Material Complementar de linguagem ML

1 Definindo Tipos de Dados Polimórficos em ML

Vamos supor que você necessite construir um tipo de dado polimórfico (que aceite diferentes elementos) e que, além disso, seja recursivo (isto é, aceite ele mesmo como elemento). Um exemplo prático seria uma lista encadeada genérica. Em linguagens imperativas é possível definir estruturas que referenciam elas mesmas e também é possível definir funções que somem os elementos dessa lista de maneira a aceitar qualquer tipo de dado nelas armazenado (estruturas genéricas), com poucas restrições. Mas a complexidade sintática e semântica de se fazer isso depende muito da linguagem em si.

Em ML, é muito simples definir um tipo abstrato polimórfico recursivo e funções genéricas associadas.

Veja o exemplo que define uma lista de Elementos genéricos:

```
datatype 'a list = Nil |
                   Elemento of ('a * ('a list))
```

A definição anterior indica que uma lista de elementos genéricos ('a list) é composta de um elemento nulo (Nil) ou de uma função construtora de elementos (denominada 'Elemento') que recebe uma *tupla* cujo formato é "(<elemento_qualquer> , <lista_genérica>)".

Para criar uma lista e adicionar elementos, faça o seguinte:

```
val lst = Elemento (1, Nil); (* Cria a lista, adicionando o número inteiro 1 *)
val lst = Elemento (2, lst); (* Adiciona 2 na lista*)
val lst = Elemento (3, lst); (* Adiciona 3 na lista*)
```

Apesar de poder ser "instanciada" para qualquer tipo, uma vez criada, os elementos tem de ser compatíveis entre si:

```
val lst = Elemento ("4", lst); (* Erro: tenta adicionar string na lst de inteiros *)
```

Mas a mesma estrutura pode ser utilizada para criar listas de outros tipos:

```
val lst2 = Elemento ("1", Nil);  (* Cria uma lista de strings *)
val lst2 = Elemento ("2", lst2); (* Adiciona "2" na lista*)
val lst2 = Elemento ("3", lst2); (* Adiciona "3" na lista*)
```

Para definir operadores genéricos sobre a lista genérica, é muito simples. Veja o exemplo de uma função que soma os elementos de uma lista:

```
fun soma (Nil) = 0
  | soma (Elemento(valor, proximo)) = valor + soma(proximo);
```

Exemplo de uso (obviamente só funciona para listas de elementos numéricos):

```
soma (Elemento(1, Elemento(2, Elemento(3, Nil))));
```

1.1 Comando abstype

Um comando interessante para definir tipos abstratos (genéricos, no caso) é o 'abstype'.

```
exception PilhaVazia;
abstype 'a Pilha = PILHA of 'a list
with
  val newPilha            = PILHA([])
  fun push (PILHA p) x    = PILHA(x::p)
  fun pop (PILHA [])      = raise PilhaVazia
  |   pop (PILHA (_::xp)) = PILHA(xp)
  fun top (PILHA [])      = raise PilhaVazia
  |   top (PILHA (x::_))  = x
end;
```

Um exemplo de uso:

```
val p = newPilha; (* cria a pilha *)

val p = push p 1; (* o resultado é uma outra pilha com o elemento extra *)
val p = push p 2; (* o resultado é uma outra pilha com o elemento extra *)
val p = push p 3; (* o resultado é uma outra pilha com o elemento extra *)

top p; (* devolve um numero - não modifica a pilha *)

pop p; (* retira um elemento da pilha? *)
pop p; (* retira outro elemento da pilha? *)
top p; (* retirou o elemento? *)

val p = pop p; (* gerou uma nova pilha com um elemento a menos! *)
pop p; (* agora sim! *)
```

2 Definindo Estruturas de Dados em ML (comando 'struct')

Estruturas são declarações nomeadas:

```
structure IntLT = struct
      type t = int
      val lt = (op <)
      val eq = (op =)
end
```

Seus componentes são acessados por referência qualificada: IntLT.lt

Assinaturas (*signatures*) são declarações de tipo:

```
signature ORDERED = sig
      type t
      val lt : t * t -> bool
      val eq : t * t -> bool
end
```

3 Funções e construções úteis com respectivos exemplos de uso

3.1 Geração de números aleatórios (random)

A *Standard ML library* possui várias bibliotecas interessantes. Uma delas é a Random, que oferece recursos para a geração de números aleatórios.

Exemplo de uso da biblioteca Random:

```
load "Random"; (* carrega biblioteca Random *)

val rg = Random.newgen(); (* cria gerador de números aleatórios *)

val n1 = Random.range(0,11)rg; (* gera numero entre 0 e 10 *)

(* abaixo, define função que gera um número qualquer entre 1 e Max-1 *)
fun geranumero max = random.range (1,max) rg;
```

Para uma lista de funções disponíveis na biblioteca Random, utilize o comando:

```
help "Random";
```

ou consulte a ajuda on-line da "*Standard ML Library*" (basta procurar no Google).

3.2 Manipulação de Listas

A biblioteca List tem várias funções interessantes de manipulação de listas. A função nth, por exemplo, retorna o enésimo elemento de uma lista. Veja o exemplo:

```
val lista = ["Leandro","Leonardo","Julia","Laura","Jose","Marcia"];

fun elemento(n) = List.nth(lista,n); (* devolve o enésimo elemento da lista *)
```

Para uma lista de funções disponíveis na biblioteca List, utilize o comando comando:

help "List";

ou consulte a ajuda on-line da "*Standard ML Library*" (basta procurar no Google).

3.3 Manipulação de Strings

A biblioteca String tem várias funções interessantes de manipulação de strings. Para uma lista de funções disponíveis na biblioteca, utilize o comando:

help "String";

ou consulte a ajuda on-line da "*Standard ML Library*" (basta procurar no Google). Outra biblioteca interessante é a StringCvt, que converte e formata Strings.

3.4 Manipulação de números inteiros

A biblioteca Int tem funções de manipulação de números, principalmente de conversão. Para uma lista de funções disponíveis na biblioteca Int, utilize o comando:

help "Int";

ou consulte a ajuda on-line da "*Standard ML Library*" (basta procurar no Google).

4 Construções imperativas

3.1 Comando while

ML não é uma linguagem funcional pura e aceita o comando **while**, que permite realizar repetições. Não é uma construção funcional. Logo, sugere-se que as repetições sejam feitas com recursão. O comando while avalia uma expressão booleana (cujo resultado é *true* ou *false*). Para um laço baseado em contador, precisamos de uma variável mutável, que em ML é denominada de *mutator cell* (cabe salientar que tais construções não existem em funcionais puras e não são recomendadas, i.e., você conseguiria fazer o mesmo com uma recursão bem elaborada). O *mutator* mais adequado para o contador usa uma estrutura de dados específica que permite a mudança de seu estado – a estrutura **ref**. Ref declara uma referência que pode ser modificada pelo operador de atribuição, que em ML é " := ".

Exemplo:

```
load "Int";                                (* para poder converter int para String *)

fun regressiva i =
   let
      val c = ref i (* ref "cria" um objeto mutável – construção não funcional! *)
   in
      while !c > 0 do (           (* operador ! recupera conteúdo da referência *)
         print (Int.toString(!c) ^ "\n");
         c := !c - 1
      )
    end;
;
```

A mesma estrutura fica muito mais coerente (além de menor e expressiva) se for trocada pelo código seguinte:

```
load "Int";                                (* para poder converter int para String *)

fun regressiva 1 = print "1\n"
|   regressiva n = ( print (Int.toString(n) ^ "\n"); regressiva (n-1) );
```

3.2 E/S

O exemplo anterior utilizou o comando print.Além dele, uma série de comandos de E/S está disponível. Consulte a URL seguinte para maiores detalhes:
http://www.cs.cornell.edu/courses/cs312/2006fa/recitations/rec09.html.