Linguagens de Programação

Programação Funcional e Haskell
Compreensão de Listas

Thiago Alves

Compreensão de Conjuntos

Na matemática, a notação de compreensão pode ser usada para construir novos conjuntos a partir de antigos.

$$\{x^2 \mid x \in \{1...5\}\}$$

O conjunto {1,4,9,16,25} de todos os números x^2 tal que x é um elemento de {1...5}.

Compreensões de Listas

Em Haskell, uma notação de compreensão similar pode ser usada para construir novas listas a partir de outras.

```
[x^2 | x <- [1..5]]
```

A lista [1,4,9,16,25] de todos os números x^2 tal que x é um elemento da lista [1..5].

- A expressão x ← [1..5] é chamada de gerador, pois indica como gerar valores para x.

- Compreensões podem ter múltiplos geradores, separados por vírgulas:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

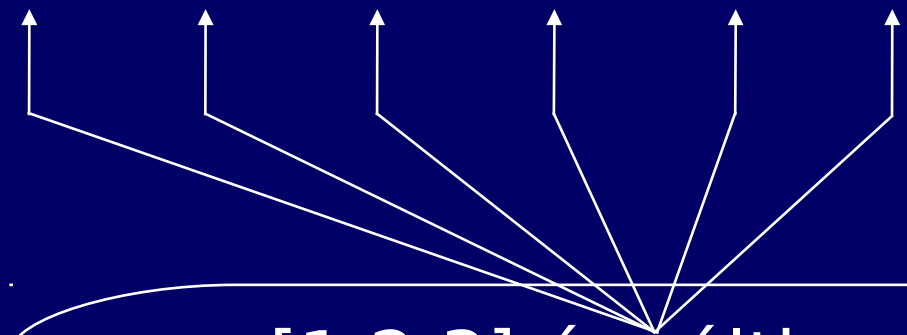
- Mudar a ordem dos geradores muda a ordem dos elementos na lista final:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Múltiplos geradores são como loops aninhados.

- Por exemplo:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```



x ← [1,2,3] é o último gerador, então o valor do componente x de cada par muda com maior frequência.

Geradores Dependentes

Geradores posteriores podem depender das variáveis introduzidas em geradores anteriores.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

A lista [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
de todos os pares de números (x,y)
tal que x,y são elementos da lista [1..3] e y ≥ x.

Usando geradores dependentes, podemos definir a função que concatena uma lista de listas:

```
concat :: [[a]] → [a]
```

Por exemplo:

```
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

Usando geradores dependentes, podemos definir a função que concatena uma lista de listas:

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

Por exemplo:

```
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

Podemos usar o _ para descartar certos elementos de uma lista

Vamos definir a função que seleciona os primeiros componentes de uma lista de pares:

```
firsts :: [(a,b)] → [a]
```

Por exemplo:

```
> firsts [(1,'a'),(5,'c'),(7,'d')]
[1,5,7]
```

Podemos usar o _ para descartar certos elementos de uma lista

Vamos definir a função que seleciona os primeiros componentes de uma lista de pares:

```
firsts :: [(a,b)] → [a]
firsts xs = [x | (x,_) ← xs]
```

Por exemplo:

```
> firsts [(1,'a'),(5,'c'),(7,'d')]
[1,5,7]
```

A função que retorna o tamanho de uma lista pode ser definida trocando cada elemento por 1 e realizando a soma:

```
length :: [a] → Int
```

Por exemplo:

```
> length [5,7,6]
3
```

A função que retorna o tamanho de uma lista pode ser definida trocando cada elemento por 1 e realizando a soma:

```
length :: [a] → Int
length xs = sum [1 | _ ← xs]
```

Por exemplo:

```
> length [5,7,6]
3
```

Guards

Compreensões de listas podem usar guards para restringir os valores produzidos por geradores anteriores.

```
[x | x ← [1..10], even x]
```

A lista [2,4,6,8,10] de todos os números x tal que x é um elemento da lista [1..10] e x é par.

Usando um guard, podemos definir uma função que mapeia um inteiro positivo para a sua lista de divisores:

```
factors  :: Int → [Int]
```

Por exemplo:

```
> factors 15
[1,3,5,15]
```

Usando um guard, podemos definir uma função que mapeia um inteiro positivo para a sua lista de divisores:

```
factors  :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

Por exemplo:

```
> factors 15
[1,3,5,15]
```

Um inteiro positivo é primo se seus únicos fatores são 1 e ele mesmo. Usando factors, podemos definir uma função que decide se um número é primo:

```
prime  :: Int → Bool
prime n = factors n == [1,n]
```

Por exemplo:

```
> prime 15
False

> prime 7
True
```

Usando guards, podemos definir uma função que retorna a lista de todos os primos até um dado limite:

```
primes :: Int → [Int]
```

Por exemplo:

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

Usando guards, podemos definir uma função que retorna a lista de todos os primos até um dado limite:

```
primes :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

Por exemplo:

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

Vamos definir uma função que retorna todos os elementos que possuem uma dada chave:

```
find :: Eq a => a → [(a,b)] → [b]
```

Por exemplo:

```
> find 'b' [('a', 1), ('b', 2), ('c', 3), ('b', 4)]
[2,4]
```

Vamos definir uma função que retorna todos os elementos que possuem uma dada chave:

```
find :: Eq a => a → [(a,b)] → [b]
find k ts = [x | (m,x) ← ts, m == k]
```

Por exemplo:

```
> find 'b' [('a', 1), ('b', 2), ('c', 3), ('b', 4)]
[2,4]
```

A Função Zip

A função zip mapeia duas listas para uma lista de pares dos seus elementos correspondentes.

```
zip :: [a] → [b] → [(a,b)]
```

Por exemplo:

```
> zip [’a’,’b’,’c’] [1,2,3,4]
[(’a’,1),(’b’,2),(’c’,3)]
```

Usando zip, podemos definir uma função que retorna a lista de todos os pares de elementos adjacentes de uma lista:

```
pairs :: [a] → [(a,a)]
```

Por exemplo:

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Usando zip, podemos definir uma função que retorna a lista de todos os pares de elementos adjacentes de uma lista:

```
pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

Por exemplo:

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Podemos definir uma função que decide se os elementos em uma lista estão ordenados usando pairs:

```
sorted :: Ord a ⇒ [a] → Bool
```

Por exemplo:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Podemos definir uma função que decide se os elementos em uma lista estão ordenados usando pairs:

```
sorted :: Ord a ⇒ [a] → Bool
sorted xs = and [x ≤ y | (x,y) ← pairs xs]
```

Por exemplo:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Usando zip, podemos definir uma função que retorna a lista de todas as posições de um valor na lista:

```
positions :: Eq a ⇒ a → [a] → [Int]
```

Por exemplo:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

Usando zip, podemos definir uma função que retorna a lista de todas as posições de um valor na lista:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs = [i | (x',i) ← zip xs [0..n], x == x']
```

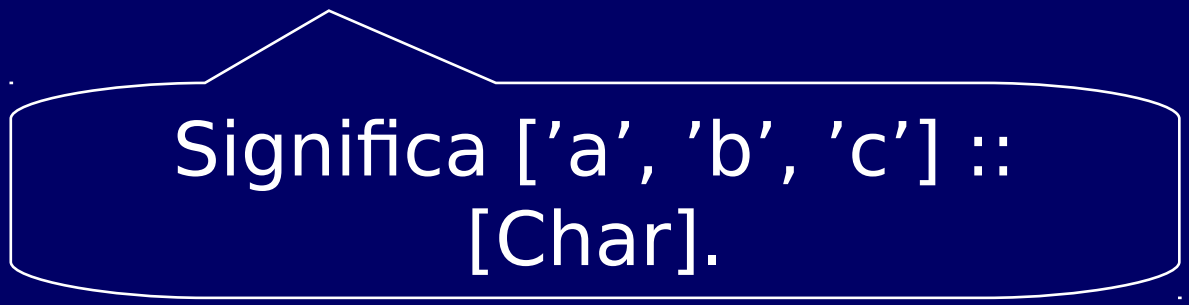
Por exemplo:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

Compreensões de Strings

Uma string é uma sequência de caracteres cercados por aspas duplas. Internamente, strings são representadas como listas de caracteres.

```
"abc" :: String
```



Como strings são tipos especiais de listas, qualquer função polimórfica que opera em listas pode ser aplicada em strings:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

De forma similar, compreensões de listas também pode ser usadas para definir funções em strings, como a contagem de quantas vezes um caractere ocorre na string:

```
count :: Char → String → Int
```

Por exemplo:

```
> count ’s’ "Mississippi"
4
```

De forma similar, compreensões de listas também pode ser usadas para definir funções em strings, como a contagem de quantas vezes um caractere ocorre na string:

```
count :: Char → String → Int
count x xs = length [x' | x' ← xs, x == x']
```

Por exemplo:

```
> count 's' "Mississippi"
4
```

Usando a função isLower :: Char → Bool, vamos definir uma função para retornar a quantidade de minúsculas de uma string:

```
lowers :: String → Int
```

Por exemplo:

```
> lowers "Haskell"
6
```

Usando a função isLower :: Char → Bool, vamos definir uma função para retornar a quantidade de minúsculas de uma string:

```
lowers :: String → Int
lowers xs = length [x | x ← xs, isLower x]
```

Por exemplo:

```
> lowers "Haskell"
6
```