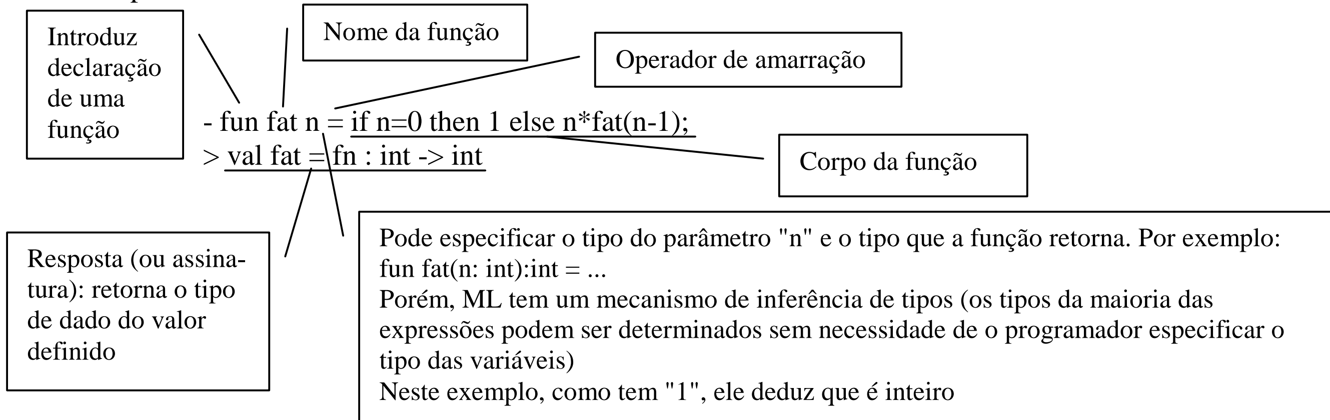
1.1. Estudo de Caso: *MosML*

1.1.1. Introdução

- MoscowML (ou simplesmente MosML) é um subconjunto do ML padrão (ML=MetaLanguage).
- O programa básico é a declaração de uma função.
- Exemplo: Fatorial em ML




```
- fun fat(n:real) : real = if n=0.0 then 1.0 else n*fat(n-1.0);
> val fat = fn : real -> real
- fat (3.0);
> val it = 6.0 : real
- fat(3.2);
! Uncaught exception:
! Out_of_memory
```

- Obs: para escrever uma mensagem na tela digite print "xxx \n"; alguns exemplos:

```
- val str = "teste \n";
> val str = "teste \n" : string
- print str;
teste
> val it = () : unit
- print (str ^ " e mais um \n" ^ str);
teste
 e mais um
teste
> val it = () : unit
- print "Hello World! \n";
Hello World!
> val it = () : unit
```

1.1.2. Funções

- ➥ Funções são tratadas como valores (por isso a resposta "val fat")
- ➥ Depois que a função foi declarada, ela pode ser chamada (ex: - fat 5; > val it = 120 : int)
- ➥ ML possui o equivalente a uma função lambda, isto é, permite definir funções sem nome

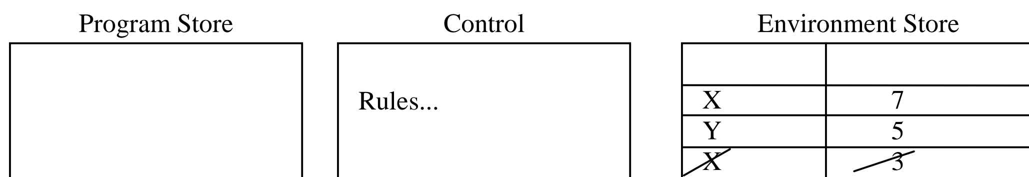
(* com nome *)	- (fn x:int => x * x) 2;
- fun cube(x) = x * x * x;	> val it = 4 : int
> val cube = fn : int -> int	
- cube(3);	
> val it = 27 : int	
(* sem nome *)	
-(fn x => x * x * x) (3);	
> val it = 27 : int	

- ➥ Uma maneira alternativa de definir uma função sem usar a palavra reservada *fun*, é val rec f = fn<match>
- ➥ A palavra reservada *rec*, abreviatura de "recursive",é necessária somente se a função f é recursiva, isto é, ela informa para ML que a função que está sendo definida é recursiva e não foi definida antes.
- ➥ Torna-se importante salientar que declarações que usam *fun* são automaticamente assumidas como recursivas

```
- val rec fat = fn n => if n = 0 then 1 else n * fat (n-1);
> val fat = fn : int -> int
- fat 4;
> val it = 24 : int
- val rec reverse = fn nil => nil
                  | x::xs => reverse(xs) @ [x];
> val reverse = fn : 'a list -> 'a list
- reverse [7,2,5,8];
> val it = [8, 5, 2, 7] : int list
- val F = fn n => if n=0 then 1 else 1+F(n-1);
! Toplevel input:
! val F = fn n => if n=0 then 1 else 1+F(n-1);
!                                    ^
! Unbound value identifier: F
- val F = fn n => if n>0 then "positivo" else "negativo";
> val F = fn : int -> string
- F(10);
> val it = "positivo" : string
- F(~20);
> val it = "negativo" : string
```

1.1.3. Amarração

- ➥ ML permite que nomes sejam amarrados a valor (declaração *value*); é diferente de um comando de atribuição; o nome é amarrado a um valor, mas não pode "trocar" de valor, a menos que seja feita uma nova amarração.



- ➥ Deve-se armazenar quais amarrações são introduzidas (no environment store)
- ➥ Cada célula do environment store contém um par: nome e valor amarrado a ele. Ex: val x = 3;

- ➥ Quanto tiver, por exemplo, "x+3", x é trocado pelo seu valor, que é encontrado no environment store
- ➥ Para cada declaração, uma nova amarração é introduzida no environment store
- ➥ É possível fazer uma nova amarração a um novo valor, porém a amarração antiga nunca mais poderá ser usada
- ➥ Obs1: se colocar, por exemplo, "x=y", ML vai tratar como igualdade e vai retornar um valor booleano (true ou false)
- ➥ Obs2: se usar um nome sem "amarrar", ele dá um erro "unbound value identifier: -"
- ➥ Amarrações temporárias:

```
- val n = 5;
> val n = 5 : int
- fun sum1to n = n * (n+1) div 2; (* n = parâmetro, escopo local *)
> val sum1to = fn : int -> int
- sum1to 100;
> val it = 5050 : int
- sum1to n;
> val it = 15 : int
```

- ➥ Declarações válidas localmente são usadas quando é necessário criar alguns valores temporários dentro da função (como se fossem variáveis locais)
- ➥ Para isto:

```
let
    val <nome1> = <expressão1>;
    val <nome2> = <expressão2>;
    ...
    val <nomeN> = <expressãoN>;
in
    <expressão>                  (* amarrações do let só são válidas aqui *)
end
```

- ➥ Exemplos (";" nos comandos dentro do let é opcional):

- val raio = 10.6;
> val raio = 10.6 : real
- let
val pi = 3.1416
in
pi * raio * raio
end;
> val it = 352.990176 : real
	- let
	val x = 3;
	val y = x * x;
	in
	x * y
	end;
	> val it = 27 : int
		load "math";
		open math;
		- fun area(a, b, c) = let
		val p = (a + b +c) / 2.0;
		in
		sqrt (p * (p-a) * (p-b) * (p-c))
		end;
		> val area = fn : Real.real * Real.real * Real.real -> Real.real

1.1.4. Tipos

- ➥ ML é fortemente tipada (o tipo de qualquer "variável" ou expressão pode ser determinado em tempo de compilação)
- ➥ ML dá erro de tipo e não converte tipos (por exemplo se uma função espera um inteiro e é fornecido um real)
- ➥ Exemplo:

```
- fun doublereal x:real = x * x;
> val doublereal = fn : Real.real -> Real.real
- doublereal 2;
! Toplevel input:
! doublereal 2;
!            ^
```

```
! Type clash: expression of type
!   int
! cannot have type
!   Real.real
- doublereal (real 2);
> val it = 4.0 : Real.real
```

➥ Outro exemplo, a operação de divisão ("/") não funciona com inteiro; neste caso deve-se usar a função "real" que converte de inteiro para real

```
- val resp = 2 / 4;
! Toplevel input:
! val resp = 2 / 4;
!            ^
! Type clash: expression of type
!   int
! cannot have type
!   Real.real
- val resp = real 2 / real 4;
> val resp = 0.5 : Real.real
```

➥ Outras funções de conversão

```
load "Real";
open Real;
(* de real para string *)
- val str = toString(3.1416);
> val str = "3.1416" : string
(* de real para inteiro *)
- val X = trunc(3.14);
> val X = 3 : int
- val Y = round(3.14);
> val Y = 3 : int
- val x = trunc(2.8);
> val x = 2 : int
- val y = round(2.8);
> val y = 3 : int
```

➥ ML permite que alguns tipos em uma expressão sejam "type variables" (tipos variáveis), nos quais o tipo de uma subexpressão pode ser indeterminado e depender do tipo atual fornecido durante a execução. Por exemplo "hd" e "tl" são aplicadas a qualquer tipo de lista (inteiros, reais, caracteres, etc.). "hd" e "tl" não podem ser aplicadas a tuplas

➥ Uma função cujo tipo contém tipos variáveis é POLIMÓRFICA (tipos das variáveis refere-se a tipos fixos mais arbitrários, que são trocados pelos tipos atuais quando as funções são aplicadas)

```
- fun fst (x, y) = x;
> val fst = fn : 'a * 'b -> 'a
- fst(3,6);
> val it = 3 : int
- fst (5.6,7.8);
> val it = 5.6 : real
- fst (9.8, 2);
> val it = 9.8 : real
- fst ("ave", 10);
> val it = "ave" : string
```

➥ ML usa pattern-matching para resolver a questão dos tipos variáveis durante a execução (** escrever só até aqui no quadro **) e para determinar estaticamente os tipos de funções com informação de tipo incompleta nas suas declarações (*type inference* ou inferência de tipo).

1.1.5. Estrutura de dados em ML

1. **LISTA**: os elementos são todos do mesmo tipo; ML fornece uma notação simples para listas que são escritas entre colchetes com vírgulas para separar os elementos

```
- [2,6,1];
> val it = [2, 6, 1] : int list
- [3.4, 5.6, 7.8];
> val it = [3.4, 5.6, 7.8] : real list
- val LC=[#"a", #"g", #"m"];
> val LC = [#"a", #"g", #"m"] : char list
- ["a", "b"];
> val it = ["a", "b"] : string list
- [];
> val it = [] : 'a list
- val L = [7,4,9,2];
> val L = [7, 4, 9, 2] : int list
- hd(L);
> val it = 7 : int
- tl(L);
> val it = [4, 9, 2] : int list
- hd(tl(tl(L)));
> val it = 9 : int
- hd([#"L", #"m", #"A", #"e"]);
> val it = #"L" : char
- (* cons *) 2::L;
> val it = [2, 7, 4, 9, 2] : int list
- (* concatenar listas do mesmo tipo *) [5,6,7]@L;
> val it = [5, 6, 7, 7, 4, 9, 2] : int list
```

Obs: Em ML strings e listas são tipos diferentes; entretanto, existe uma grande similaridade entre um string e uma lista de caracteres e é possível fazer uma conversão entre as duas representações usando as funções built-in *explode* e *implode*. Outra função semelhante a *implode*, *concat*, trabalha com listas de strings e produz um string que é a concatenação de todos os strings de L, em ordem.

```
(* explode pega um string e converte para uma lista de caracteres *)
- explode ("universidade");
> val it = [#"u", #"n", #"i", #"v", #"e", #"r", #"s", #"i", #"d", #"a", #"d", #"e"]: char list
- explode("");
> val it = [] : char list

(* implode pega uma lista cujos elementos sao caracteres e concatena-os formando um string *)
- implode([#"o", #"v", #"o"]);
> val it = "ovo" : string

- (* concat concatena todos os strings de uma lista formando um string final *)
concat(["PUC ", "fica ", "em ", "POA!"]);
> val it = "PUC fica em POA!" : string
```

2. **TUPLA**: é formada por uma lista de duas ou mais expressões de qualquer tipo, separadas por vírgulas e que ficam entre parênteses; é semelhante a registro, mas os campos são identificados pela posição na tupla.

```
- val t = (4, 5.0, "seis");
> val t = (4, 5.0, "seis") : int * Real.real * string
- (1,2,3);
> val it = (1, 2, 3) : int * int * int
- (1, (4, 6.7));
```

```
> val it = (1, (4, 6.7)) : int * (int * Real.real)
- (* "#i" eh usado para acessar um elemento da tupla *)
#1(t);
> val it = 4 : int
- #2(t);
> val it = 5.0 : Real.real
- #3(t);
> val it = "seis" : string
- #4(t);
! Toplevel input:
! #4(t);
!   ^
! Type clash: expression of type
!   int * Real.real * string
! cannot have type
!   {1 : int, 2 : Real.real, 3 : string, 4 : 'a, ...}
! because record label  4  is missing
(* é possível até inserir uma lista de inteiros numa tupla *)
- val T=([4,5,6],(8,"o"),"asddf",#"a");
> val T = ([4, 5, 6], (8, "o"), "asddf", #"a"): int list * (int * string) * string * char
```

3. **Definições TYPE**: é um tipo de macro para expressões definidas previamente, isto é, limita-se a definições de abreviações; *type* define um novo tipo de uma maneira simples, como uma abreviação para outros tipos; a forma mais simples é type <identificador> = <expressão>; <identificador> é o nome para o novo tipo.

```
- type signal = real list;
> type signal = Real.real list
- val v = [1.0, 2.0] : signal;
> val v = [1.0, 2.0] : Real.real list
- type pessoa = string * int;
> type pessoa = string * int
- ("joao", 20);
> val it = ("joao", 20) : string * int
```

O nome dado ao tipo pode ser usado como substituto para o tipo string * int, mas o sistema não usa o nome; mas é possível usar o nome para criar outros tipos.

4. **Definições DATATYPE**: são regras para construção de novos tipos com novos valores (definição de tipos estruturados e enumerados); a definição de um datatype envolve dois tipos de identificadores: I) um construtor que é o nome do datatype; II) um ou mais construtores de dados, que são identificadores usados como operadores para construir valores que pertencem a um novo datatype.

```
- datatype fruta = maca | pera | uva;
> datatype fruta
  con maca = maca : fruta
  con pera = pera : fruta
  con uva = uva : fruta
- fun eh_uva (x) = x = uva;
> val eh_uva = fn : fruta -> bool
- eh_uva(maca);
> val it = false : bool
- eh_uva(uva);
> val it = true : bool
- eh_uva(morango);
! Toplevel input:
! eh_uva(morango);
!        ^^^^^^^
```

```
! Unbound value identifier: morango
(* "of" indica que uma função construtora é introduzida-não é exatamente uma função, o resultado *)
(* não é um valor, apenas um novo objeto de dado é criado *)
(* "con" indica que o contrutor foi definido *)
- datatype ponto = ponto of real * real;
> datatype ponto
  con ponto = fn : Real.real * Real.real -> ponto
- datatype peso = peso of real;
> datatype peso
  con peso = fn : Real.real -> peso
- datatype reg = reg of pessoa * peso * string;
> datatype reg
  con reg = fn : (string * int) * peso * string -> reg
- val joao = reg(("Joao",20), peso 60.5, "Av.Ipiranga 1200");
> val joao = reg(("Joao", 20), peso 60.5, "Av.Ipiranga 1200") : reg
- val maria = reg (("Maria", 30), peso 53.5, "Av. Ganzo 780/402");
> val maria = reg(("Maria", 30), peso 53.5, "Av. Ganzo 780/402") : reg
- datatype arq = file of reg list;
> datatype arq
  con file = fn : reg list -> arq
- val cadcli = file[joao, maria];
> val cadcli =
    file [reg(("Joao", 20), peso 60.5, "Av.Ipiranga 1200"),
          reg(("Maria", 30), peso 53.5, "Av. Ganzo 780/402")]: arq

- datatype direction = norte | sul | leste | oeste;
> datatype direction
  con norte = norte : direction
  con sul = sul : direction
  con leste = leste : direction
  con oeste = oeste : direction
(* ML vê esta declaração como a criação de um padrão que deve ser matched=correspondido *)
(* Pattern-Matching é usado para criar funções do tipo abaixo *)
(* Se não colocarmos todos os casos, ML envia a mensagem "pattern matching is not exhaustive *)
(* ML vai pegar o primeiro pattern que corresponder *)
- fun heading  norte = 0.0
     |heading  sul = 180.0
     |heading leste = 90.0
     |heading oeste = 270.0;
> val heading = fn : direction -> Real.real

- datatype LOGICO = Verdadeiro | Falso | Indefinido;
> datatype LOGICO
  con Verdadeiro = Verdadeiro : LOGICO
  con Falso = Falso : LOGICO
  con Indefinido = Indefinido : LOGICO
- fun AND (Verdadeiro, Verdadeiro) = Verdadeiro
    | AND (Falso,_) = Falso
    | AND (_,Falso) = Falso
    | AND (_,_) = Indefinido;
> val AND = fn : LOGICO * LOGICO -> LOGICO
- AND(Falso,Verdadeiro);
> val it = Falso : LOGICO
- AND(Verdadeiro, Verdadeiro);
> val it = Verdadeiro : LOGICO
```