Linguagens e Ambientes de Programação (2018/2019)

Prática 1

1ªhora: Primeiro contacto com a linguagem de programação OCaml e o seu ambiente de desenvolvimento na consola.
2ªhora: Exprimir ou aproximar os mecanismos disponíveis numa dada linguagem de programação numa outra linguagem que os não possua como primitivos.
Exercícios de 1 a 10.

Ambiente de referência - Linux

Nas aulas práticas o nosso ambiente de referência, tanto para trabalhar, como para os projetos, será o Linux instalado nos laboratórios. Os projetos também serão corrigidos e avaliados no Linux dos laboratórios.

Primeiro contacto com a linguagem de programação OCaml e o seu ambiente de desenvolvimento na consola

A linguagem de programação OCaml é um dialeto da linguagem de programação ML.

Na 1ª parte da cadeira LAP, vamos explorar a vertente funcional da linguagem OCaml. Em OCaml, um programa é essencialmente uma sequência de definições de constantes e de definições de funções. Constantes e funções são conceitos já conhecidos da matemática. Note que numa linguagem funcional pura **não estão disponíveis, nem variáveis mutáveis nem ciclos**.

Um programa OCaml pode ser executado de duas formas distintas: (1) interpretado; ou (2) compilado e executado autonomamente. Um **interpretador** é um programa que lê o texto de um programa e o executa diretamente. Um **compilador** transforma o texto de um programa numa linguagem mais simples para mais tarde ser executado numa máquina virtual ou diretamente no hardware.

Nestes primeiros exercícios vamos fazer experiências com as duas vertentes de execução de um programa OCaml.

- 1 - Numa consola Linux, lance o **interpretador** da linguagem OCaml usando o o comando `ocaml`. O interpretador aceita três tipos de entidades:
  - Expressões para avaliar (e.g. `1+2;;`) - Neste caso, cada expressão é avaliada, sendo produzido e apresentado o seu valor.
  - Definições (e.g. `let x = 1;;`) - Servem para dar um nome a um valor. O valor pode ser, por exemplo, um inteiro ou uma função.
  - Diretivas para executar (e.g. `#quit;;`) - São comandos que permitem controlar a sessão de trabalho.

Note que todas as entradas do interpretador são finalizadas com um ponto e vírgula duplo.

Experimente introduzir o seguinte e observe e interpete os resultados produzidos:

```
# 1+2;;
- : int = 3
# let x = 1;;
val x : int = 1
# let y = x*2+1;;
val y : int = 3
# #quit;;
```

Geralmente, é dado um valor como resposta, sendo ainda indicado o seu tipo e o identificador associado no caso de ser uma definição.

Nota: Em vez da diretiva `#quit;;`, pode inserir `CTRL-D` no Linux ou `CTRL-Z` no Windows.

- 2 - Para além da produção de valores, as expressões OCaml também podem originar ações de input/output. A escrita de valores no terminal é conseguida através da invocações de funções da biblioteca padrão do OCaml. Repare que existe um "print" diferente para cada tipo de valores. (Para já não se preocupe com o tipo "unit", mas pode-se desde já dizer que é usado em muitas das situações em que o tipo "void" é usado em Java).

Experimente isto:

```
# print_string "Hello World!";;
Hello World!- : unit = ()
# print_int (10+3);;
13- : unit = ()
# print_char 'c';;
c- : unit = ()
# print_float 1.0;;
1.- : unit = ()
```

- 3 - Neste exercício vamos abandonar temporariamente o interpretador de OCaml e usar o **compilador** de OCaml.

Usando um editor de texto, crie um ficheiro de texto chamado `hello.ml`, escreva o miniprograma

```
print_string "Hello World!\n";;
```

e depois compile-o na linha de comando assim:

```
$ ocamlc hello.ml -o hello
```

Este comando produz um ficheiro executável chamado `hello` na diretoria corrente. Execute-o assim:

```
$ ./hello
Hello World!
```

Note que o interpretador escreve automaticamente o valor das expressões avaliadas na consola, mas o compilador não faz isso. Num programa que se destine a ser compilado, temos de ser nós a mandar escrever tudo usando as operações de input/output exemplificadas no exercício anterior.

- 4 - Neste exercício regressamos ao interpretador, mas agora o código a executar provirá dum ficheiro e não da consola. Carregue o ficheiro "hello.ml" no interpretador através da diretiva "use":

```
# #use "hello.ml";;
Hello World!
- : unit = ()
```

Observe o resultado obtido e tire conclusões.

- 5 - Em OCaml há duas maneiras de definir funções: com nome e anónimas. A definição de funções com nome e a sua invocação seguem a sintaxe que se exemplifica:

```
# let f x = x+1;;
 val f : int -> int = <fun>
# f 2 ;;
- : int = 3
```

Note que não se escrevem os tipos das entidades e que o interpretador de OCaml infere o seguinte tipo para a função: `int -> int`.

A seguinte função anónima

```
# fun x -> x+1;;
- : int -> int = <fun>
```

tem o mesmo tipo, mas não fica identificada por nenhum nome. É possível usar diretamente funções anónimas em expressões, como no exemplo:

```
# (fun x-> x+1) 2;;
- : int = 3
```

A definição da função `f`, também pode ser escrita assim:

```
# let f = (fun x -> x+1);;
```

- 6 - As definições em OCaml podem ser recursivas, sendo para isso necessário indicar a palavra reservada `rec`. Reconhece esta função?

```
# let rec fact x = if x = 0 then 1 else x * fact (x-1) ;;
val fact : int -> int =
# fact 5 ;;
- : int = 120
```

- 7 - Experimente ainda as seguintes expressões para conhecer outras características da linguagem OCaml. Com a ajuda do professor, interprete os resultados que obtiver. Tudo isto será objeto de discussão nas aulas teóricas.

```
# let f x = x + x ;;

# let f x = x +. x ;;

# let f x = x ^ x ;;

# let f (x,y) = x + y ;;

# let f (x,y) = x +. y ;;

# let f (x,y) = (y, x) ;;

# let f (x,y) = x = y ;;

# let g x y = x + y ;;
```

Ambiente de trabalho baseado na consola e no interpretador

Na primeira parte da aula de hoje, trabalhámos da forma que se descreve a seguir. Na próxima aula começaremos a usar o ambiente de trabalho Eclipse.

Manter três janelas abertas, no computador:

- Uma com um browser ativo, para terem acesso à página de LAP, onde se encontra o enunciado dos exercícios;
- Outra com um editor de texto para escreverem os programas;
- Uma terceira com uma consola, para correrem o interpretador OCaml.

Para facilitar a criação destas três janelas no início de cada sessão, arraste para a barra que se situa no topo do ecrã os ícones das três aplicações relevantes: "Browser", "Editor de texto" e "Terminal".

O processo de desenvolvimento de programas OCaml é o seguinte:

- Escreve-se o programa num ficheiro de texto com a ajuda do editor de texto e grava-se;
- Carrega-se o programa no OCaml, usando a diretiva "#use";
- Testa-se o programa.
- Se houver erros:
  - volta-se a editar, a gravar, a carregar e a testar

Exprimir ou aproximar os mecanismos disponíveis numa dada linguagem de programação numa outra linguagem que os não possua como primitivos.

Por vezes, quando estamos a trabalhar com uma linguagem L1, intessa-nos imitar um mecanismo de programação conhecido da linguagem L2 mas não disponível na linguagem L1. Exemplos: imitar as exceções do Java numa linguagem sem exceções; imitar os objetos do Java numa linguagem sem objetos; imitar o mecanismos de execução do Prolog numa linguagem sem esses mecanismos, etc.

Este tipo de exercícios são muito instrutivos pois ajudam a entender melhor as linguagens envolvidas e dos seus mecanismos. O esforço, nada trivial, de resolver este tipo de exercícios, obriga a perceber completamente o mecanismo da linguagem L2 que se quer imitar, assim como os mecanismos da linguagem L1 que são usados na imitação.

Segue-se alguns exercícios deste género.

- 8 - Dada a ausência de variáveis imperativas e de ciclos na parte funcional da linguagem OCaml, proponha uma função recursiva em OCaml para implementar uma função equivalente ao seguinte método Java. Aqui a ideia é mesmo simular de alguma forma variáveis imperativas em OCaml.

A função calcula o máximo divisor comum entre dois números inteiros positivos.

```
int mdc(int m, int n) {
  int x;
  while( n != 0 ) {
    x = n; n = m % n; m = x ;
  }
  return m;
}
```

Nota: Neste exercício, o ponto de partida foi um método escrito usando os mecanismos imperativos do Java. De forma natural, somos levados a simular esses mecanismos em OCaml. Contudo, como veremos já a partir da próxima semana, quando se resolve um problema em OCaml, não há necessidade nem interesse, sendo até prejudicial, pensar em termos de variáveis imperativas e de ciclos. Este é um exercício académico de "imitação de outra linguage" e constitui uma exceção.

- 9 - Considere o seguinte programa em Java:

```
abstract class Shape {
    abstract double area();
}
```

```
class Line extends Shape {
    private int x0, y0;
    private int x1, y1;
    double area() {
        return 0;
    }
}

class Circle extends Shape {
    private int x, y;
    private double radius;
    double area() {
        return 3.14159 * radius * radius;
    }
}

class Rect extends Shape {
    private int x0, y0;
    private  int x1, y1;
    double area() {
        return Math.abs((x0 - x1) * (y0 - y1));
    }
}

class Canvas {
    private java.util.Vector<Shape> shapes;
    double area() {
        double sum = 0;
        for(Shape s : shapes)
            sum += s.area();
        return sum;
    }
}
```

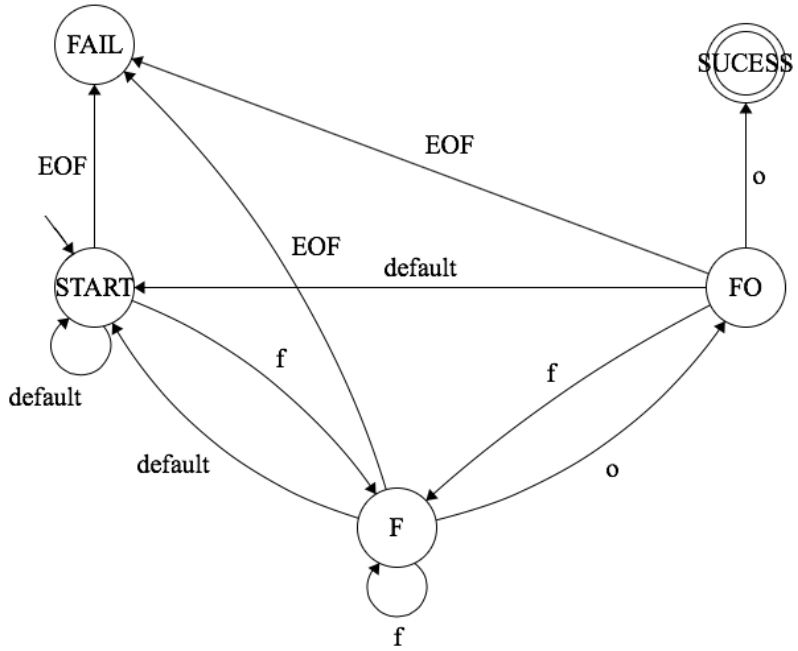
Como você sabe, uma variável de tipo Shape consegue referir objetos de tipo Line, Circle e Rect. Ao aplicar a operação "area" a essa variável, obtém-se a área da figura geométrica em causa.

Agora, imagine que eram retirados do Java os mecanismos de herança e subtipo, que aqui são usados. Como é que reescreveria o programa anterior, tentando capturar o conceito da classe "Shape" (que representa uma forma geométrica geral) e, já agora, sem alterar a classe cliente Canvas? [Sugestão: a melhor solução mantém as cinco classes, sendo a classe Shape que precisa duma grande reescrita.]

- 10 - Em Java não existe a instrução **goto** (apesar desta palavra ter ficado reservada nessa linguagem). Em geral o goto não faz falta, e na maioria das situações até seria contraproducente o seu uso por favorecer o aparecimento de programas difíceis de entender [Artigo: Edsger W. Dijkstra, Go To Statement Considered Harmful, 1969].

Mas há uma situação rara em que o uso do **goto** é recomendado, se estiver disponível. Trata-se da programação de autómatos finitos deterministas (máquinas de estados). Num tal autómato, para cada par (estado, input) existe uma única transição para o estado seguinte. Você vai estudar este assunto na disciplica de Teoria da Computação.

O seguinte diagrama representa um autómato finito com 5 estados que verifica se o input contém, algures lá no meio, a string "foo". Partindo do estado "START", navege mentalmente do diagrama para perceber como ele funciona.



O programa mais abaixo, implementa em C o autómato anterior. Olhe para o código e repare que o **goto** permite transitar de estado de forma natural e intuitiva.

O problema a resolver é o seguinte: **Escreva um programa em Java equivalente ao programa em C e teste-o.**

O Java não tem instrução **goto**. Qual será o melhor plano para efetuar a tradução? Será que existe alguma solução que permita manter a estrutura geral do código C que estamos a ver? [A resposta é **sim**! Se fosse **não**, então o goto faria falta em Java.]]

```
#include <stdio.h>

int main() {
    START:  switch( getchar() ) {
                case 'f' : goto F ;
                case EOF : goto FAIL ;
                default  : goto START ;
            }

    F:      switch( getchar() ) {
                case 'o' : goto FO ;
                case 'f' : goto F ;
                case EOF : goto FAIL ;
                default  : goto START ;
             }

    FO:     switch( getchar() ) {
                case 'o' : goto SUCCESS ;
                case 'f' : goto F ;
                case EOF : goto FAIL ;
                default  : goto START ;
            }

    FAIL:   printf("REJECT.\n") ;
            return 0 ;

    SUCCESS: printf("ACCEPT.\n") ;
             return 0 ;
}
```

Para ler carateres, pode usar o seguinte método, que imita a função `getchar` do C:

```
private static int getchar() {
    try {
```

```
            return System.in.read() ;
        } catch(IOException e) {
            return -1 ;
        }
    }
```

- 11 - Imagine que o mecanismo das exceções era retirado da linguagem Java. Reescreva o seguinte programa Java, mantendo o seu comportamento, mas agora sem usar exceções. A solução que encontrar deverá poder ser aplicada a qualquer outro programa Java. Verá que existe uma solução teoricamente correta, mas muito pouco prática de usar: isso prova que o mecanismo das exceções faz falta no Java, pois não há forma simples de o imitar usando os outros mecanismos.

```
import java.util.*;

// Este e' um tipo-objecto:

interface IntStack {
    int Capacity() ;
    int Size() ;
    boolean Empty() ;
    boolean Full() ;
    int Top() ;
    void Push(int v) ;
    void PushN(int... vs) ;
    void Pop() ;
    void PopN(int n) ;
}

// Esta e' uma implementacao do tipo-objecto anterior:

class IntStackClass implements IntStack {
    private final static int defaultCapacity = 100 ;
    private int[] elems ;
    private int top ;

    IntStackClass(int capacity) {
        elems = new int[capacity] ;
        top = 0 ;
    }
    IntStackClass() {
        this(defaultCapacity) ;
    }

    public int Capacity() {
        return elems.length ;
    }
    public int Size() {
        return top ;
    }
    public boolean Empty() {
        return Size() == 0 ;
    }
    public boolean Full() {
        return Size() == Capacity() ;
    }
    public int Top() {
        if( Empty() )
            throw new EmptyStackException() ;
        return elems[top-1] ;
    }
    public void Push(int v) {
        if( Full() )
            throw new StackOverflowError() ;
        elems[top++] = v ;
    }
    public void PushN(int... vs) {
        for(int v : vs)
```

```
                Push(v) ;
        }
        public void Pop() {
            if( Empty() )
                throw new EmptyStackException() ;
            top-- ;
        }
        public void PopN(int n) {
            for( int i = 0 ; i < n ; i++ )
                Pop() ;
        }

      // TESTING
        public static void main(String[] args) {
            System.out.println("Welcome to the IntStackClass class!!") ;
            IntStack s = new IntStackClass() ;
            System.out.println(s.Capacity()) ;
            try {
                s.PushN(1,2,3,4,5) ;
            }
            catch ( StackOverflowError e ) {
                System.out.println("Stack Overflow!") ;
                System.out.println("Bye bye!") ;
                System.exit(1) ;
            }
            catch ( EmptyStackException  e ) {
                System.out.println("Stack Underflow!") ;
                System.out.println("Bye bye!") ;
                System.exit(1) ;
            }
            s.PushN(1,2,3,4,5) ;
            s.PopN(9) ;
            System.out.println(s.Top()) ;
        }

    }
```

Ajuda: Pode usar a seguinte classe enumerada auxiliar. Há uma solução baseada em ideias simples que passa por inserir no código alguns ifs, alguns returns, criar uma nova função main com código fixo, e pouco mais. Mesmo assim é uma solução complicada de usar, na prática.

```
public enum Exc {
  // Enumeration values
    noneExc, stackOverflowExc, emptyStackExc ;
  // Current exception
    private static Exc curr = noneExc ;
  // Static methods
    public static Exc Catch() { Exc x = curr ; curr = noneExc ; return x ; }
    public static void Throw(Exc e) { curr = e ; }
    public static boolean Throwed() { return curr != noneExc ; }
    public static void DefaultHandling() {
        switch( curr ) {
            case noneExc :
                break ;
            case stackOverflowExc:
                System.err.println("StackOverflow Exception") ;
                System.exit(0) ;
            case stackUnderflowExc:
                System.err.println("StackUnderflow Exception") ;
                System.exit(0) ;
            default:
                System.err.println("Unknown Exception") ;
                System.exit(0) ;
        }
    }
}
```

- 12 - Reescreva o método Java abaixo sem usar nenhum ciclo. Use apenas recursividade. Depois de resolver o problema diga se aceitável eliminar os ciclos do Java, deixando apenas a recursividade?

```
int f(int[] array) {
    int sum = 0;
    for(int v : array)
        sum += v;
    return sum;
}
```
