > 1) routes
;;
```

A função `extend` faz crescer o caminho `r` com todas as "continuações de caminho alternativas" que estão em `alt`. Assume-se que o último elemento de `r` aparece no início de todas as alternativas em `alt`.

```
let extend r alt =
    List.map (fun l -> r @ List.tl l) alt
;;
```

A função `bestAux` implementa um algoritmo com lógica imperativa:

- Todos os percursos possíveis sem mudar de comboio;
- Todos os percursos possíveis alternativos, mudando de comboio apenas uma vez;
- Todos os percursos possíveis alternativos, mudando duas vezes de comboio.

E assim sucessivamente...

Existem três razões que podem parar a execução da função:

- Foi descoberto um percurso para a estação `b`;
- Deixou de haver percursos alternativos;
- Todos os percursos alternativos têm tamanho superior a 50.

```
let rec bestAux a b ll alt =
    let d = direct a b alt in
        if d <> [] then d
        else if alt = [] then []
        else if List.for_all
                    (fun l -> List.length l > 50) alt then []
        else bestAux a b ll (flatMap (fun l -> extend l
                                (next (fst (last l)) ll)) alt)
;;
```

A função `best` limita-se a chamar a função auxiliar `bestAux`, passando no 4º argumento `next a ll`.

```
let best a b ll =
    bestAux a b ll (next a ll)
;;
```

4.

25:	PC	?	-- Y --
	SL	18	
	DL	18	
	j	19	
	i	0	-- X --
20:	PC	?	
	SL	00	
	DL	14	
	a	9	
	PC	?	-- Y --
15:	SL	09	
	DL	09	
	j	9 -> 10	
	i	0	-- X --
	PC	?	
10:	SL	00	
	DL	05	
	a	0	
	PC	?	-- main --
	SL	00	
05:	DL	00	
	j	9 -> 10 -> 11	-- start --
	i	8	
	PC	?	
	SL	?	
00:	DL	?	

5.

a) Imprime o dígito 0 dez vezes e termina

b) Nenhuma das anteriores – Imprime uma sequência infinita de 5

c) Imprime os dígitos de 0 a 9 repetidamente e não termina

6. ***.h**

```
#ifndef _Image_
#define _Image_

#include "Point.h"

class Image {
    public:
        Image();
        virtual ~Image();
        virtual bool Belongs(const Point& p) const;
};

class BasicImage : public Image {
    public:
        BasicImage();
        virtual ~BasicImage();
};

class ComposedImage : public Image {
    protected:
        Image *a, *b;
    public:
        ComposedImage(Image *a, Image *b);
        virtual ~ComposedImage();
};

class Circle : public BasicImage {
    private:
        Point center;
        double radius;
    public:
        Circle(const Point& center, double radius);
        virtual ~Circle();
        bool Belongs(const Point& p) const;
};

class Rectangle : public BasicImage {
    private:
        Point tl, br;
    public:
        Rectangle(const Point& tl, const Point& br);
        virtual ~Rectangle();
        bool Belongs(const Point& p) const;
};

class Square : public Rectangle {
    public:
        Square(const Point& tl, double side);
        virtual ~Square();
};

class Union : public ComposedImage {
    public:
        Union(Image *a, Image *b);
        virtual ~Union();
        bool Belongs(const Point& p) const;
};
```

```
class Difference : public ComposedImage {
    public:
        Difference(Image *a, Image *b);
        virtual ~Difference();
        bool Belongs(const Point& p) const;
};

#endif
```

***.cpp**

```
#include <iostream>
#include "Point.h"
#include "Image.h"

Image::Image() {}
Image::~Image() {}

BasicImage::BasicImage() {}
BasicImage::~BasicImage() {}

ComposedImage::ComposedImage(Image *a, Image *b) : a(a), b(b) {}
ComposedImage::~ComposedImage() {}

Circle::Circle(const Point& center, double radius) :
    center(center), radius(radius) {}
Circle::~Circle() {}

bool Circle::Belongs(const Point& p) const {
    return center.dist(p) <= radius;
}

Rectangle::Rectangle(const Point& tl, const Point& br) :
    tl(tl), br(br) {}
Rectangle::~Rectangle() {}

bool Rectangle::Belongs(const Point& p) const {
    return (tl.x() <= p.x()) && (p.x() <= br.x()) &&
                (tl.y() <= p.y()) && (p.y() <= br.y());
}

Square::Square(const Point& tl, double side) :
    Rectangle(tl, Point(tl.x() + side, tl.y() - side)) {}
Square::~Square() {}

Union::Union(Image *a, Image *b) : ComposedImage(a, b) {}
Union::~Union() {}

bool Union::Belongs(const Point& p) const {
   return a->Belongs(p) || b->Belongs(p);
}

Difference::Difference(Image *a, Image *b) : ComposedImage(a, b) {}
Difference::~Difference() {}

bool Difference::Belongs(const Point& p) const {
    return a->Belongs(p) && !b->Belongs(p);
}
```