==========
==========

Técnicas Alternativas de Programação: Programação Funcionalista

Alexandre Direne - Dep. de Informática - UFPR

==========
==========

Recomendação de Páginas Web:

- http://www.cs.cmu.edu/People/rwh/introsml/
- http://www.cs.cmu.edu/~petel/smlguide/smlnj.htm/
- http://cm.bell-labs.com/cm/cs/what/smlnj/index.html
- http://cm.bell-labs.com/cm/cs/what/smlnj/smlnj.html
- http://www.cs.nott.ac.uk/~gmh/faq.html
- http://www.brunel.ac.uk/~csstcmr/ml.html
- http://www.paulgraham.com/paulgraham/avg.html (sendo esta o relato de uma grande experiência de uso da linguagem de programação funcionalista LISP como base para a construção de um dos mais famosos softwares de comércio eletrônico.

Recomendação de Software e Bibliografia:

- Pacote de software para Programação Funcionalista com linguagem ML (Standard ML) - ambiente Poplog - obtido no seguinte endereço:

  `http://www.cs.bham.ac.uk/research/poplog/freepoplog.html`

- Introdução à Programação Funcional. Silvio R. de L. Meira, VI Escola de Computação de Campinas - SP, 1988.
- ML for the Working Programmer (2nd edition) Author: Larry C. Paulson Publisher: Cambridge University Press, 1996 ISBN (hardback): 0-521-57050-6 ISBN (paperback): 0-521-56543-X
- Robin Milner, Mads Tofte & Robert Harper The Definition of Standard ML MIT Press, 1990
- Elements of ML Programming, ML97 Edition Author: Jeffrey D. Ullman Publisher: Prentice-Hall Year: 1998 ISBN: 0-13-790387-1
- Introduction to Programming using SML Authors: Michael R. Hansen, Hans Rischel Publisher: Addison-Wesley Year: 1999 ISBN: 0-201-39820-6
- Concurrent Programming in ML Authors: John Reppy Publisher: Cambridge University Press Year: 1999 ISBN: 0-521-48089-2

- Purely Functional Data Structures Author: Chris Okasaki Publisher: Cambridge University Press Year: 1998 ISBN: 0-521-63124-6
- The Little MLer Authors: Matthias Felleisen and Dan Freidman Publisher: The MIT Press Year: 1998 ISBN: 0-262-56114-X
- Programming with Standard ML Authors: Chris Clack, Colin Myers, and Ellen Poon Publisher: Prentice-Hall Year: 1993 ISBN: 0-13-722075-8
- Elementary Standard ML Authors: Greg Michaelson Publisher: UCL Press Year: 1995 ISBN: 1-85728-398-8
- A Practical Course in Functional Programming Using Standard ML Author: Richard Bosworth Publisher: McGraw-Hill Year: 1995 ISBN: 0-07-707625-7
- Abstract Data Types in Standard ML Author: Rachel Harrison Publisher: John Wiley & Sons Year: 1993 ISBN: 0-471-938440
- ML Primer Authors: Ryan Stansifer Publisher: Prentice-Hall Year: 1992 ISBN: 0-13-561721-9
- Applicative High Order Programming: the Standard ML perspective Authors: Stefan Sokolowski Publisher: Chapman & Hall Computing Year: 1991 ISBN: 0-442-30838-8
- Elements of Functional Programming Authors: Chris Reade Publisher: Addison-Wesley Year: 1989 ISBN: 0-201-12915-9
- Pacote de software para Programação Funcionalista com linguagem Haskell - obtido juntamente com o tutorial nos seguintes endereços:

  `http://haskell.cs.yale.edu/hugs/`
  `http://www.dcs.gla.ac.uk/~jtod/discrete-mathematics/`
  `http://www.haskell.org/tutorial`

1 INTRODUÇÃO

O paradigma de programação funcionalista (comumente referida pelo termo "funcional") tem como principal conceito de programação, a abordagem das estruturas de dados ("objetos") do programa fonte como funções matemáticas. De forma mais detalhada, isto significa que funções podem (ou devem) ser:

- passadas como argumento;
- retornadas como resultado;
- guardadas como valores de variáveis;
- definidas recursivamente (na formação dos principais mecanismos de controle);
- utilizadas como os mais nobres veículos de modularização do código para atingir a definição abstrata de dados.

1.1 PRINCÍPIOS DE PROGRAMAÇÃO

1.1.1 FUNDAMENTAÇÃO MATEMÁTICA

Na matemática clássica, a definição do termo "função" é dada como o mapeamento entre dois conjuntos, um de entrada e um de saída, onde cada elemento do conjunto de entrada só tem um correspondente no conjunto de saída. No exemplo da função *fatorial*, entre os números *Naturais* (N), temos:

$fatorial : N \times N$
$fatorial = (0,1),(1,1),(2,2),(3,6),(4,24),(5,120),\ldots$

Em outras palavras, *fatorial* é um subconjunto do produto cartasiano $N \times N$. Tipicamente, a definição funcionalista matemática para *fatorial* seria a seguinte:

```
fatorial(n) = 1 para n = 0
fatorial(n) = n x fatorial(n-1) para n > 0
```

De forma sememlhante, no paradigma de programação funcionalista, podemos definir estruturas análogas às funções matemáticas para o cômputo de valores na memória. Assim, até mesmo aspectos da ordem na qual as instruções funcionalistas são apresentadas não é importante. Em outras palavras, ao contrário da programação no paradigma imperativo, no paradigma funcionalista as instruções acima poderiam ser apresentadas, sem prejuizo semântico, da seguinte forma:

```
fatorial(n) = n x fatorial(n-1) para n > 0
fatorial(n) = 1 para n = 0
```

1.1.2 VALORAÇÃO INDIRETA DA MEMÓRIA

Além de aspectos que diminuem a importância da ordem na qual as instruções são apresentadas, em programação funcionalista a alocação de valores na memória da máquina é feita de forma indireta (em relação ao programador). Isto também é típico na maioria das linguagens do paradigma de programação em Lógica. Sendo assim, a atribuição direta de valor (como no paradigma de programação Imperativa) não faz parte das tarefas de programação (estrimente) funcionalista. Analogamente, controles explícitos de repetição não são elementos que reinam neste mundo de programação.

Para elucidar melhor as idéias expostas, consideremos a seguinte equação:

```
X = 1
```

Esta equação só tem uma solução possível (com X assumindo o valor 1). De forma correspondente, se interpretarmos a mesma equação como um comando de linguagem iperativa, como "C", a variável X também receberá o valor 1.

Porém, se considerarmos a equação abaixo, a correposndência apresentada acima não se mantém.

```
X = X + 1
```

Na verdade, não há valor possível de X que satisfaça matematicamente à equação. Para expressar esta impossibilidade de cálculo, o significado da equação acima, quando interpretada segundo a maioria das linguagens funcionalistas, se traduz por uma substituição infinita de valores entre $X + 1$ e X. Já em linguagens imperativas, como "C", o valor da variável X seria 8, por exemplo, se ela valesse 7 imediatamente antes da avaliação do comando acima.

Um outro ponto a ser observado é que estaríamos cometendo um "erro de execução" (no sentido de linguagens imperativas) se a variável X não tivesse valor definido no momento da avaliação do comando acima. Todvia, em linguagens funcionalistas, é esperada uma uma maior flexibilidade no que se refere ao contexto da definição de valores de variáveis, além de uma correspondência entre os formatos das equações da matemática clássica e os das instruções de um programa. Veja, por exemplo, a sequência de euquações abaixo:

```
X = Y + Z + 1
Z = Y + 1
Y = 1
```

Note que, independente da ordem, por meio de uma linguagem funcionalista, o valor de X será calculado naturalmente como 4.

1.1.3 POLIMORFISMO E SOBRECARGA

O conceito de tipos é muito importante em liguagens funcionalistas, especialmente sob o prisma de tipos estáticos. Para facilitar a manutenção de tipos de forma coerente ao mesmo tempo que reaproveitamos código, um dos princípios de programação que devemos ter em mente quando utilizamos o paradigma funcionalista é o de polimorfismo (multi-tipos de argumentos e funções). Isto significa que uma ou mais funções genéricas podem ser criadas e, com o mesmo "nome" (ou símbolo), servir como veículo de definição abstrata entre mapeamentos de mais de um tipo.

Seja, por exemplo, uma instrução de troca entre os valores de duas variáveis. Em uma linguagem de programação imperativa, como PASCAL, isto só pode ser feito de forma modular, por meio de uma instrução de "procedure". Mesmo assim, o efeito colateral de troca precisa contar com a recepção dos parâmetros do procedimento em modo de "referência", já com os tipos pré-definidos e imutáveis para ambos os parâmetros. O código do procedimento "troca" em PASCAL poderia ser o seguinte:

```
procedure troca(var x, y : integer);
var aux : integer;
begin
  aux := x;
  x   := y;
  y   := aux;
and;
```

Em PASCAL, assim como em outras linguagens imperativas, como não é possível passar nem mesmo tipos por parâmetro, o procedimento acima só serviria mesmo para trocar valores entre variáveis inteiras, dois-a-dois. Trocar valores reais, que seja dois-a-dois, exigirá a definição de um outro procedimento, com código muito semelhante, e ainda, com nome forçosamente diferente.

Já em linguagens funcionalistas, este problema é amenizado por meio de polimorfismo. Assim, no caso de troca, dois-a-dois, de valores do tipo inteiro, real, string, ou outro qualquer, uma única função "troca" pode ser utilizada. O seu código seria algo próximo de:

```
troca : (A,B) --> (B,A)
```

Com isso, pares de objetos de qualquer tipo podem ter seus valores invertidos.

Caso seja necessário especializar o código da troca, dependendo do tipo dos argumentos, isto poderá ser feito por meio de multiplas definições de funções de mesmo nome.

1.1.4 A MODELAGEM DE DADOS COMO FUNÇÕES

Como visto anteriormente, os mais nobres elementos de dados deste paradigma são as funções. Na programação funcionalista, funções podem ser passadas como argumentos e retornadas como valores para, então, serem ou não armazenadas em variáveis.

Imagine como exemplos de aplicação direta e vantajosa desta característica, as funções *integral indefinida* e *derivada*. Elas tomam como argumento uma função e devolvem como resultado, outra função. Assim, a função integral indefinida da função x^3 é a função $\frac{x^4}{4}$.

Mesmo algumas das primeiras linguagens de programação que sugiram, as quais eram imperativas, já admitiam que uma abordagem funcionalista para a manipulação de dados seria muito intuitiva, pelo menos em certas situações. Por exemplo, em Fortran, uma definição de função poderia ser feita da seguinte maneira:

```
F(X,Y) = X**2 + 2*Y + 1
```

Até mesmo o tipo dos argumentos e dos valores retornados por uma função são mais bem definidos, assim como a quantidade destes agumentos e valores retornados.

1.2 TÉCNICAS DE PROGRAMAÇÃO

Em vistas das principais linguagens funcionalistas, tais como ML (STANDARD-ML), LISP, MIRANDA, SCHEME, HASKELL e outras, pode-se dizer que o conjunto de técnicas de programação é pouco variável e significativamente mais restrito do que em linguagens orientadas-a-objeto. Sendo assim, passaremos diretamente às instruções da linguagem alvo de nossa abordagem prática.

2 A LINGUAGEM STANDARD-ML

A sigla ML é a abreviatura de *Mata-Language*. Foi concebida originalmente com o nome de ML por Mike Gordon, Robin Milner e Chris Wadsworth da Universidade de Edinburgo (Escócia). Ela é uma linguagem fortemente enquadrada no paradigma funcionalista, com definição estática de tipos. ML inclui os seguintes fatores:

- aplicação recursiva de funções como o principal mecanismo de controle;
- tipos fortes (determinados em tempo de compilação);
- polimorfismo de tipos;
- tipos abstratos de dados para facilitar a modularização de programas;
- escopo estático (referências a identificadores são resolvidas em tempo de compilação);
- mecanismo para tratamento de exceção (exception handling) com verificação de tipo;
- definição incremental de instruções, as quais podem ser compiladas em separado, permitindo a implementação de sistemas de larga escala.

2.1 MODO INTERATIVO

Tudo que for digitado como elemento de entrada no modo interativo do ML será considerado como uma declaração. Por exemplo, a instrução abaixo significa que o nome "x" foi declarado para ter o valor inteiro 3.

```
- val x = 3;
  val x = 3 : int
```

Note que o símbolo reservado "val" indica que a instrução é de declaração de valor. O interpretador ML respondeu à instrução entrada com uma sequência das "amarrações" que conseguiu realizar, confirmado, na ordem em que aparecem: (1) a natureza declarativa -*val*- da instrução; (2) o nome de variável que recebeu valor; (3) o valor envolvido; (4) o tipo do valor.

Uma vez definido um nome, seu valor está disponível para apoiar a definição de novos nomes, como no exemplo abaixo.

```
- val y = 2 * x;
  val y = 6 : int
```

Por vezes, não há interesse em dar nomes explícitos para valores computados para o último nível de cálculo. Nestes casos, recomenda-se a utilização do nome especial "it", conforme o exemplo abaixo:

```
- val it = (x + y) * (x + y);
  val it = 81 : int
```

Pode-se ainda lançar mão de uma forma abreviada de declaração por meio de uma expressão fornecida individualmente, sem valor inicial. No caso de uma expressão qualquer, o interpretador ML irá considerá-la da mesma forma que:

```
- (x + y) * (x + y);
  val it = 81 : int
```

Em outras palavras, o nome "it" é sempre associado ao resultado da última expressão avaliada. Veja mais um exemplo:

```
- 4;
  val it = 4 : int
- it * it;
  val it = 16 : int
- it * it;
  val it = 256 : int
```

2.2 TIPOS BÁSICOS E COMPOSTOS

Toda expressão em ML tem um tipo. Ela é dada pela instrução:

```
<EXPRESSAO> : <TIPO>
```

Neste caso, o símbolo ":" significa "é do tipo". Os tipos básicos da linguagem ML são apresentados a seguir.

2.2.1 int

Exemplos:

```
1  ~1  256  ~300000
```

OBS: O operador pré-fixado "~" indica um número negativo.

2.2.2 real

Exemplos:

```
1.0  ~1.0  0.3333  1E5  2.35E~4
```

Reais e inteiros não podem ser misturados diretamente em expressões de um programa. No caso de mistura, as funções "`real`" e "`floor`" devem ser aplicadas. Exemplo:

```
- floor(22.0 / 7.0);
  val it = 3 : int
- real it;
  val it = 3.0 : real
```

2.2.3 string

Exemplos:

```
""  "a"  "Eva, depois de andar, viu a uva"  "2 + 2 = 4"
```

Os operadores de strings são "`size`" (que retorna o tamanho do string) e "`^`" (que concatena dois valores do tipo string). Exemplo:

```
- val mensagem = "Dia lindo\n";
  val mensagem = "Dia lindo\n" : string
- size(mensagem ^ mensagem ^ mensagem);
  val it = 30 : int
```

2.2.4 bool

Seus valores são apenas "`true`" e "`false`". Exemplo de seu uso pode ser:

```
- if x = y then "SIM" else "NAO";
  val it = "NAO" : string
```

2.2.5 unit

Este tipo só contém um único valor, denominado "`()`". Se assumirmos tal valor como expressão, teremos o seguinte:

```
- ();
  val it = () : unit
```

Este tipo é análogo ao tipo "`void`" de outras linguagens de programação, como C. Seu uso faz sentido nos casos onde se deve retornar um valor onde nenhum valor é necessário. Por exemplo, na ativação da função de saída (output), apenas o efeito colateral de gravação é necessário.

```
- output(std_out, mensagem);
Dia lindo
  val it = () : unit
```

2.2.6 Construindo Tipos Compostos

Por meio de construtores de tipos, pode-se construir tipos compostos a partir dos mais básicos. O primeiro construtor de tipos compostos é o tipo "`list`". Alguns exemplos de tipos compostos que podem ser formados com o construtor "`list`" são:

```
  string list
  unit list
  int list list
```

Os valores de listas dos tipos acima podem ser escritos com o auxílio de colchetes ("`[`" e "`]`") e vírgulas. Por exemplo:

```
- ["Eva" , "viu" , "a" , "uva" ] : string list;
  val it = ["Eva", "viu", "a", "uva"] : string list
- [(), (), (), ()] : unit list;
  val it = [(), (), (), ()] : unit list
- [[1, 2], [3, 4], [5, 6, 7]] : int list list;
  val it = [[1, 2], [3, 4], [5, 6, 7]] : int list list
```

Listas de elementos também podem ser construídas dinamicamente por meio do operador "`::`" o qual adiciona um ítem na frente de uma lista. Por exemplo:

```
- val numlist = [3, 4, 5];
  val numlist = [3, 4, 5] : int list
- 1 :: 2 :: numlist;
  val it = [1, 2, 3, 4, 5] : int list
```

Na verdade, na sua forma mais sistemática, e por isso útil para situações reais de programação, uma lista pode ser construída a partir da lista vazia ("`[]`" ou "`nil`). Isto pode ser feito assim:

```
- "Eva" :: "viu" :: "a" :: "uva " :: [];
  val it = ["Eva", "viu", "a", "uva "] : string list
- 1 :: 2 :: 3 :: 4 :: 5 :: nil;
  val it = [1, 2, 3, 4, 5] : int list
```

Atenção para o tipo especial de uma lista vazia!!!

```
- [];
  val it = [] : 'a list
```

O código de identificação " `'a` " (também chamado de ALPHA) é um "TIPO VARIÁVEL" que pode assumir qualquer tipo da linguagem ML. Devido à existência de um número infinito de tipos construídos com o tipo "list" mas apenas uma lista vazia, todos os tipos "list" devem poder compartilhar a mesma lista vazia. Sendo assim, o tipo da lista vazia é portanto um tipo genérico o qual pode ser instanciado para qualquer tipo particular.

Para listas não vazias, todo item da lista deve ser do mesmo tipo. Não há nenhum tipo válido em ML que possa ser associado a uma lista como a seguinte:

```
- [1, "Cardoso"] : ???
```

A mistura de tipos em expressões resulta em erro. Exemplo:

```
- val lista_mal_formada = [1, "Cardoso"];

Error: in expression
    [1, "Cardoso"]
Not all the list members have the same type:
    1 : int
    "Cardoso" : string
```

O mesmo grau de desvio ocorre quando se faz:

```
- val soma_mal_formada = 1 + "Cardoso";

Error: in expression
    1 + "Cardoso"
Cannot apply function
    op + : int * int -> int
to argument
    (1, "Cardoso") : int * string
```

Para construirmos agregados heterogêneos, a linguagem ML oferece o construtor de tipo tupla "`tuple`", aplicado por meio de um "`*`" na forma infixada. A sintaxe para formar valores do tipo-tupla inclui parêntesis e vírgulas. Exemplo:

```
- (1, 1.0) : int * real;
  val it = (1, 1.0) : int * real
- (1, 1.0, "a") : int * real * string;
  val it = (1, 1.0, "a") : int * real * string
- (1, 1.0, "a", true) : int * real * string * bool;
  val it = (1, 1.0, "a", true) : int * real * string * bool
```

No exemplo acima temos a definição de uma 2-tupla, uma 3-tupla e uma 4-tupla.
Outros construtores de tipos em ML são os seguintes:

- "`ref`"
  Serve para estruturas alteráveis da linguagem;
- "`labelled record`"
  Serve como extensão do construtor de tipos *tupla* que permite a etiquetagem de campos de uma estrutura com identificadores adicionais (será abordado adiante);
- "`function`"
  Serve como um dos principais veículos de modularização (será abordado adiante).

O tipo de um valor é normalmente deduzido pelo compilador ML sem a necessidade de declaração explícita. Porém, uma exceção a esta regra ocorre quando existe sobrecarga na definição de funções (mais detalhes adiante).

Sempre que uma declaração explícita de tipo é necessária, esta pode ser anexada a uma expressão (ou associação de valor) da mesma maneira vista anteriormente. Exemplo:

```
- val par : bool * int = (true, 0) and n : int = 1;
  val par = (true, 0) : bool * int
  val n = 1 : int
- par;
  val it = (true, 0) : bool * int
- n;
  val it = 1 : int
- val num1 = 7 and num2 = 8;
  val num1 = 7 : int
  val num2 = 8 : int
```

Tais construtores de tipos são importantes até mesmo para fins de documentação interna do programa, especialmente nas expressões que envolvem valores de tipos complexos, definidos pelo usuário-programador.

2.3 DEFINIÇÃO DE FUNÇÕES

Na linguagem ML, uma função é simplesmente um valor, assim como qualquer valor de outro tipo. O tipo mais genérico de uma função (ou módulo) é o seguinte:

```
'a -> 'b
```

A representação acima (pronunciada como "ALPHA VAI_EM BETA") expressa uma função que consome um argumento de um certo tipo ALPHA e devolve um resultado do tipo BETA (possivelmente diferente de ALFA). Funções específicas terão os TIPOS-VARIÁVEIS " `'a` " e " `'b` " instanciados de maneiras diferentes. Assim:

```
- real;
  val it = fn : int -> real
- floor;
  val it = fn : real -> int
```

Devido ao fato de valores funcionalistas não terem nenhum significado representativo em termos de impressão na saída padrão, estas são apresentadas pelo compilador ML apenas pelo termo "`fn`".

Como não há construtores de valor para o tipo função, uma sintaxe especial de instrução (utilizando o símbolo "`=>`") deve ser utilizada para se definir novos valores funcionalistas. O exemplo abaixo apresenta uma função que dobra o seu argumento.

```
- fn x => 2 * x;
  val it = fn : int -> int
- it 4;
  val it = 8 : int
```

Porém, cuidado:

```
- it;
  val it = 8 : int
- it 4;

Error: in expression
    it 4
The value
    it : int
is not a function

Setml
- fn x => 2 * x;
  val it = fn : int -> int
- it;
  val it = fn : int -> int
- it 4;
  val it = 8 : int
```

Também é possível dar nome a funções como se faz com outras declarações padronizadas de valor:

```
- val dobro = fn x => 2 * x;
  val dobro = fn : int -> int
- dobro 4;
  val it = 8 : int
```

Funções podem ser definidas de forma recursiva por meio da alocação do símbolo "`rec`", como no exemplo abaixo:

```
- val rec fat = fn n => if n = 0 then 1 else n * fat(n-1);
  val fat = fn : int -> int
- fat 6;
  val it = 720 : int
```

O uso de parênteses não é necessário, exceto nos casos onde a precedência de operadores deve ser subjulgada, tanto no que se refere à composição de argumentos quanto de funções. Veja os exemplos abaixo e compare-os:

```
- dobro 4 + 4;
  val it = 12 : int
- dobro (4 + 4);
  val it = 16 : int
```

OBSERVAÇÃO IMPORTANTE: Cada função consome apenas 1 (um) argumento. Isto é tradicional da teoria de funções na qual só há a especificação de um resultado para cada argumento.

Há duas maneiras de especificar funções que operam sobre mais de um argumento. Na primeira maneira, os valores de argumento são fornecidos para a função por meio de uma tupla. O exemplo abaixo mostra a definição da função "máximo", que vai de pares de inteiros em inteiros:

```
- val max : int * int -> int = fn (n, m) => if n >= m then n else m;
  val max = fn : int * int -> int
- max(0, 1);
  val it = 1 : int
```

Este conceito pode confundir programadores de outras linguagens, tais como "Pascal" ou "C", pois a ativação de "`max`" se parece com a passagem de 2 (dois) argumentos. Todavia, este não é o caso, pois, por definição sintática, o argumento de "`max`" é uma 2-tupla e é único. Veja o exemplo abaixo que reforça isto. O exemplo também demonstra o uso de uma expressão "`let`", a qual localiza uma declaração para uma expressão. Neste caso, a associação do valor de "`arg`" não está disponível no escopo global (como estaria se coincidisse com o valor de "`it`").

```
- let val arg = (0, 1) in max arg end;
  val it = 1 : int
- arg;

Error: unbound variable
    arg
```

Várias das funções internas já vistas da linguagem ML (tais como "`+`" e "`*`") são definidas com o mesmo sentido. Apenas a forma de ativação é que muda um pouco pois, em geral, estas funções internas são aplicadas de forma in-fixada aos argumentos. Mas se quisermos ativar estas funções internas de maneira pré-fixada, basta fazer o seguinte:

```
- op ^;
  val it = fn : string * string -> string
- op *(4, 2);
  val it = 8 : int
```

Se quisermos declarar "`max`" como sendo ativável de forma in-fixada também, devemos fazer:

```
- infix max;
  infix 0 max
- 0 max 1;
  val it = 1 : int
```

Na segunda maneira de definir uma função que requer mais de um argumento é por meio do uso de funções que retornam novas funções como resultados. Abaixo encontra-se uma função que adiciona um item no fim de uma lista (OBS: "`rev`" é uma função interna que inverte a ordem dos elementos de uma lista).

```
- val adiciona_no_fim = fn item => fn list => rev (item :: (rev list));
  val adiciona_no_fim = fn : 'a -> 'a list -> 'a list
```

Note bem que o argumento de "`adiciona_no_fim`" é um valor genérico ALPHA e o resultado é uma função que vai de listas de valores ALPHA em listas de valores ALPHA. Nesta segunda modalidade, o uso da função implicará em duas ativações a partir de uma única chamada:

```
- adiciona_no_fim 4 [1, 2, 3];
  val it = [1, 2, 3, 4] : int list
```

A vantagem desta modalidade é que podemos aplicar a função apenas uma vez, de forma mais específica para um dos argumentos, e guardar tal resultado parcial para ser aplicado posteriormente, com o argumento complementar.

```
- val adiciona_4_no_fim = adiciona_no_fim 4;
  val adiciona_4_no_fim = fn : int list -> int list
- adiciona_4_no_fim [1, 2, 3];
  val it = [1, 2, 3, 4] : int list
- adiciona_4_no_fim it;
  val it = [1, 2, 3, 4, 4] : int list
```

A sintaxe de declaração de funções em ML pode ser um pouco mais concisa por meio do termo "`fun`", da seguinte forma:

```
- fun adiciona_no_fim item list = rev (item :: (rev list));
  val adiciona_no_fim = fn : 'a -> 'a list -> 'a list
```

Funções declaradas com "`fun`" são implicitamente recursivas, não sendo necessário o uso do "`rec`".

Um outro exemplo é o da declaração de "`inteiros`" que, quando aplicada a um argumento inteiro "n", devolve uma lista contendo os inteiros de "1" a "n". A definição faz uso de uma expressão com "`let`" para especificar uma função local chamada "`de_menor_a_maior`", a qual gera uma lista de inteiros de um certo valor inicial a um valor final. Veja o código abaixo:

```
- fun inteiros n =
= let fun de_menor_a_maior A1 An = if A1 > An then []
= else A1 :: de_menor_a_maior(A1+1) An
= in de_menor_a_maior 1 n
= end;
  val inteiros = fn : int -> int list
- inteiros 10;
  val it = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : int list
```

2.4 FUNÇÕES COM SOBRECARGA

A complexidade de definição de tipos em ML aumenta quando o mesmo nome de função é aplicado a funções de tipos diferentes. Este conceito é chamado de "sobrecarga". Várias das funções matemáticas possuem esta qualidade. Por exemplo:

```
- 2 + 3;
  val it = 5 : int
- 2.0 + 3.0;
  val it = 5.0 : real
- "a" < "b";
  val it = true : bool
- 1 < 2;
  val it = true : bool
```

Tais nomes com sobrecarga devem ser utilizados de acordo com um tipo particular, o qual deve ser restringido por meio da indicação do mesmo como o pretendido. Por exemplo, como definição alternativa para a função "dobro", temos:

```
- fun dobro x = x + x;

Error: cannot determine a type for overloaded identifier
    val + : 'A * 'A -> 'A
```

O erro foi acarretado pela impossibilidade do compilador ML inferir se o argumento "x" é inteiro ou real. Neste caso, uma restrição de tipo se faz necessária para o argumento.

```
- fun dobro x : int = x + x;
  val dobro = fn : int -> int
```

Este conceito de sobrecarga se distingue do conceito de polimorfismo pois, neste último, a mesma função pode ser usada em objetos de tipos diferentes. É o caso do construtor de listas "`@`":

```
- fun duplica x = x @ x;
  val duplica = fn : 'a list -> 'a list
- duplica [2];
  val it = [2, 2] : int list
- duplica [2.0];
  val it = [2.0, 2.0] : real list
```

2.5 CASAMENTO DE PADRÕES

A declaração de uma função pode ser escrita com mais de uma cláusula onde as alternativas devem vir separadas por meio de uma "|" . Veja o exemplo para a definição de fatorial:

```
- fun fat 0 = 1
= |   fat n = n * fat(n-1);
  val fat = fn : int -> int
- fat 6;
  val it = 720 : int
```

O processo pelo qual uma única cláusula é escolhida para a aplicação da função é chamado de "casamento de padrão". Na definição acima, "0" e "n" são chamados de padrões.

O casamento de padrão se aplica também a construtores de tipos. Sendo assim, podemos casar padões de listas dinâmicas. Todavia, não é permitido casar padrões de funções pois estas não têm construtores de valor-objeto.

No contexto onde foram definidos

```
a : 'a
x : 'a list
```

podemos também definir o valor

```
(a :: x) : 'a list
```

A função "tamanho" (definida abaixo), usa casamento de padrão para medir o tamanho de uma lista:

```
- fun tamanho [] = 0
= |   tamanho (a::x) = 1 + tamanho x;
  val tamanho = fn : 'a list -> int
- tamanho [];
  val it = 0 : int
- tamanho [ 4 , 6 , 14 ];
  val it = 3 : int
```

OBS: é importante que a definição de uma função tenha cláusulas suficientes para casa com dados fornecidos para qualquer ativação da referida função. Quando isto não ocorre, uma mensagem "alerta" é dada em tempo de compilação.

```
- fun somalistas [] [] : int list = []
= |   somalistas (n :: ns) (m :: ms) = (n + m) :: somalistas ns ms;

Warning: possible Match exception in function
    somalistas

  val somalistas = fn : int list -> int list -> int list
```

Veja alguns exemplos de ativação da função "somalistas", alguns cobertos pela definição e outros não:

```
- somalistas [1] [1];
  val it = [2] : int list
- somalistas [1 , 6] [4 , 1];
  val it = [5, 7] : int list
- somalistas [1] [];
```

```
;;; Uncaught exception: Match

Setml
- somalistas [] [1];

;;; Uncaught exception: Match

Setml
- somalistas [4] [7 , 12];

;;; Uncaught exception: Match
```

Uma maneira de corrigir tal desvio é por meio da seguinte complementação da definição acima:

```
- fun somalistas [] [] : int list    = []
= |   somalistas L1 [] : int list    = L1
= |   somalistas [] L2 : int list    = L2
= |   somalistas (n :: ns) (m :: ms) = (n + m) :: somalistas ns ms;
```

Veja também o uso dos elementos "_" e "as" para tornar mais poderosa a definição de um padrão para a função que remove duplicatas de uma lista:

```
- fun remove_dups (a :: (l as b :: _)) =
=   if a = b then
=     remove_dups l
=   else
=     a :: remove_dups l
= | remove_dups l = l;
  val remove_dups = fn : ''a list -> ''a list
- remove_dups [1, 1, 2, 3, 4, 4, 5, 6, 6, 6, 7];
  val it = [1, 2, 3, 4, 5, 6, 7] : int list
```

Porém, não confunda a função "remove_dups" com alguma outra de nome parecido, tal como a famosa "apaga_todos". Como já era esperado pelo seu código, repare o comportamento da função "remove_dups" na ativação apresentada abaixo:

```
- remove_dups [1, 1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 4, 4, 4, 4];
  val it = [1, 2, 3, 4, 5, 6, 7, 4] : int list
```

Para a definição de "poda", precisaremos da função "apaga_todos".

```
fun apaga_todos (itm, [])      = []
|   apaga_todos (itm, cab :: cau) =
      if itm = cab then
        apaga_todos(itm, cau)
      else
        cab :: apaga_todos(itm, cau);
```

Após a compilação, temos o seguinte comportamento:

```
  val apaga_todos = fn : ''a * ''a list -> ''a list

- apaga_todos (4,  [4,5,4,5,5,5,5,5,5,5,6,6,6,4,7,4]);
  val it = [5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 7] : int list
```

Agora, em uma primeira aproximação, erramos ao construir "poda" da maneira abaixo:

```
fun poda []  =  []
|   poda(a :: (l as b :: _)) =
  if a = b then
    apaga_todos(a, l)
  else
    a :: apaga_todos(b, l);
```

Após a compilação, temos o seguinte comportamento (repare um valor "4" a mais, e ainda repare o sumio sinistro do "5"):

```
  val poda = fn : ''a list -> ''a list

- poda ([4,5,4,5,5,5,5,5,5,5,6,6,6,4,7,4]);
  val it = [4, 4, 6, 6, 6, 4, 7, 4] : int list
```

Em uma segunda versão, erramos novamente ao construir "poda":

```
fun poda []  =  []
|   poda (a :: b)  = a :: apaga_todos(a, b);
```

Após a compilação, temos o seguinte comportamento (repare agora que só o primeiro valor da lista, o "4", teve suas duplicatas eliminadas com sucesso):

```
  val poda = fn : ''a list -> ''a list

- poda ([4,5,4,4,4,4,4,4,4,5,5,5,5,5,5,5,6,6,6,4,4,4,4,7,4,4,4,4]);
  val it = [4, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 7] : int list
```

Finalmente conseguimos construir "poda" com sucesso. Repare bem a aplicação de "poda(apaga_todos(a,b))" (função-de-ção), o que é bem típico do paradigma funcionalista.

```
fun poda []  =  []
|   poda (a :: b)  =
      a :: poda(apaga_todos(a, b));
```

Após a compilação, temos o comportamento esperado:

```
  val poda = fn : ''a list -> ''a list

- poda ([4,5,4,4,4,4,4,4,4,5,5,5,5,5,5,5,6,6,6,4,4,4,4,7,4,4,4,4]);
  val it = [4, 5, 6, 7] : int list
```

Em outras palavras, na programação funcionalista, não se usa, nem é permitido fazer:

```
fun poda []  =  []
|   poda (a :: b)  =
      x = apaga_todos(a, b)
      a :: poda(x);

Error: unbound variable x
```