Linguagens de Programação

Programação Funcional: SML Básico

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Programação funcional
- Noções básicas de SML
- Anotações de tipos
- Padrões
- Variáveis locais
- Um exemplo de ordenação

INTRODUÇÃO

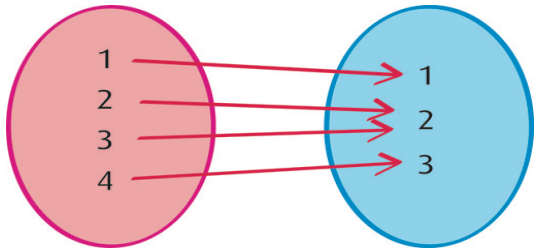
Linguagens de Programação

Introdução

- Programação funcional é baseada no conceito de funções matemáticas

$$f(x)$$

- Uma função matemática é um mapeamento de membros de um conjunto (**domínio**) em outro conjunto (**imagem**)



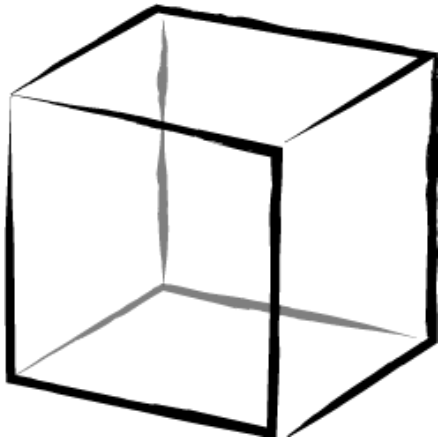
- A ordem de avaliação é controlada por recursões e expressões condicionais, ao invés de sequenciamento e iterações

- Não possuem efeito colateral

Uma Simples Função

- Exemplo: função *cubo*

```
cubo(x) = x * x * x
```



  - **Domínio:** conjunto dos números reais
  - `=` **:** a função é definida como
  - `x`**:** pode representar qualquer elemento do domínio, mas é fixo para representar apenas um elemento na avaliação

São diferentes de variáveis no paradigma imperativo

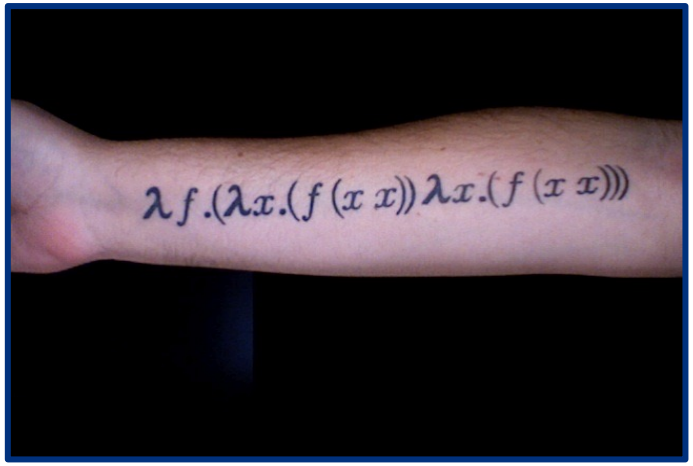
Formas Funcionais

- **Forma funcional** ou **função de ordem superior**
  - Aceita funções como parâmetros
  - Aceita funções como retorno

- Composição de funções ($h = f \circ g$)
  - Exemplo

```
f(x) = x + 2
g(x) = 3 * x
h(x) = f(g(x))
     = (3 * x) + 2
```

PROGRAMAÇÃO FUNCIONAL



Linguagens de Programação

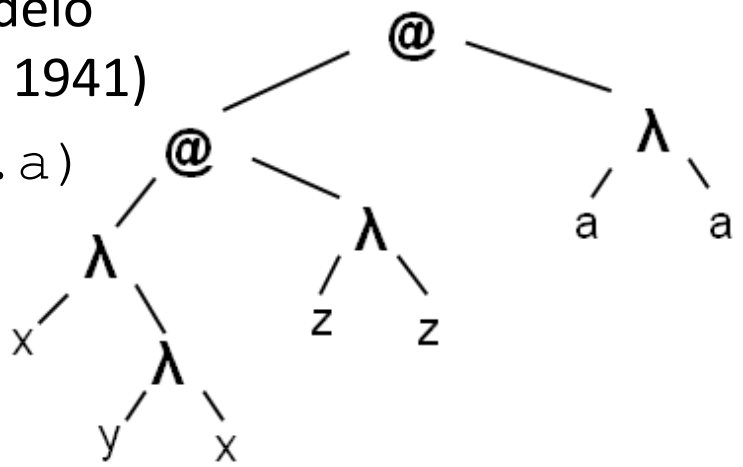
Programação Funcional

- **Um programa** no paradigma funcional **é uma função,** o qual pode ser uma composição de outras funções

$$P \equiv f\ (x_1, x_2, \ldots, x_n)$$
$$\equiv f_1 \circ f_2 \circ \ldots \circ f_n\ (x_1, x_2, \ldots, x_n)$$

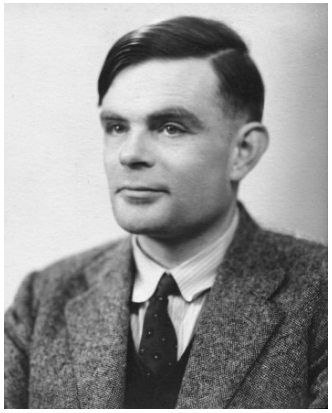
- O objetivo da programação é imitar uma função matemática

- Programação funcional é baseada no modelo computacional *Lambda Calculus* (Church, 1941)
  - Ex.: $((\lambda x.(\lambda y.x))(\lambda z.z))(\lambda a.a)$



Programação Funcional vs. Imperativa

- Exemplo de avaliação de `(x + y) / (a - b)`



Alan Turing

Programação imperativa

- Avalia `(x + y)` e armazena resultado na memória
- Avalia `(a - b)` e armazena resultado na memória
- Carrega ambos valores da memória e faz a divisão

Alonzo Church

Programação funcional

- Não usa variáveis como endereços de memória
- Usa recursão ao invés de iterações
- Não possui efeitos colaterais – dados os mesmos parâmetros sempre dará o mesmo resultado (**transparência referencial**)

NOÇÕES BÁSICAS DE SML

Linguagens de Programação

SML

- Standard ML (SML) é um dialeto de ML (*Meta Language*)
  - Embora existam vários outros dialetos, SML é bastante popular

- SML é uma linguagem de programação prática que suporta várias funcionalidades modernas
  - Inferência de tipos polimórficos
  - Funções de primeira ordem e de ordem superior
  - Continuações
  - Exceções
  - Tipos de dados com casamento de padrão
  - Coletor de lixo automático

Por que aprender SML?

Exemplo

- Considere o algoritmo de Euclides para cálculo do máximo divisor comum (MDC)

$$mdc(m,n) = \begin{cases} n & , \text{se } m = 0 \\ mdc(n \bmod m,\ m) & , \text{se } m > 0 \end{cases}$$

- Este algoritmo pode ser executado pelo sistema de linguagem (interpretador) SML/NJ♣

```
$ sml
Standard ML of New Jersey (64-bit)
- fun mdc m n = if m = 0 then n else mdc (n mod m) m;
val mdc = fn : int -> int -> int
- mdc 20 24;
val it = 4 : int
- ^D
```

O que acontece se você esquecer o ponto e vírgula?

Para que serve o `val it`?

Inferência de tipos

`Control+D` para sair

♣: https://www.smlnj.org/

Constantes

- Inteiros e reais

```
- 1234;
val it = 1234 : int
- 123.4;
val it = 123.4 : real
```

ML usa ~ (*til*) como operador de negação

- Lógicos (`bool`)

```
- true;
val it = true : bool
- false;
val it = false : bool
```

Cuidado: ML é *case-sensitive*, ou seja, deve-se usar `true` e não `True`, `TRUE` ou outra coisa

- Strings e caracteres

```
- "Hello";
val it = "Hello" : string
- "h";
val it = "h" : string
- #"h";
val it = #"h" : char
```

O que acontece se usar `#"abc"`?

Operadores

Cuidado: `div` e `mod` são usados com números inteiros, enquanto `/` com números reais.

- Operadores aritméticos (+, –, *, /, `div`, `mod`)

```
- ~1 + 2 - 3 * 4 div 5 mod 6;
val it = ~1 : int
- ~1.0 + 2.0 - 3.0 * 4.0 / 5.0;
val it = ~1.4 : real
```

O que acontece se usar `4/5`? E quanto é `~5 mod 2`?

- Operador para concatenação de strings (^)

```
- "oi " ^ "mundo";
val it = "oi mundo" : string
```

- Operadores de comparações de ordem (<, <=, >, >=)

```
- 2 < 3;
val it = true : bool
- 1.0 <= 1.0;
val it = true : bool
- #"d" > #"c";
val it = true : bool
- "abce" >= "abd";
val it = false : bool
```

Por que `"abce"` não é maior ou igual a `"abd"`?

Operadores

- Operadores de igualdade (=) e desigualdade (<>)

```
- "abc" = "abc";
val it = true : bool
- 2 <> 3;
val it = true : bool
```

Cuidado: nem todos os tipos podem ser testados por igualdade, como os números reais

- Operadores lógicos de disjunção (`orelse`), conjunção (`andalso`) e negação (`not`)

```
- 1 < 2 orelse 3 > 4;
val it = true : bool
- 1 < 2 andalso not (3 < 4);
val it = false : bool
```

O que acontece se fizer `true orelse 1 div 0 = 0`?

Os operadores `orelse` e `andalso` usam curto-circuito

Precedência dos Operadores

- Os operadores vistos são associativos à esquerda e são organizados nos seguintes níveis de precedência (mais alta para mais baixa)

```
not ~
```

```
* / mod div
```

```
+ - ^
```

```
< <= > >= = <>
```

```
andalso
```

```
orelse
```

Expressões condicionais

- Em ML, condicionais são expressões (ao invés de comandos)

```
- if 1 < 2 then #"x" else #"y";
val it = #"x" : char
- if 1 > 2 then 34 else 56;
val it = 56 : int
- (if 1 < 2 then 34 else 56) + 1;
val it = 35 : int
```

É possível usar `if` sem `else` em ML?

  - A sintaxe da expressão condicional é dada a seguir

<conditional-expression> ::=
if *<expression>* **then** *<expression>* **else** *<expression>*

Expressão lógica (`bool`)

Expressões do mesmo tipo

Conversões de Tipo

- Considere o seguinte erro de tipo em ML

```
- 1 * 2;
val it = 2 : int
- 1.0 * 2.0;
val it = 2.0 : real
- 1.0 * 2;
stdIn:5.1-5.8 Error: operator and operand do not agree
[overload - bad instantiation]
```

Operador * (e outros como + e <) são sobrecarregados para operar sobre diferentes tipos

Como resolver essa situação? Como outras linguagens lidam com ela?

Funções de Conversão

- Algumas funções pré-definidas para realizar conversões

Função	Parâmetro (Tipo)	Resultado (Tipo)	Descrição
real	int	real	Converter inteiro para real
floor	real	int	Arredondar para baixo
ceil	real	int	Arredondar para cima
round	real	int	Arredondar para o inteiro mais próximo
trunc	real	int	Truncar a parte decimal
ord	char	int	Encontrar o código ASCII do caractere
chr	int	char	Encontrar o caracter com o código ASCII
str	char	string	Converter de caracter para string

Aplicações de Funções

- Alguns exemplos de conversões

```
- real(123);
val it = 123.0 : real
- floor(3.6);
val it = 3 : int
- ceil 3.6;
val it = 4 : int
- str #"a";
val it = "a" : string
```

Você reparou que as funções podem usar parênteses ou não?

As seguintes expressões são equivalentes (função `f` chamada com parâmetro `1`):
`f(1)`, **`(f)1`**, **`(f)(1)`**, **`(f 1)`** e **`f 1`**

Associatividade de Funções

- Aplicação de funções é associativa à esquerda e tem maior precedência que qualquer outro operador
  - `f a+1` é equivalente a `(f a) + 1` e não `f (a + 1)`

```
- abs 1-2;
val it = ~1 : int
- abs (1-2);
val it = 1 : int
```

Variáveis

- Definição de variáveis

```
- val x = 1 + 2 * 3;
val x = 7 : int
- x;
val it = 7 : int
- val y = if x = 7 then 1.0 else 2.0;
val y = 1.0 : real
```

Variáveis começam com letra, seguido de zero ou mais letras, dígitos e/ou sublinhados

- Variáveis podem ser redefinidas, inclusive podem ter um novo tipo

```
- val n = 23;
val n = 23 : int
- n;
val it = 23 : int
- val n = true;
val n = true : bool
- n;
val it = true : bool
```

Cuidado: embora possam ser redefinidas, variáveis em linguagens funcionais são conceitualmente diferentes de variáveis em linguagens imperativas

Tuplas

- A maioria das linguagens de programação permitem chamar uma função com uma lista de parâmetros
  - Ex.: `f(1,2,3)`

Colocar `(1,2,3)` entre parênteses agrupa estes parâmetros em ordem para ser passado para a função

- SML suporta tuplas de uma forma mais geral que outras linguagens
  - Tuplas podem ser expressões em qualquer lugar (não só como parâmetros para funções)

```
- val barney = (1 + 2, 3.0 * 4.0, "brown");
val barney = (3,12.0,"brown") : int * real * string
- val point = ("red", (300,200));
val point = ("red",(300,200)) : string * (int * int)
```

Uma coleção ordenada de valores de diferentes tipos é normalmente chamada de **tupla**

Qual o significado do * (asterisco) nesta definição?

Tuplas

- O `*` (asterisco) neste caso não está sendo usado como operador de multiplicação, mas como um **construtor de tipos**
- Dados quaisquer dois tipos ML `a` e `b`, `a * b` é o tipo ML para a tupla de duas coisas, sendo o primeiro do tipo `a` e o segundo do tipo `b`
- Parênteses são importantes na definição de tuplas, os seguintes tipos são todos diferentes uns dos outros
  - `string * (int * int)`
  - `(string * int) * int`
  - `string * int * int`

Você consegue mostrar exemplos de definições para estes diferentes tipos?

Extrair Elementos de Tuplas

- Para extrair o n-ésimo elemento de uma tupla, usa-se `#n t`

```
- #2 barney;
val it = 12.0 : real
- #1 (#2 point);
val it = 300 : int
```

Em SML, existe tupla de apenas um elemento? Por exemplo, `(1+2)`?

Listas

- ML suporta listas, tal que a diferença de tuplas é que os elementos da lista devem possuir o mesmo tipo

- Listas são formadas com colchetes (ao invés de parênteses) e podem conter qualquer quantidade de elementos

```
- [1,2,3];
val it = [1,2,3] : int list
- [1.0, 2.0];
val it = [1.0,2.0] : real list
- [true];
val it = [true] : bool list
```

Neste caso, `list` é outro **tipo de construtor**

Listas

- Dado qualquer tipo ML `a`, o tipo ML `a list` aplica a uma lista de coisas de tipo `a`

- Listas podem conter quaisquer tipos de elementos, até tuplas e outras listas, desde que os elementos da lista tenham o mesmo tipo

```
- [(1,2), (1,3)];
val it = [(1,2),(1,3)] : (int * int) list
- [[1,2,3], [1,2]];
val it = [[1,2,3],[1,2]] : int list list
```

Tuplas vs Listas

- Considere o seguinte exemplo da diferença entre tuplas e listas

```
- val x = (1,2,3);
val x = (1,2,3) : int * int * int
- val y = [1,2,3];
val y = [1,2,3] : int list
```

  - A variável `x` é uma **tupla** de `3` inteiros
  - A variável `y` é uma **lista** de `3` inteiros

Cuidado: note que as definições de tipos delas são diferentes; isto significa que o que se pode fazer com cada uma delas é bem diferente uma da outra

Lista Vazia

- A lista vazia em ML é `nil` ou somente `[]`

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

Qual o tipo exato destas listas? É uma lista vazia de inteiros ou uma lista vazia de strings?

  - Nomes que começam com apóstrofo, como `'a list`, são **variáveis de tipos**; ou seja, cujo tipo é desconhecido

- A função `null` pode ser usada para testar se a lista é vazia

```
- null [];
val it = true : bool
- null [1,2,3];
val it = false : bool
```

Por que não usar o operador de comparação de igualdade (`v = []`)?

Concatenação de Listas

- O operador `@` é usado para concatenar duas listas

```
- [1,2,3] @ [4,5,6];
val it = [1,2,3,4,5,6] : int list
- 1 @ [2,3];
stdIn:1.2-1.11 Error: operator
and operand do not agree
```

É similar ao operador ^ para concatenação de strings

Cuidado: ambos os operandos devem ser listas

- Uma outra opção é usar o operador cons, escrito como `::`
  - É como se fosse colado um novo elemento no começo da lista, por exemplo, `1::[2,3]` avalia em `[1,2,3]`

```
- val x = #"c" :: [];
val x = [#"c"] : char list
- val y = #"b" :: x;
val y = [#"b",#"c"] : char list
- val z = #"a" :: y;
val z = [#"a",#"b",#"c"] : char list
```

Qual operador é mais útil, o operador `@` ou `::`?

`hd` e `tl`

- Duas funções importantes para extração de partes de uma lista são `hd` e `tl` (abreviações de *head* e *tail*)
  - `hd` retorna o primeiro elemento da lista
  - `tl` retorna a cauda da lista, ou seja, o restante da lista após o primeiro elemento

```
- val z = 1 :: 2 :: 3 :: [];
val z = [1,2,3] : int list
- hd z;
val it = 1 : int
- tl z;
val it = [2,3] : int list
- tl(tl z);
val it = [3] : int list
- tl(tl(tl z));
val it = [] : int list
```

O que acontece se usar as funções `hd` ou `tl` em uma lista vazia?

`explode` e `implode`

- Embora strings em ML não sejam equivalentes a lista de caracteres, elas possuem bastante coisa em comum

- A função `explode` converte um valor do tipo string em uma `char list`, enquanto a função `implode` faz a operação inversa

```
- explode "hello";
val it = [#"h",#"e",#"l",#"l",#"o"] : char list
- implode [#"h", #"i"];
val it = "hi" : string
```

Funções

- Para definir funções em ML, usa-se a definição `fun`

```
- fun firstChar s = hd (explode s);
val firstChar = fn : string -> char
- firstChar "abc";
val it = #"a" : char
```

Como ML descobriu o tipo desta função?

- O símbolo `->` é outro **construtor de tipos**
  - Dados quaisquer dois tipos `a` e `b`, `a -> b` é o tipo para funções que recebem o tipo `a` (domínio) e retornam um resultado do tipo `b` (imagem)

Funções

- A sintaxe da definição de `fun` é simples

```
<fun-def> ::= fun <function-name> <parameter> = <expression> ';'
```

  - `<function-name>`: o nome da função sendo definida
  - `<parameter>`: o nome do parâmetro (o mais simples é apenas uma variável)
  - `<expression>`: qualquer expressão ML; o valor desta expressão que é o que é retornado por esta função

Como escrever funções com mais de dois parâmetros?

Funções

- Para criar funções com mais parâmetros, pode-se usar tuplas

```
- fun quot (a, b) = a div b;
val quot = fn : int * int -> int
- quot (6, 2);
val it = 3 : int
```

- A sintaxe parece como em outras linguagens, mas em ML lida com tuplas de um jeito bem mais flexível

```
- val pair = (6, 2);
val pair = (6,2) : int * int
- quot pair;
val it = 3 : int
```

Não há nada especial sobre lista de parâmetros em funções, podem ser simplesmente tuplas

Exemplos de Funções

- Função para calcular o fatorial de números inteiros não negativos

```
- fun fact n =
=    if n = 0 then 1
=    else n * fact (n - 1);
val fact = fn : int -> int
```

A função pode ser escrita em mais de uma linha

  - Esta é uma definição de uma função recursiva
    - O caso base (`if n = 0 then 1`) que indica o que retornar para o menor valor legal de entrada possível
    - O passo recursivo (`else n * fact (n - 1)`) que a função chama ela mesmo com valores mais próximos à base

Linguagens funcionais fazem uso mais pesado de recursão do que linguagens imperativas, já que não possuem construções como laços `while`

Como fazer uma função que calcula o somatório de uma lista de inteiros?

Exemplos de Funções

- Função para calcular o somatório de uma lista de inteiros

```
- fun listsum x =
=   if null x then 0
=   else hd x + listsum (tl x);
val listsum = fn : int list -> int
- listsum [1, 2, 3, 4, 5];
val it = 15 : int
```

  - Esta função ilustra um padrão comum de funções recursivas
    - Aplica-se o caso base à lista vazia (`nil`) e o passo recursivo tem a oportunidade de olhar a cabeça da lista (`hd x`) e chamar a própria função recursivamente com a cauda da lista (`tl x`)

Como fazer uma função que calcula o tamanho de uma lista?

Exemplos de Funções

- Função para calcular o tamanho de uma lista

```
- fun length x =
=    if null x then 0
=    else 1 + length (tl x);
val length = fn : 'a list -> int
```

Repare o tipo (`a' list`) desta função `length`

  - Este é um exemplo de uma função polimórfica que permite parâmetros de tipos diferentes; ou seja, não é preciso escrever uma função de tamanho para cada tipo diferente

```
- length [true, false, true ];
val it = 3 : int
- length [4.0, 3.0, 2.0, 1.0];
val it = 4 : int
```

O que acontece se usar o teste `x = []` ao invés de `null x`?

Exemplos de Funções

- Função para calcular o tamanho de uma lista

```
- fun badlength x =
=   if x = [] then 0
=   else 1 + length (tl x);
val badlength = fn : ''a list -> int
```

Repare que o tipo agora é `''a list` (com um apóstrofo extra)

  - Variáveis de tipos que começam com apóstrofo duplo indicam que são testáveis por igualdade

Cuidado: como números reais não são testáveis por igualdade, essa função não funciona com lista de reais

```
- badlength [true, false, true];
val it = 3 : int
- badlength [4.0, 3.0, 2.0, 1.0];
stdIn:83.1-83.31 Error: operator and operand do not agree
[equality type required]
```

Como criar uma função que reverte uma lista?

Exemplos de Funções

- Função para reverter uma lista

```
- fun reverse L =
=   if null L then nil
=   else reverse (tl L) @ [hd L];
val reverse = fn : 'a list -> 'a list
- reverse [1,2,3];
val it = [3,2,1] : int list
```

ANOTAÇÕES DE TIPOS

Linguagens de Programação

Tipos

- Até agora foram vistos os tipos
  - `int`, `real`, `bool`, `char`, `string`

- E os construtores de tipos
  - `*`: para tuplas
  - `list`: para listas
  - `->`: para funções

Quando combinados, `list` tem a maior precedência e `->` a menor precedência

Como interpretar o tipo `int * int list`?

Anotação de Tipos

- ML descobriu e determinou todos os tipos que usamos até agora

Para que anotar explicitamente os tipos então?

- Porém, às vezes o sistema de inferência tipos de ML precisa de uma ajudinha e anotações de tipos são necessárias

```
- fun prod (a, b) = a * b;
val prod = fn : int * int -> int
```

Por que ML decidiu pelo tipo inteiro neste caso? Uma função para calcular o produto de dois números reais não teria a mesma definição?

Anotação de Tipos

- Para o exemplo anterior, ML não tem informações de tipos para `a` e `b` em `prod` a não ser pelo operador de multiplicação (`*`) aplicado a elas
  - O operador `*` poderia ser aplicado a inteiros e reais

- Quando não há dicas para o sistema de tipos, ML usa o operador padrão para a multiplicação (ou para a soma ou subtração) que é `int * int -> int`

- Para redefinir essa função usando números reais, é preciso dar uma dica mais definitiva: uma anotação de tipos

```
- fun prod (a: real, b: real) : real = a * b;
val prod = fn : real * real -> real
```

A anotação é dada por um `:` (dois pontos) seguida de um tipo

Anotação de Tipos

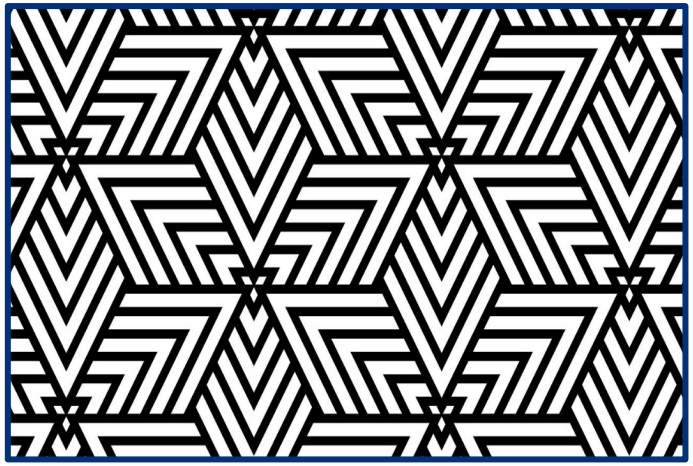
- Diferente de outras linguagens, ML permite anotação de tipos depois de qualquer variável ou expressão

```
fun prod (a, b) : real = a * b;
fun prod (a : real, b) = a *b;
fun prod (a, b : real) = a * b;
fun prod (a, b) = (a : real) * b;
fun prod (a, b) = a * b : real;
fun prod (a, b) = (a * b) : real;
fun prod ((a, b) : real * real) = a * b;
```

Todas estas definições de funções são equivalentes

Dica: às vezes apenas uma anotação é o suficiente para auxiliar na inferência de tipos, porém, o exemplo anterior talvez seja melhor por ser mais legível

PADRÕES



Linguagens de Programação

Padrões

- O suporte a padrões em ML é uma parte importante da linguagem

```
fun f n = n * n;
fun f (a, b) = a * b;
```

Padrões são um conceito diferente que não possuem equivalência em linguagens imperativas como C e Java

  - `n` parece um parâmetro e `(a, b)` parece uma lista de parâmetros, mas eles na verdade são **padrões** (*patterns*)

- ML tenta **casar** (*match*) valores com seus parâmetros e tomar ação dependendo se casam ou não
  - O padrão `n` casa qualquer parâmetro, enquanto o padrão `(a, b)` casa qualquer tupla de dois itens
  - Repare que novas variáveis (`n`, `a` e `b`) são criadas neste processo

Padrões ocorrem em várias partes de ML, não só em parâmetros

Padrões Simples

- O padrão mais simples de ML é o caractere _ (*underscore*)
  - É um padrão que casa qualquer coisa e não introduz uma nova variável

```
- fun f _ = "yes";
val f = fn : 'a -> string
```

Repare o tipo polimórfico 'a

Por que não usar a seguinte definição?
`fun f x = "yes";`

  - Agora esta função pode ser aplicada a um parâmetro de qualquer tipo

```
- f 34.5;
val it = "yes" : string
- f [];
val it = "yes" : string
```

Padrões Simples

- No outro extremo, você pode criar um padrão que casa apenas uma única coisa: uma constante

```
- fun f 0 = "yes";
stdIn:4.5-4.16 Warning: match nonexhaustive
          0 => ...
val f = fn : int -> string
```

Repare que o aviso indica que não cobriu todo o domínio de inteiros

  - Esta função define apenas uma aplicação (para a constante zero) e nenhuma outra

```
- f 0;
val it = "yes" : string
- f 1;
uncaught exception Match [nonexhaustive match failure]
  raised at: stdIn:4.16
```

Cuidado: a constante deve ter um tipo testável por igualdade, ou seja, você não pode usar uma constante real

Padrões Complexos

- Qualquer lista de padrões é um padrão legal

```
- fun f [a, _] = a;
stdIn:1.6-1.18 Warning: match nonexhaustive
          a :: _ :: nil => ...
val f = fn : 'a list -> 'a
- f [#"f", #"g"];
val it = #"f" : char
```

Repare o uso de colchetes indicando que o padrão é uma lista e não uma tupla

  - O padrão `[a, _]` casa qualquer lista com exatamente dois itens e introduz uma nova variável `a` para seu primeiro elemento

Cuidado: esta definição é uma função não exaustiva que irá falhar se usar uma lista com menos ou mais de dois elementos

Padrões Complexos

- O operador `::` (*cons*) também pode aparecer em um parâmetro

```
- fun f (x::xs) = x;
stdIn:1.6-2.1 Warning: match nonexhaustive
          x :: xs => ...
val f = fn : 'a list -> 'a
- f [1,2,3];
val it = 1 : int
```

  - O padrão `x::xs` casa qualquer lista não vazia e introduz a variável `x` associada ao elemento da cabeça da lista e `xs` a cauda da lista

Cuidado: os parênteses envolvendo `x::xs` não são realmente parte do padrão, mas são obrigatórios por causa da precedência

Múltiplos Padrões

- ML permite especificar múltiplos padrões para os parâmetros de uma função usando | (barra em pé)

```
- fun f 0 = "zero"
= |   f 1 = "one";
stdIn:10.5-11.16 Warning: match nonexhaustive
          0 => ...
          1 => ...
val f = fn : int -> string
```

  - Esta função possui dois corpos, um quando o parâmetro é 0 (zero) e outro quando o parâmetro é 1 (um)
  - Esta definição ainda é não exaustiva, pois não cobriu os outros números inteiros

Múltiplos Padrões

- A sintaxe geral para definição de funções com múltiplos padrões, tal que ela pode ter um ou mais corpos separados por | (barra em pé)

```
<fun-def>    ::= fun <fun-bodies> ';'
<fun-bodies> ::= <fun-body> | <fun-body> '|' <fun-bodies>
<fun-body>   ::= <fun-name> <parameter> = <expression>
```

  - Cada corpo da função repete o nome da função, porém provê um padrão diferente e uma expressão diferente

- Um padrão pode ter padrões sobrepostos

```
- fun f 0 = "zero"
= |   f _ = "non-zero";
val f = fn : int -> string
```

Em padrões sobrepostos, ML tenta os padrões na ordem em que aparecem e usa o primeiro que casa

  - Neste caso, `f` tem uma alternativa para o caso 0 e outra que cobre todos os outros casos

Estilo de Casamento de Padrões

- As seguintes definições são equivalentes

```
fun f 0 = "zero"
|   f _ = "non-zero";
```

```
fun f n =
  if n = 0 then "zero"
  else "non-zero"
```

  - A da esquerda usa o estilo de casamento de padrões, enquanto o da direita atinge o mesmo objetivo sem usar padrões alternativos

Qual destes dois estilos é mais recomendado?

Estilo de Casamento de Padrões

- A função para calcular fatorial pode também ser reescrita usando este estilo de casamento de padrões

```
fun fact n =
  if n = 0 then 1
  else n * fact (n-1);
```

```
fun fact 0 = 1
|   fact n = n * fact (n-1);
```

- Assim como a função para reverter uma lista

```
fun reverse L =
  if null L then nil
  else reverse (tl L) @ [hd L];
```

O operador `::` (*cons*) dividiu a lista em cabeça e cauda sem a necessidade de usar as funções `hd` e `tl`

```
fun reverse nil = nil
|   reverse (first::rest) = reverse rest @ [first];
```

Estilo de Casamento de Padrões

- Funções que operam em listas normalmente possuem a mesma estrutura: uma alternativa para o caso base (`nil`) e outra para o passo recursivo (`first::rest`) que chama a própria função (`rest`)
  - Para computar a soma dos elementos de uma lista

```
fun f nil = 0
|   f (first::rest) = first + f rest;
```

  - Para contar quantos valores verdadeiro têm em uma lista

```
fun f nil = 0
|   f (true::rest)  = 1 + f rest
|   f (false::rest) = f rest;
```

  - Para criar uma nova lista de seus elementos + 1

```
fun f nil = nil
|   f (first::rest) = first + 1 :: f rest;
```

Restrição

- Padrões em ML são poderosos, mas possuem uma restrição importante: o mesmo nome de variável não pode ser usado mais de uma vez no padrão
  - O código a seguir não funciona, pois é ilegal usar `a` duas vezes

```
fun f (a, a) = ... para pares de mesmo elemento
|   f (a, b) = ... para pares de elementos diferentes
```

  - É preciso reescrevê-lo

```
fun f (a, b) =
  if a = b then ... para pares de mesmo elemento
  else ... para pares de elementos diferentes
```

O Aviso *polyEqual*

- Comparar dois valores cujo tipo não pode ser determinado em tempo de execução gera um aviso de *polyEqual*

```
- fun eq (a, b) = if a = b then 1 else 0;
stdIn:40.22 Warning: calling polyEqual
val eq = fn : ''a * ''a -> int
```

Este tipo de comparação de igualdade é ineficiente, mas nem sempre é possível evitá-la

Padrões em Todos os Lugares

- Padrões não são usados apenas nas definições de funções, eles podem ser usados também em outros lugares

- Exemplos de padrões em definições `val`

```
- val (a, b) = (1, 2.3);
val a = 1 : int
val b = 2.3 : real
- val a :: b = [1, 2, 3, 4, 5];
val a = 1 : int
val b = [2,3,4,5] : int list
```

VARIÁVEIS LOCAIS

Expressões `let`

- Até agora foram usadas somente dois tipos de definições de variáveis
  - Em declaração `val` no alto nível
  - Em declarações de padrão em parâmetros de funções

- Existe uma forma alternativa, usando expressões `let` para definir variáveis locais

```
<let-exp> ::= let <definitions> in <expression> end
```

  - A parte `<definitions>` é uma sequência de qualquer quantidade de declarações (`val`)
  - A parte `<expression>` é avaliada em um ambiente em que estão as definições acima

Expressões `let`

- Exemplo de uma expressão `let`
  - A expressão `x + y` é avaliada em um ambiente em que `x` é 1 e `y` é 2

```
- let val x = 1 val y = 2 in x + y end;
val it = 3 : int
```

O valor `3` é o resultado desta expressão `let`

  - Estas definições (`x` e `y`) não são permanentes; são locais a esta expressão e são visíveis do ponto da definição (`let`) ao final (`end`)

```
- x;
stdIn:3.1 Error: unbound variable or constructor: x
```

Expressões `let`

- Por legibilidade, recomenda-se indentar expressões `let`

```
let
  val x = 1
  val y = 2
in
  x + y
end;
```

Dica: o uso de ; (ponto e vírgula) depois de cada definição `val` é opcional neste caso

- Expressões `let` são úteis para quebrar expressões grandes e dar nomes significativos para seus pedaços

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

Cada definição pode ser usada em definições subsequentes

UM EXEMPLO DE ORDENAÇÃO

Merge Sort

- O algoritmo Merge Sort será dividido em três partes, usando as seguintes funções
  - A função `halve` para dividir uma lista em duas partes
  - A função `merge` para juntar duas listas ordenadas
  - A função `mergeSort` que efetivamente fará a ordenação

A função `halve`

- A função `halve` que divide uma lista em um par de duas listas

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

É possível escrever o corpo do `let` de outra forma? Qual é melhor?

  - Exemplos de uso

```
- halve [1];
val it = ([1],[]) : int list * int list
- halve [1,2];
val it = ([1],[2]) : int list * int list
- halve [1,2,3,4,5,6];
val it = ([1,3,5],[2,4,6]) : int list * int list
```

A função `merge`

- A função `merge` que junta duas listas ordenadas

```
fun merge (nil, ys) = ys
|   merge (xs, nil) = xs
|   merge (x::xs, y::ys) =
      if x < y then x::merge (xs, y::ys)
      else y::merge (x::xs, ys);
```

  - Exemplos de uso

```
- merge ([2], [1,3]);
val it = [1,2,3] : int list
- merge ([1,3,4,7,8], [2,3,5,6,10]);
val it = [1,2,3,3,4,5,6,7,8,10] : int list
```

A função `mergeSort`

- A função `mergeSort` que usa as duas funções auxiliares criadas anteriormente (`halve` e `merge`)

```
fun mergeSort nil = nil
|   mergeSort [a] = [a]
|   mergeSort theList =
      let
        val (x,y) = halve theList
      in
        merge (mergeSort x, mergeSort y)
      end;
```

  - Exemplos de uso

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

A função `mergeSort` com funções encadeadas

```
(* Ordenar uma lista de inteiros *)
fun mergeSort nil = nil
|   mergeSort [a] = [a]
|   mergeSort theList =
      let
        (* Dada uma lista, dividir em duas *)
        fun halve nil = (nil, nil)
        |   halve [a] = ([a], nil)
        |   halve (a::b::cs) =
              let
                val (x, y) = halve cs
              in
                (a::x, b::y)
              end;

        (* Juntar duas listas ordenadas em uma *)
        fun merge (nil, ys) = ys
        |   merge (xs, nil) = xs
        |   merge (x::xs, y::ys) =
              if x < y then x::merge (xs, y::ys)
              else y::merge (x::xs, ys);

         val (x,y) = halve theList
      in
        merge (mergeSort x, mergeSort y)
      end;
```

Dica: use comentários entre (* e *)

ISSO É TUDO, PESSOAL!