Linguagens de Programação

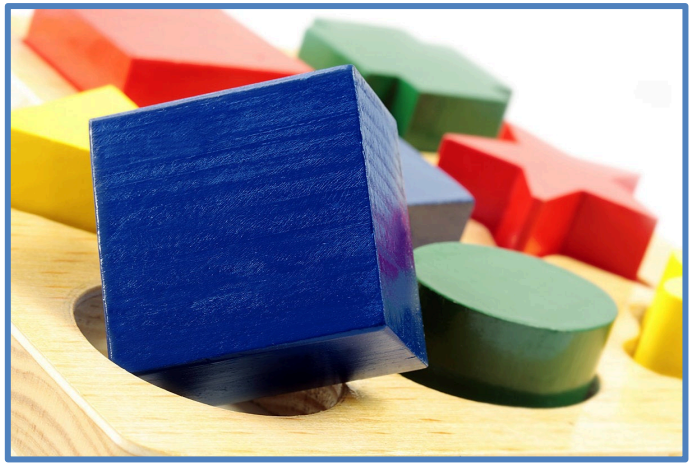
Programação Funcional: SML Avançado

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Casamento de padrões
- Valores funções e funções anônimas
- Funções de ordem superior e *currying*
- Tipos de dados
  - Enumerações
  - Construtores de dados com parâmetros
  - Construtores de tipos com parâmetros
  - Construtores de tipos definidos recursivamente

CASAMENTO DE PADRÕES



Linguagens de Programação

Casamento de Padrões

- Anteriormente foi visto o estilo de casamento de padrões em definições de funções

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

- A mesma sintaxe para casamento de padrões pode ser usada em diversas outras partes de ML, como expressões `case`

```
case n of
  0 => "zero" |
  _ => "non-zero"
```

Casamento de Padrões

- Um casamento (`match`) consiste em uma ou mais regras (`rule`) separadas por `|`

```
<rule>  ::= <pattern> => <expression>
<match> ::= <rule> | <rule> '|' <match>
```

  - O padrão de uma regra pode definir variáveis limitadas ao próprio casamento; o escopo destas definições vai do ponto de definição até o fim da regra

- Uma regra é como um bloco em ML, cada regra em um casamento deve ter o mesmo tipo da expressão do lado direito

Cuidado: casamentos não são expressões ML por si só, mas formam partes de diferentes tipos de expressões ML

Casamento de Padrões

- Uma importante expressão em ML que usa casamento é `case`

```
<case> ::= case <expression> of <match>
```

  - Segue um exemplo de uso de `case`

```
- case 1 + 1 of
=    3 => "three" |
=    2 => "two" |
=    _ => "hmm";
val it = "two" : string
```

Dica: ML não restringe onde se coloca `|`, mas é recomendado colocar cada caso em uma linha

Casamento de Padrões

- Muitas linguagens possuem algum tipo de construção que casa valores de expressões com constantes (ex.: `switch` em C), mas poucas (como ML) suportam casamento de padrões geral

```
case x of
   _::_::c::_ => c |
   _::b::_ => b |
   a::_ => a |
   nil => 0
```

O que esta expressão faz?

- A expressão `case` pode facilmente fazer tudo que uma expressão condicional pode fazer

```
if exp1 then exp2 else exp3
```


```
case exp1 of
   true => exp2 |
   false => exp3
```

VALORES FUNÇÕES E FUNÇÕES ANÔNIMAS

Linguagens de Programação

Valores Funções

- Quando o sistema de linguagens de ML é iniciado, muitas variáveis são definidas e ligadas, como é o caso de algumas funções

```
- ord;
val it = fn : char -> int
- ~;
val it = fn : int -> int
```

- Como essas variáveis carregam valores que são funções, é possível ligá-los a outras variáveis

```
- val x = ~;
val x = fn : int -> int
- x 3;
val it = ~3 : int
```

Valores Funções

- Na maioria das linguagens imperativas, uma definição de função cria um nome único e permanente a ela, porém em ML uma função não é ligada a um nome específico

Em ML, funções por si só não possuem nome; um ou mais nomes podem ser ligados a uma função

- A sintaxe `fun` para definição de funções ML cria a função e a liga a um nome automaticamente

```
- fun f x = x + 2;
val f = fn : int -> int
- f 1;
val it = 3 : int
```

Como criar uma função sem nome?

Funções Anônimas

- Para definir uma função anônima, basta usar a palavra reservada `fn` seguida de um casamento (`match`)

```
<fun-expr> ::= fn <match>
```

  - Por exemplo

```
- fn x => x + 2;
val it = fn : int -> int
- (fn x => x + 2) 1;
val it = 3 : int
```

- As seguintes sintaxes possuem o mesmo efeito♣

```
fun f x = x + 2;
```


```
val f = fn x => x + 2;
```

♣: O escopo da função `fun f` inclui o próprio corpo da função, enquanto a definição `val f` não; portanto somente a primeira definição pode ser usada recursivamente

Funções Anônimas

- Funções anônimas são úteis quando se precisa de uma função em apenas um lugar e não quer poluir o programa com mais um nome
  - Considere a função `quicksort` que recebe duas entradas, a lista a ser ordenada e uma função de comparação

```
- fun intBefore (a,b) = a < b;
val intBefore = fn : int * int -> bool
- quicksort ([1,4,3,2,5], intBefore);
val it = [1,2,3,4,5] : int list
```

Como aplicar o `quicksort` usando uma função anônima?

Funções Anônimas

- Pode-se criar uma função anônima para aplicação apenas no ponto necessário

```
- quicksort ([1,4,3,2,5], fn (a,b) => a<b);
val it = [1,2,3,4,5] : int list
- quicksort ([1,4,3,2,5], fn (a,b) => a>b);
val it = [5,4,3,2,1] : int list
```

- Existe ainda uma sintaxe alternativa ainda mais curta, tal qual é possível extrair a função de um operador usando a palavra reservada `op`

```
- op *;
val it = fn : int * int -> int
- quicksort ([1,4,3,2,5], op <);
val it = [1,2,3,4,5] : int list
```

Cuidado: infelizmente não é possível usar `quicksort(x, <)`, pois < é um operador binário

FUNÇÕES DE ORDEM SUPERIOR E CURRYING

Linguagens de Programação

Funções de Ordem Superior

- Toda função possui uma ordem
  - Uma função que não recebe nenhuma função como parâmetro e não retorna nenhuma função tem **ordem 1**
  - Uma função que recebe uma função como parâmetro ou retorna uma função tem **ordem `n`+1**, tal que `n` é a ordem da função de parâmetro ou retorno de maior ordem

- Funções com qualquer ordem maior que `1` são chamadas de **funções de ordem superior**

Qual a ordem da função `quicksort` vista anteriormente?

Funções de Ordem Superior

- Funções em ML recebem apenas um parâmetro, mas é possível passar mais de uma informação usando tuplas

```
- fun f (a,b) = a + b;
val f = fn : int * int -> int
- f (2,3);
val it = 5 : int
```

- Existe uma outra forma de fazer isto usando funções de ordem superior: basta escrever uma função que recebe o primeiro parâmetro e retorna uma função

```
- fun g a = fn b => a + b;
val g = fn : int -> int -> int
- g 2 3;
val it = 5 : int
```

Este truque é chamado de *currying*

Cuidado: aplicação de funções é associativa à esquerda, ou seja, `g 2 3` equivale a `(g 2) 3`

Currying

- Uma das claras vantagens de se usar *currying* é que não é preciso mais usar tuplas

```
- f (2,3);
val it = 5 : int
```

```
- g 2 3;
val it = 5 : int
```

- Porém, a maior vantagem é poder chamar funções *currying* passando apenas alguns de seus parâmetros (é possível especializar funções com valores iniciais específicos)

```
- val add2 = g 2;
val add2 = fn : int -> int
- add2 3;
val it = 5 : int
- add2 10;
val it = 12 : int
```

Currying

- Considere uma versão alternativa do `quicksort` com a ordem dos parâmetros invertida: o primeiro parâmetro agora é a função de comparação e o segundo a lista a ser ordenada

```
- quicksort (op <) [1,4,3,2,5];
val it = [1,2,3,4,5] : int list
```

- Pode-se usar *currying* nesta função da seguinte maneira

```
- val sortBackward = quicksort (op >);
val sortBackward = fn : int list -> int list
- sortBackward [1,4,3,2,5];
val it = [5,4,3,2,1] : int list
```

A função *curried* é muito mais útil que a versão com tuplas

Currying com Múltiplos Parâmetros

- *Currying* pode ser generalizado para qualquer quantidade de parâmetros

```
- fun g a = fn b => fn c => a + b + c;
val g = fn : int -> int -> int -> int
- g 1 2 3;
val it = 6 : int
```

- Existe uma forma mais curta, sem usar funções anônimas explicitamente, para escrever funções com *currying*

```
- fun g a b c = a + b + c;
val g = fn : int -> int -> int -> int
- g 1 2 3;
val it = 6 : int
```

As duas formas são equivalentes, mas qual delas é mais prática de escrever?

Funções de Ordem Superior Pré-Definidas

- ML define três funções de ordem superior que serão úteis
  - `map`
  - `foldr`
  - `foldl`

As funções `foldr` e `foldl` são similares, como o próprio nome indica, com uma sutil, mas importante, diferença

A Função `map`

- A função `map` é usada para aplicar uma função a todos os elementos de uma lista, coletando uma lista como resultado

```
- map ~ [1,2,3,4];
val it = [~1,~2,~3,~4] : int list
- map (fn x => x + 1) [1,2,3,4];
val it = [2,3,4,5] : int list
- map (fn x => x mod 2 = 0) [1,2,3,4];
val it = [false,true,false,true] : bool list
- map (op +) [(1,2), (3,4), (5,6)];
val it = [3,7,11] : int list
```

- Pode-se usar *currying* com a função `map`

```
- val f = map (op +);
val f = fn : (int * int) list -> int list
- f [(1,2), (3,4)];
val it = [3,7] : int list
```

A Função `foldr`

- A função `foldr` é usada para combinar todos os elementos de uma lista em um valor; esta função recebe três parâmetros
  1) Uma função `f`
  2) Um valor inicial `c`
  3) Uma lista de valores `[x1, ..., xn]`

  ...e computa o resultado usando a fórmula

$$f(x_1, f(x_2, \ldots f(x_{n-1}, f(x_n, c)) \ldots))$$

- Por exemplo

```
- foldr (op +) 0 [1,2,3,4];
val it = 10 : int
```

A computação é dada por:
`1+(2+(3+(4+0)))`

A Função `foldr`

Cuidado: foi preciso adicionar um espaço extra, já que `*)` é fechamento de comentário

- Outros exemplos de aplicação de `foldr`

```
- foldr (op * ) 1 [1,2,3,4];
val it = 24 : int
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldr (op ::) [5] [1,2,3,4];
val it = [1,2,3,4,5] : int list
```

Como usar `foldr` para eliminar os valores negativos de uma lista de inteiros?

- A função `foldr` também pode ser *currificada*

```
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr (op +);
val it = fn : int -> int list -> int
- foldr (op +) 0;
val it = fn : int list -> int
```

```
- val addup = foldr (op +) 0;
val addup = fn : int list -> int
- addup [1,2,3,4,5];
val it = 15 : int
```

A Função `foldl`

- A função `foldl` é usada para combinar todos os elementos de uma lista em um valor; esta função recebe três parâmetros
  1) Uma função `f`
  2) Um valor inicial `c`
  3) Uma lista de valores $[x_1, \ldots, x_n]$

  ...e computa o resultado usando a fórmula

$$f(x_n, f(x_{n-1}, \ldots f(x_2, f(x_1, c))\ldots))$$

- Por exemplo

```
- foldl (op +) 0 [1,2,3,4];
val it = 10 : int
```

A computação é dada por:
`4+(3+(2+(1+0)))`

Qual a diferença de `foldl` para `foldr`?

A Função `foldl`

- A diferença sutil, mas importante, é que `foldl` começa com o elemento mais a esquerda, e processa os elementos da esquerda para a direita
  - Para algumas operações que são associativas e comutativas, como adição e multiplicação, o resultado será o mesmo

```
- foldr (op +) 0 [1,2,3,4];
val it = 10 : int
- foldr (op * ) 1 [1,2,3,4];
val it = 24 : int
```

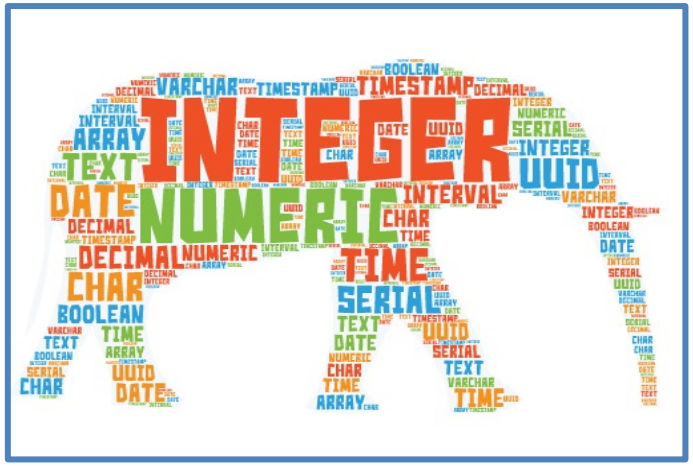

```
- foldl (op +) 0 [1,2,3,4];
val it = 10 : int
- foldl (op * ) 1 [1,2,3,4];
val it = 24 : int
```

  - Para outras, a ordem de aplicação é relevante

```
- foldr (op ^) "" ["ab","cd"];
val it = "abcd" : string
- foldr (op -) 0 [1,2,3,4];
val it = ~2 : int
```


```
- foldl (op ^) "" ["ab","cd"];
val it = "cdab" : string
- foldl (op -) 0 [1,2,3,4];
val it = 2 : int
```

TIPOS DE DADOS



Definição de Tipos

- Alguns tipos de dados em ML não são primitivos, mas pré-definidos, como é o caso dos tipos
  - bool

```
datatype bool = true | false;
```

  - list

```
datatype 'element list = nil |
                  :: of 'element * 'element list;
```

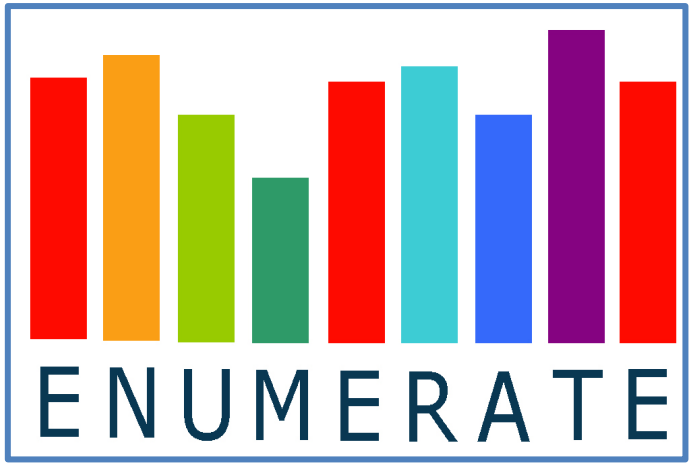
Em ML pode-se definir seus próprios tipos

Definição de Tipos

- Novos tipos podem ser definidos usando a palavra-reservada `datatype`, que definem ambos
  - **Construtores de tipos**: para criar novos tipos (possivelmente polimórficos)
  - **Construtores de dados**: para criar valores destes novos tipos

Por que eles são chamados de **construtores**?

TIPOS DE DADOS > ENUMERAÇÕES



Enumerações

- `datatype` pode ser usado para criar um novo tipo enumerado

```
- datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
datatype day = Fri | Mon | Sat | Sun | Thu | Tue | Wed
- fun isWeekDay x = not (x = Sat orelse x = Sun);
val isWeekDay = fn : day -> bool
- isWeekDay Mon;
val it = true : bool
- isWeekDay Sat;
val it = false : bool
```

Dica: capitalize a primeira letra dos construtores de dados.

  - `day` é o novo **construtor de tipos**
    - Não possui parâmetros, ou seja, não é polimórfico (existe apenas um tipo `day`)
  - `Mon`, `Tue`, ... são os novos **construtores de dados**
    - Também não recebem parâmetros, são valores constantes do tipo `day`

Tipagem Estrita

- ML é estrito com relação aos novos tipos

```
- datatype flip = Heads | Tails;
datatype flip = Heads | Tails
- fun isHeads x = (x = Heads);
val isHeads = fn : flip -> bool
- isHeads Tails;
val it = false : bool
- isHeads Mon;
stdIn:10.1-10.12 Error: operator and operand do not agree
[tycon mismatch]
  operator domain: flip
  operand:         day
```

Diferentemente de enumerações (`enum`) de C, ML não expõe detalhes de implementação ao programador

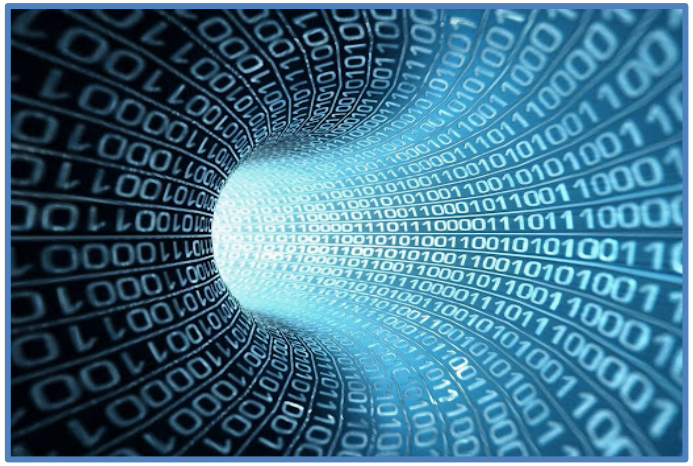
Construtores de Dados em Padrões

- É possível utilizar construtores de dados em padrões

```
fun isWeekDay Sat = false
|   isWeekDay Sun = false
|   isWeekDay _ = true;
```

Neste exemplo os construtores de dados são usados como constantes

TIPOS DE DADOS > CONSTRUTORES DE DADOS COM PARÂMETROS



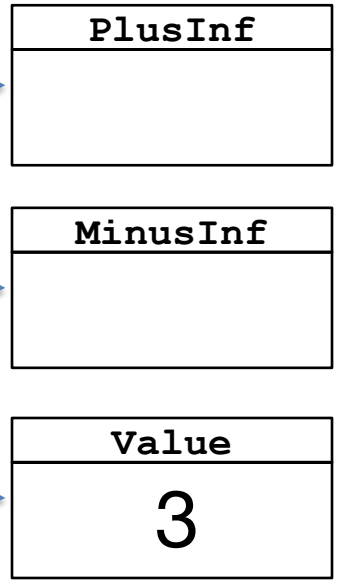
Invólucros (*Wrappers*)

- Pode-se adicionar um parâmetro de qualquer tipo a um construtor de dados usando a palavra reservada `of`

```
datatype exint = Value of int | PlusInf | MinusInf;
```

  - Na prática, este construtor é como se fosse um invólucro que contém um item de determinado tipo ou vazio

```
- PlusInf;
val it = PlusInf : exint
- MinusInf;
val it = MinusInf : exint
- Value 3;
val it = Value 3 : exint
```



  - `Value` é uma função que, se aplicada a `int`, retorna `exint`

```
- Value;
val it = fn : int -> exint
```

Extraindo Parâmetros com Casamento de Padrões

- Embora um `Value` contenha um `int`, ele não pode ser tratado como um inteiro

```
- val x = Value 5;
val x = Value 5 : exint
- x * x;
stdIn:5.3 Error: overloaded variable not defined at type
  symbol: *
  type: exint
```

- Para usar o inteiro, pode-se usar casamento de padrões

```
- val (Value y) = x;
stdIn:1.6-1.19 Warning: binding not exhaustive
          Value y = ...
val y = 5 : int
```

Este caso só funcionou porque `x` é `Value` e não `PlusInf` ou `MinusInf`

Casamento de Padrões Exaustiva

- Em alguns contextos, será preciso usar um padrão exaustivo em uma cláusula `case` que cobre todas as possibilidades do tipo

```
- val s = case x of
=    PlusInf => "infinity" |
=    MinusInf => "-infinity" |
=    Value y => Int.toString y;
val s = "5" : string
```

`Int.toString` é uma função pré-definida que converte um `int` em `string`

- Estes casamento de padrões em funções é especialmente importante

```
- fun square PlusInf = PlusInf
= |    square MinusInf = PlusInf
= |    square (Value x) = Value (x*x);
val square = fn : exint -> exint
- square (Value 3);
val it = Value 9 : exint
```

O que acontece na chamada `square (Value 1000000000)`?

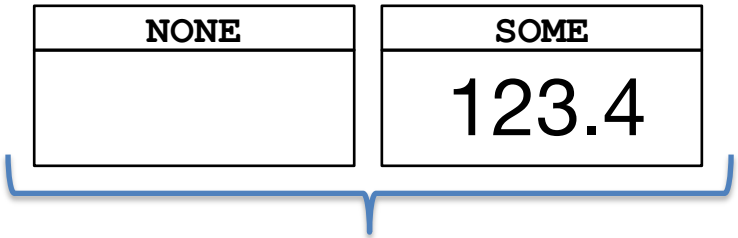
TIPOS DE DADOS > CONSTRUTORES DE TIPOS COM PARÂMETROS

Construtores de Tipos com Parâmetros

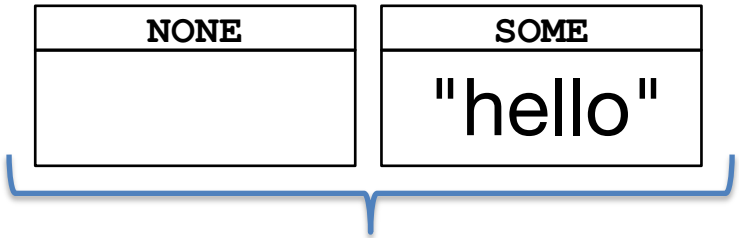
- O construtor de tipos de uma definição `datatype` também pode receber parâmetros

```
datatype 'a option =
    NONE |
    SOME of 'a;
```

  - Os parâmetros de um construtor de tipos são **parâmetros de tipos** ou **variáveis de tipos**
    - `option` forma tipos como `int option` ou `real option`, como `list` forma tipos como `int list` ou `real list`
  - O resultado é um **construtor de tipos polimórfico**



Possíveis valores para `real option`



Possíveis valores para `string option`

Usos para Tipos com Parâmetros

- Um outro exemplo de tipo com parâmetros: `bunch`

```
datatype 'x bunch =
    One of 'x |
    Group of 'x list;
```

`'x bunch` é uma coisa do tipo `'x` ou uma lista de coisas do tipo `'x`

  - ML, como sempre, infere automaticamente os tipos de `bunch`

```
- One 1.0
val it = One 1.0 : real bunch
- Group [true,false];
val it = Group [true,false] : bool bunch
```

Usos para Tipos com Parâmetros

- Funções polimórficas como `size` podem usar `bunch` sem resolver seus tipos

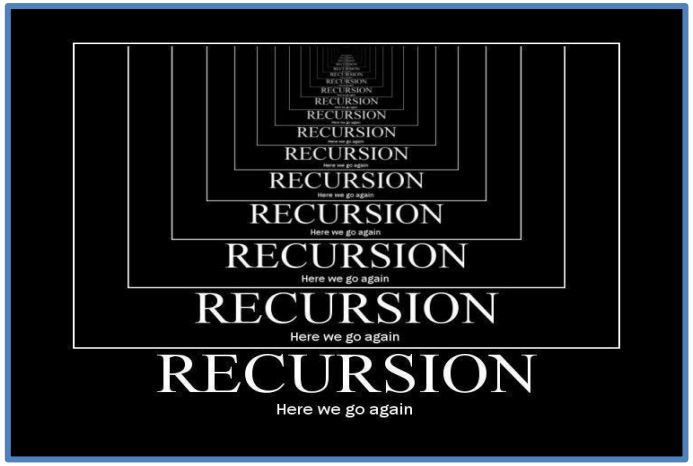
```
- fun size (One _) = 1
= |   size (Group x) = length x;
val size = fn : 'a bunch -> int
- size (One 1.0);
val it = 1 : int
- size (Group [true,false]);
val it = 2 : int
```

- Ou pode-se usar `bunch` em uma função sem polimorfismo

```
- fun sum (One x) = x
= |   sum (Group xlist) = foldr (op +) 0 xlist;
val sum = fn : int bunch -> int
- sum (One 5);
val it = 5 : int
- sum (Group [1,2,3]);
val it = 6 : int
```

`foldr` é usado com o operador + que resolve para o tipo `int`

TIPOS DE DADOS >
CONSTRUTORES DE TIPOS DEFINIDOS RECURSIVAMENTE



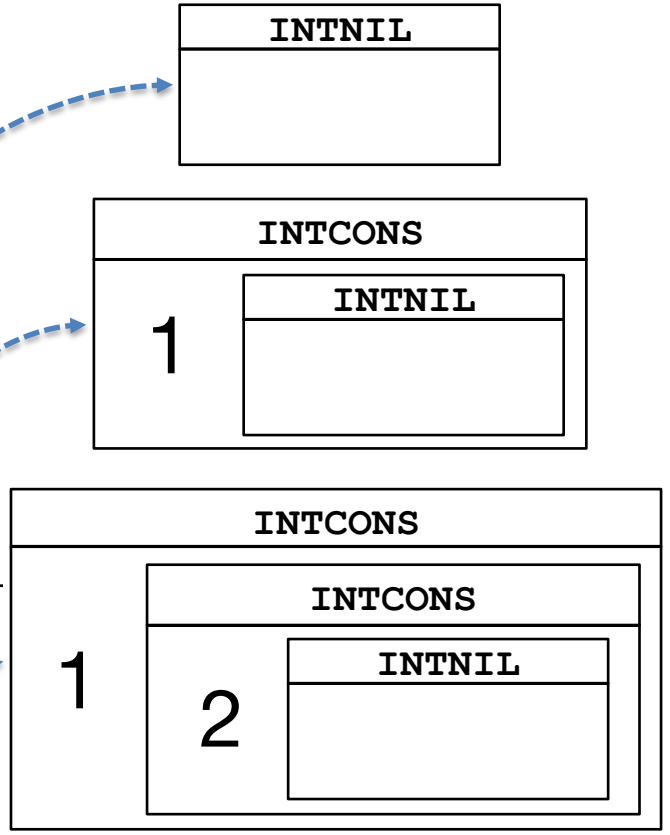
Construtores de Tipos Definidos Recursivamente

- O construtor de tipos sendo definido pode ser usado em seus próprios construtores de dados

```
datatype intlist =
    INTNIL |
    INTCONS of int * intlist;
```

- Por exemplo

```
- INTNIL;
val it = INTNIL : intlist
- INTCONS (1, INTNIL);
val it = INTCONS (1,INTNIL) : intlist
- INTCONS (1, INTCONS (2, INTNIL));
val it = INTCONS (1,INTCONS (2,INTNIL)) : intlist
```



Usos para Tipos Definidos Recursivamente

- A função `length` para o tipo `int list` (de ML) vs `intlist` (previamente construído)

```
fun listLength nil = 0
|   listLength (_::tail) =
        1 + (listLength tail);
```


```
fun intlistLength INTNIL = 0
|   intlistLength (INTCONS(_,tail)) =
        1 + (intlistLength tail);
```

A função `intlistLength` funciona somente com inteiros, como consertá-la para funcionar com qualquer tipo?

Tipos Paramétricos Definidos Recursivamente

- Definição de um tipo recursivo paramétrico `mylist` quase igual ao tipo pré-definido `list`

```
datatype 'element mylist =
    NIL |
    CONS of 'element * 'element mylist;
```

  - ML, como sempre, infere automaticamente os tipos de `mylist`

```
- NIL;
val it = NIL : 'a mylist
- CONS (1.0, NIL);
val it = CONS (1,NIL) : real mylist
- CONS (1, CONS (2, NIL));
val it = CONS (1,CONS (2,NIL)) : int mylist
```

Usos para Tipos Paramétricos Definidos Recursivamente

- A função `myListLength` para contar a quantidade de elementos

```
fun myListLength NIL = 0
|   myListLength (CONS (_, tail)) =
        1 + myListLength(tail);
```

- A função `addUp` para somar os números de uma lista de inteiros

```
fun addup NIL = 0
|   addup (CONS (head, tail)) =
        head + addup(tail);
```

- A função `myfoldr` similar a função `foldr`

```
fun myfoldr f c NIL = c
|   myfoldr f c (CONS (head, tail)) =
        f(head, myfoldr f c tail);
```

Como usar essa função `myfoldr`?

Cuidado: `: :` é um operador, mas `CONS` não

Tipos Paramétricos Definidos Recursivamente

- Definição de um tipo recursivo paramétrico `tree` para modelar uma árvore binária

```
datatype 'data tree =
    Empty |
    Node of 'data tree * 'data * 'data tree;
```

  - ML, como sempre, infere automaticamente os tipos de `tree`

```
- val treeEmpty = Empty;
val treeEmpty = Empty : 'a tree
- val tree2 = Node (Empty, 2, Empty);
val tree2 = Node (Empty,2,Empty) : int tree
- val tree123 = Node (Node (Empty, 1, Empty),
=                    2,
=                    Node (Empty, 3, Empty));
val tree123 = Node (Node (Empty,1,Empty),2,Node (Empty,3,Empty))
: int tree
```

Usos para Tipos Paramétricos Definidos Recursivamente

- A função `incall` que obtém uma nova árvore, tal que cada inteiro é somado em 1

```
- fun incall Empty = Empty
= |   incall (Node (x, y, z)) =
          Node (incall x, y+1, incall z);
val incall = fn : int tree -> int tree
- incall tree123;
val it = Node (Node (Empty,2,Empty),3,Node (Empty,4,Empty)) :
int tree
```

- A função `sumall` que soma todos os elementos de uma árvore

```
- fun sumall Empty = 0
= |   sumall (Node (x, y, z)) =
=         sumall x + y + sumall z;
val sumall = fn : int tree -> int
- sumall tree123;
val it = 6 : int
```

Usos para Tipos Paramétricos Definidos Recursivamente

- A função `listall` que obtém todos os elementos de uma árvore

```
- fun listall Empty = nil
= |   listall (Node (x, y, z)) =
=       listall x @ y :: listall z;
val listall = fn : 'a tree -> 'a list
- listall tree123;
val it = [1,2,3] : int list
```

- A função `isintree` que verifica se um elemento está na árvore

```
- fun isintree x Empty = false
= |   isintree x (Node (left, y, right)) =
=       x = y orelse isintree x left orelse isintree x right;
stdIn:48.9 Warning: calling polyEqual
val isintree = fn : ''a -> ''a tree -> bool
- isintree 4 tree123;
val it = false : bool
- isintree 3 tree123;
val it = true : bool
```

ISSO É TUDO, PESSOAL!