Funções recursivas

- Recursividade
- Recursividade mútua
- Recursividade de cauda
- Vantagens da recursividade

Recursividade

- Recursividade é uma idéia que desempenha um papel central na programação funcional e na ciência da computação em geral.
- Recursividade é o mecanismo de programação no qual uma definição de função ou de outro objeto refere-se ao próprio objeto sendo definido.
- Em programação funcional vamos tratar de funções recursivas

Recursividade

- Assim função recursiva é uma função que é definida em termos de si mesma.

- $R(x) = 10*R(x-1) + 2$ para $x > 1$ e $R(1) = 1$

- São sinônimos: recursividade, recursão, recorrência.

Recursividade

- Recursividade é o mecanismo básico para repetições nas linguagens funcionais.

- Laços (loops) versus Recursão — iterações

Recursividade

.

- Estratégia para a definição recursiva de uma função:
- 1. dividir o problema em problemas menores do mesmo tipo
- 2. resolver os problemas menores (dividindo-os em problemas ainda menores, se necessário)
- 3.combinar as soluções dos problemas menores para formar a solução final

Recursividade

- Ao dividir o problema sucessivamente em problemas menores eventualmente os casos simples são alcançados:
  - não podem ser mais divididos
  - suas soluções são definidas explicitamente
    - exemplo anterior: R(1) = 1 — caso base
    - valor conhecido para determinado argumento

Recursividade

- De modo geral, uma definição de função recursiva é dividida em duas partes:
  - Há um ou mais **casos base** que dizem o que fazer em situações simples, onde não é necessária nenhuma recursão.
    – Nestes casos a resposta pode ser dada de imediato, sem chamar recursivamente a função sendo definida.
    – Isso garante que a recursão eventualmente pode parar.
  - Há um ou mais **casos recursivos** que são mais gerais, e definem a função em termos de uma chamada mais simples a si mesma.

Recursividade

- Como exemplo de função recursiva, considere a seguinte função recorrente.

- $R(x) = 10*R(x-1) + 2$ para $x > 1$ e $R(1) = 1$

- caso recursivo e caso base

Recursividade

.

- $R(x) = 10*R(x-1) + 2$ para $x > 1$ e $R(1) = 1$
- para $x = 2$
- $R(2) = 10*R(1) + 2 = 12$
- para $x = 3$
- $R(3) = 10*R(2) + 2 = 122$
- para $x = 4$
- $R(4) = 10*R(3) + 2 = 1222$

Recursividade

- Um outro exemplo de função recursiva, considere o cálculo do fatorial de um número natural.
- 6! = 6*5!
- 5! = 5*4!
- 4! = 4*3!
- Fatorial de um número inteiro positivo é dado por:
- $n! = n*(n-1)!$ para $n > 0$ caso recursivo
- $0! = 1$ para $n = 0$ caso base

Recursividade

.

- A função que calcula o fatorial de um número natural pode ser definida recursivamente como segue (em Haskell):

```
fatorial :: Integer -> Integer

fatorial n  = if n == 0 then 1 else n * fatorial (n-1)
```

caso base

caso recursivo

Recursividade

.

- A função que calcula o fatorial de um número natural pode ser definida recursivamente como segue (em Haskell):

```
fatorial :: Integer -> Integer

fatorial n
    | n == 0     = 1                          - - caso base
    | n>0        =  n * fatorial(n-1)     - - caso recursivo
```

Recursividade

- Aplicando a função fatorial:
- fatorial 4
  ? fatorial 3 * 4
  ? (fatorial 2 * 3) * 4
  ? ((fatorial 1 * 2) * 3) * 4
  ? (((fatorial 0 * 1) * 2) * 3) * 4 ?
- (((1 * 1) * 2) * 3) * 4
  ? ((1 * 2) * 3) * 4
  ? (2 * 3) * 4
  ?6*4
  ? 24

Recursividade

- Em muitas implementações de linguagens de programação uma **chamada de função** usa um espaço de memória (chamado de quadro, frame ou registro de ativação) em uma área da memória (chamada **pilha** ou stack) onde são armazenadas informações importantes, tais como:
  - argumentos da função
  - variáveis locais
  - variáveis temporárias
  - endereço de retorno da função

Recursividade

- Normalmente para cada nova chamada de função que é realizada, um novo quadro é alocado na pilha.
- Quando a função retorna, o quadro é desalocado.
- De maneira bastante simplificada a sequência de alocação de quadros na pilha de execução na chamada da função fatorial do exemplo anterior é indicada nas figuras seguintes.

Recursividade

fatorial 4	fatorial 4	fatorial 4	fatorial 4	fatorial 4
n = 4	n = 4	n = 4	n = 4	n = 4
$\leadsto$4 * fatorial 3	$\leadsto$4 * fatorial 3	$\leadsto$4 * fatorial 3	$\leadsto$4 * fatorial 3	$\leadsto$4 * fatorial 3
	fatorial 3	fatorial 3	fatorial 3	fatorial 3
	n = 3	n = 3	n = 3	n = 3
	$\leadsto$3 * fatorial 2	$\leadsto$3 * fatorial 2	$\leadsto$3 * fatorial 2	$\leadsto$3 * fatorial 2
		fatorial 2	fatorial 2	fatorial 2
		n = 2	n = 2	n = 2
		$\leadsto$2 * fatorial 1	$\leadsto$2 * fatorial 1	$\leadsto$2 * fatorial 1
			fatorial 1	fatorial 1
			n = 1	n = 1
			$\leadsto$1 * fatorial 0	$\leadsto$1 * fatorial 0
				fatorial 0
				n = 0
				$\leadsto$1

Recursividade

fatorial 4	fatorial 4	fatorial 4	fatorial 4
n = 4	n = 4	n = 4	n = 4
$\leadsto$4 * fatorial 3	$\leadsto$4 * fatorial 3	$\leadsto$4 * fatorial 3	$\leadsto$4 * 6 = 24
fatorial 3	fatorial 3	fatorial 3	
n = 3	n = 3	n = 3	
$\leadsto$3 * fatorial 2	$\leadsto$3 * fatorial 2	$\leadsto$3 * 2	
fatorial 2	fatorial 2		
n = 2	n = 2		
$\leadsto$2 * fatorial 1	$\leadsto$2 * 1 = 2		
fatorial 1			
n = 1			
$\leadsto$1 * 1 = 1			

Recursividade

- Vejamos outro exemplo.
- A função que calcula a potência de dois (isto é, a base é dois) ——> 2^n
- Vamos resolver de maneira recursiva este problema.
- Precisamos encontrar o caso base e o caso recursivo.

Recursividade

- A função que calcula a potência de dois ——> 2^n
- A primeira cláusula estabelece que $2^0 = 1$.
  - Este é o caso base.
- A segunda cláusula estabelece que $2^n = 2 \times 2^{n-1}$, sendo $n > 0$.
  - Este é o caso recursivo.

Recursividade

- A função que calcula a potência de dois (isto é, a base é dois) para números naturais pode ser definida recursivamente como segue (em Haskell):

```
potDois :: Integer -> Integer

potDois n

      | n == 0     = 1                          - - caso base
      | n > 0      = 2 * potDois(n-1)           - - caso recursivo
```

Recursividade

- Aplicando a função potência de dois:
- potDois 4
  ? 2 * potDois 3
  ? 2 * (2 * potDois 2)
  ? 2 * (2 * (2 * potDois 1))
  ? 2 * (2 * (2 * (2 * potDois 0))) ?
- 2 * (2 * (2 * (2 * 1)))
  ? 2 * (2 * (2 * 2))
  ? 2 * (2 * 4)
  ?2*8
  ? 16

Recursividade

.

- Vamos analisar outro exemplo de função recursiva: a sequência de Fibonacci.
- Na seqüência de Fibonacci
- 0,1,1,2,3,5,8,13,...
- os dois primeiros elementos são 0 e 1, e cada elemento subseqüente é dado pela soma dos dois elementos que o precedem na seqüência.

Recursividade

.

- A função a seguir calcula o n-ésimo número de Fibonnaci, para $n \geq 0$:
- função de Fibonacci
- fib(n)=fib(n-1) + fib(n-2) para n > 1 caso recursivo
- fib(0)=0 e fib(1)=1 para n=0 e n=1 dois casos bases

Recursividade

- A função a seguir calcula o n-ésimo número de Fibonnaci, para n ≥ 0 em Haskell:

```
fib :: Int -> Int

fib n

    | n == 0        = 0       - - primeiro caso base
    | n == 1        = 1       - - segundo caso base
    | n>1           = fib(n-2) + fib(n-1)   - - caso recursivo
```

Recursividade

- Nesta definição:
- A primeira e a segunda cláusulas são os casos base.
- A terceira cláusula é o caso recursivo.
- Neste caso temos **recursão múltipla**, pois a função sendo definida é usada mais de uma vez em sua própria definição.

Recursividade

- Aplicando a função de fibonacci:

- fib 5
  ⍰ fib 3 + fib 4
  ⍰ (fib 1 + fib 2) + (fib 2 + fib 3)
  ⍰ (1 + (fib 0 + fib 1)) + ((fib 0 + fib 1) + (fib 1 + fib 2))

  ⍰ (1 + (0 + 1)) + ((0 + 1) + (1 + (fib 0 + fib 1)))

- ⍰ (1 + 1) + (1 + (1 + (0 + 1)))
  ⍰ 2 + (1 + (1 + 1))
  ⍰ 2 + (1 + 2)
  ⍰2+3
  ⍰5

Recursividade mútua

- Recursividade mútua ocorre quando duas ou mais funções são definidas em termos uma da outra.
- Para exemplificar vamos definir as funções par e ímpar usando recursividade mútua
- Se um número é par, o seu antecessor é ímpar e vice-versa
  - se n é par então n-1 é ímpar
  - se n é ímpar então n-1 é par
  - se um número negativo é par (ou ímpar) o seu oposto é ímpar (ou par).

Recursividade mútua

- par :: Int —> Bool
- par n

  | n == 0 = True

  | n > 0 = impar (n-1)

  | otherwise = par (-n)
- impar :: Int —> Bool
- impar n

  | n == 0 = False

  | n > 0 = par (n-1)

  | otherwise = impar (-n)

Recursividade mútua

- Nestas definições observamos que:
- Zero é par, mas não é ímpar.
- Um número positivo é par se seu antecessor é ímpar.
- Um número positivo é ímpar se seu antecessor é par.
- Um número negativo é par (ou ímpar) se o seu oposto for ímpar (ou par).

Recursividade de cauda

- Uma função recursiva apresenta recursividade de cauda se o resultado final da chamada recursiva é o resultado final da própria função.

- Se o resultado da chamada recursiva deve ser processado de alguma maneira para produzir o resultado final, então a função **não** apresenta recursividade de cauda.

Recursividade de cauda

- Por exemplo, a função recursiva a seguir **não** apresenta recursividade de cauda:
- fatorial :: Integer —> Integer
- fatorial n

  | n == 0 = 1

  | n > 0 = n * fatorial(n-1)
- No caso recursivo, o resultado da chamada recursiva fatorial (n-1) é multiplicado por n para produzir o resultado final.

Recursividade de cauda

- Já a função recursiva pdois' a seguir apresenta recursividade de cauda:
- pdois :: Integer —> Integer
- pdois n = pdois' n 1
- pdois' :: Integer —> Integer —> Integer
- pdois' n y

  | n == 0 = y

  | n > 0 = pdois' (n-1) (2*y)   - - só um argumento
- No caso recursivo, o resultado da chamada recursiva pdois' (n-1) (2*y) é o resultado final.

Vantagens de recursividade

- A recursividade permite que:
- Muitas funções possam ser naturalmente definidas em termos de si mesmas
- Propriedades de funções definidas usando recursão podem ser provadas usando indução, uma técnica matemática simples, mas poderosa.
- Trabalhar com repetições sem usar laços

APS 5

- Crie o programa fonte aps5.hs (com todas as funções e variáveis) como codificado nesta aula. Teste todas as funções do programa fonte, carregando no GHCi.
- Observe a necessidade de definir os tipos de dados das variáveis e também das funções.
- Essa APS não precisa ser enviada para o professor, mas deve ser realizada.