

# Revisão de programação funcional em Quint

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC  
Universidade do Estado de Santa Catarina - UDESC

02 de setembro de 2024

# Conteúdo

Introdução

Conjuntos

Maps

Tuplas

Records

Listas

Tipos

# Outline

Introdução

Conjuntos

Maps

Tuplas

Records

Listas

Tipos

# Restrições de Quint e TLA+

- Não há recursão\*
  - \*Existe recursão em TLA+, mas foi adicionado posteriormente. Não suportado pelo Apache.
- Não há laços de repetição (`for`, `while`)
- Não há manipulação de `string`

## Forma dos operadores em Quint

Todos os operadores (exceto os com símbolos, como `+`) podem ser aplicados de duas formas em Quint:

- 1 `operador(arg0, ..., argn)`
- 2 `arg0.operador(arg1, ..., argn)`

Escolha a forma que você acha mais fácil de ler!

# Outline

Introdução

**Conjuntos**

Maps

Tuplas

Records

Listas

Tipos

# Conjuntos!

Conjuntos, ou *Sets*, são a principal estrutura de dados em Quint em TLA+.

- A não ser que a **ordem** dos elementos seja realmente importante, use conjuntos em vez de listas.
- Importante! Isso é um ponto que vou avaliar no trabalho de vocês.

# Conjuntos!

Conjuntos, ou *Sets*, são a principal estrutura de dados em Quint em TLA+.

- A não ser que a **ordem** dos elementos seja realmente importante, use conjuntos em vez de listas.
- Importante! Isso é um ponto que vou avaliar no trabalho de vocês.

O tipo de um conjunto é dado por `Set[<elemento>]`. Ou seja, um conjunto de inteiros tem tipo `Set[int]`.

Criando conjuntos:

```
1 Set(1, 2, 3) // Set(1, 2, 3)
2 1.to(3) // Set(1, 2, 3)
```



## map, seu novo melhor amigo

Em linguagens funcionais, usamos muito a função `map`, que permite a aplicação de uma função a cada elemento de um conjunto.

```
1 Set(1, 2, 3).map(x => x * 2) // Set(2, 4, 6)
```

## map, seu novo melhor amigo

Em linguagens funcionais, usamos muito a função `map`, que permite a aplicação de uma função a cada elemento de um conjunto.

```
1 Set(1, 2, 3).map(x => x * 2) // Set(2, 4, 6)
```

Pode ser usado com lambdas (operadores anônimos), como acima, ou com operadores nomeados:

```
1 pure def quadrado(x: int): int = x * x  
2  
3 Set(1, 2, 3).map(quadrado) // Set(1, 4, 9)
```

## map com operadores de múltiplos argumentos

Dado um conjunto de duplas, podemos aplicar um operador em cada uma das duplas. Mas cuidado! Se o operador espera dois argumentos, temos que fazer o **unpacking** das duplas, utilizando parênteses duplos.

```
1 pure def soma(x: int, y: int): int = x + y
2
3 Set((1, 1), (2, 3)).map(soma)
4 // static analysis error: error: [QNT000] Expected 1
   arguments, got 2
5
6 Set((1, 1), (2, 3)).map(((a, b)) => soma(a, b))
7 // Set(2, 5)
```

## map com operadores que esperam uma dupla

```
1 pure def somaDupla(t: (int, int)): int = t._1 + t._2
2
3 Set((1, 1), (2, 3)).map(somaDupla)
4 // Set(2, 5)
```

## map não resolve tudo!

O `map` só nos ajuda quando queremos um conjunto como retorno.

- Dado um conjunto de números, retorne um conjunto do quadrado desses números.
- Dado um conjunto de pessoas, retorne um conjunto com as idades das pessoas.

# map não resolve tudo!

O `map` só nos ajuda quando queremos um conjunto como retorno.

- Dado um conjunto de números, retorne um conjunto do quadrado desses números.
- Dado um conjunto de pessoas, retorne um conjunto com as idades das pessoas.

`map` não pode ser usado para as seguintes operações:

- Dado um conjunto de números, retorne a soma de todos esses números.
- Dado um conjunto de números, retorne um conjunto apenas com os números pares.

## map não resolve tudo!

O `map` só nos ajuda quando queremos um conjunto como retorno.

- Dado um conjunto de números, retorne um conjunto do quadrado desses números.
- Dado um conjunto de pessoas, retorne um conjunto com as idades das pessoas.

`map` não pode ser usado para as seguintes operações:

- Dado um conjunto de números, retorne a soma de todos esses números.
- Dado um conjunto de números, retorne um conjunto apenas com os números pares.

Lembram quais funções podem ajudar com esses casos?

## filter permite filtrar o conjunto

Exemplo: Dado um conjunto de números, retorne um conjunto apenas com os números pares.

```
1 Set(1, 2, 3, 4).filter(x => x % 2 == 0)
2 // Set(2, 4)
```



# fold permite acumular um valor ao percorrer o conjunto

Argumentos do `fold`

- 1 O conjunto sobre o qual iterar
- 2 Um valor inicial para o acumulador
- 3 Um operador que recebe dois argumentos (o acumulador, e o elemento iterado), e retorna o novo valor para o acumulador

# fold permite acumular um valor ao percorrer o conjunto

## Argumentos do `fold`

- 1 O conjunto sobre o qual iterar
- 2 Um valor inicial para o acumulador
- 3 Um operador que recebe dois argumentos (o acumulador, e o elemento iterado), e retorna o novo valor para o acumulador

Exemplo: Dado um conjunto de números, retorne a soma de todos esses números.

```
1 Set(1, 2, 3, 4).fold(0, (acc, i) => acc + i)
2 // 10
```

# fold permite acumular um valor ao percorrer o conjunto

## Argumentos do `fold`

- 1 O conjunto sobre o qual iterar
- 2 Um valor inicial para o acumulador
- 3 Um operador que recebe dois argumentos (o acumulador, e o elemento iterado), e retorna o novo valor para o acumulador

Exemplo: Dado um conjunto de números, retorne a soma de todos esses números.

```
1 Set(1, 2, 3, 4).fold(0, (acc, i) => acc + i)
2 // 10
```

**Atenção:** Não assumir nada sobre a ordem em que os elementos são iterados.

## Exercício: map e filter com fold

Exercício: Re-escreva nossos exemplos anteriores usando **fold** ao invés de **map** e **filter**:

- 1 Dado um conjunto de números, retorne um conjunto do quadrado desses números.
- 2 Dado um conjunto de números, retorne um conjunto apenas com os números pares.

# Operações de conjuntos

- 1 União: `union`
- 2 Intersecção: `intersect`
- 3 Diferença: `exclude`

# Operadores booleanos para conjuntos

- 1 Pertence,  $\in$ : `in`, `contains`  
`e.in(S)` é equivalente a `S.contains(e)`
- 2 Contido,  $\subseteq$ : `subsetq`
- 3 Para todo,  $\forall$ : `forall`
- 4 Existe,  $\exists$ : `exists`

## Powerset - Conjunto das partes

```
1 Set(1, 2).powerset()  
2 // Set(Set(), Set(1), Set(2), Set(1, 2))
```

Útil quando queremos gerar várias possibilidades para escolher dentre elas.

## flatten, para conjuntos de conjuntos

Um conjunto de conjuntos de elementos do tipo `t` pode ser convertido em um conjunto de elementos do tipo `t` com o operador `flatten`.

```
1 Set(Set(1, 2), Set(1, 3)).flatten()  
2 // Set(1, 2, 3)
```



# Outline

Introdução

Conjuntos

**Maps**

Tuplas

Records

Listas

Tipos

# Maps

Map é a estrutura de dicionário em Quint. Em TLA+, essa mesma estrutura tem nome de função.

# Maps

Map é a estrutura de dicionário em Quint. Em TLA+, essa mesma estrutura tem nome de função.

O tipo de um mapa é dado por `<chave> -> <valor>`. Ou seja, um mapa de inteiros para strings tem tipo `int -> str`.

# Maps

Map é a estrutura de dicionário em Quint. Em TLA+, essa mesma estrutura tem nome de função.

O tipo de um mapa é dado por `<chave> -> <valor>`. Ou seja, um mapa de inteiros para strings tem tipo `int -> str`.

Criando Maps:

```
1 Map(1 -> "a", 2 -> "b")
2 // Map(1 -> "a", 2 -> "b")
3
4 Set((1, "a"), (2, "b")).setToMap()
5 // Map(1 -> "a", 2 -> "b")
6
7 Set(1, 2).mapBy(x => if (x < 2) "a" else "b")
8 // Map(1 -> "a", 2 -> "b")
```

# Chaves e valores

Para obter todas as chaves:

```
1 Map(1 -> "a", 2 -> "b").keys()  
2 // Set(1, 2)
```

E os valores?

```
1 val m = Map(1 -> "a", 2 -> "b")  
2 m.keys().map(k => m.get(k))  
3 // Set("a", "b")
```

## Acessando e atualizando

`set` atualiza um elemento existente, e `put` pode criar um novo par chave-valor.

```
1 val m = Map(1 -> "a", 2 -> "b")
2
3 m.get(1)
4 // "a"
5
6 m.set(1, "c")
7 // Map(1 -> "c", 2 -> "b")
8
9 m.set(3, "c")
10 // runtime error: error: [QNT507] Called 'set' with a
    non-existing key
11
12 m.put(3, "c")
13 // Map(1 -> "a", 2 -> "b", 3 -> "c")
```

## Atualizando com setBy

`setBy` é uma utilidade para quando queremos fazer uma operação sobre um valor existente no mapa.

```
1 val m = Map("a" -> 1, "b" -> 2)
2
3 m.set("a", m.get("a") + 1)
4 // Map("a" -> 2, "b" -> 2)
5
6 m.setBy("a", x => x + 1)
7 // Map("a" -> 2, "b" -> 2)
```

## Criando todos os Maps possíveis

Para criar todos os **Maps** possíveis dado um domínio e um co-domínio, podemos usar o **setOfMaps**:

```
1 Set(1, 2).setOfMaps(Set("a", "b"))  
2 // Set(Map(1 -> "a", 2 -> "a"), Map(1 -> "b", 2 -> "a"  
   ")),  
3 // Map(1 -> "a", 2 -> "b"), Map(1 -> "b", 2 -> "b"  
   ")))
```



# Outline

Introdução

Conjuntos

Maps

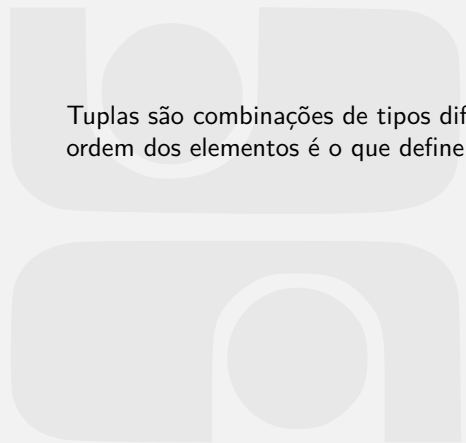
**Tuplas**

Records

Listas

Tipos

# Tuplas



Tuplas são combinações de tipos diferentes em um mesmo valor, onde a ordem dos elementos é o que define o tipo esperado.

# Tuplas

Tuplas são combinações de tipos diferentes em um mesmo valor, onde a ordem dos elementos é o que define o tipo esperado.

O tipo de uma tupla é dado por `(t0, ..., tn)`. Uma tupla com tipo `(int, str, bool)` permite valores como `(1, "a", true)`.

# Tuplas

Tuplas são combinações de tipos diferentes em um mesmo valor, onde a ordem dos elementos é o que define o tipo esperado.

O tipo de uma tupla é dado por `(t0, ..., tn)`. Uma tupla com tipo `(int, str, bool)` permite valores como `(1, "a", true)`.

Existe um único jeito de criar uma tupla:

```
1 (1, "a", true)
```

## Acessando itens

Itens de tuplas são acessados com `._1`, `._2`, `._3`, ...

Não existe `._0`, a contagem inicia do 1.

```
1 val t = (1, "a", true)
2
3 t._1
4 // 1
5
6 t._3
7 // true
```

## Criando todas as tuplas possíveis

Para criar um conjunto com todas as tuplas possíveis com elementos em dados conjuntos, usamos o `tuples`:

```
1 tuples(Set(1, 2), Set("a", "b"))  
2 // Set((1, "a"), (2, "a"), (1, "b"), (2, "b"))  
3  
4 tuples(Set(1), Set("a", "b"), Set(false))  
5 // Set((1, "a", false), (1, "b", false))
```

# Outline

Introdução

Conjuntos

Maps

Tuplas

**Records**

Listas

Tipos

# Records

**Records** são combinações de tipos diferentes em um mesmo valor, onde os elementos são nomeados.



# Records

**Records** são combinações de tipos diferentes em um mesmo valor, onde os elementos são nomeados.

O tipo de um *record* é dado por `{ field0: t0, ..., fieldn: tn }`. Um *record* com tipo `{ nome: str, idade: int }` permite valores como `{ nome: "Gabriela", idade: 26 }`.

## Acessando e atualizando

```
1 val r = { nome: "Gabriela", idade: 26 }
2
3 r.nome
4 // "Gabriela"
5
6 r.with(idade, 27)
7 // { nome: "Gabriela", idade: 27 }
8
9 { ...r, idade: 27 }
10 // { nome: "Gabriela", idade: 27 }
11
12 r
13 // { nome: "Gabriela", idade: 26 }
```

# Outline

Introdução

Conjuntos

Maps

Tuplas

Records

**Listas**

Tipos

# Listas

Listas são como conjuntos, porém com uma ordem definida e, possivelmente, com elementos repetidos. Em TLA+, essa mesma estrutura tem nome de sequência.

# Listas

Listas são como conjuntos, porém com uma ordem definida e, possivelmente, com elementos repetidos. Em TLA+, essa mesma estrutura tem nome de sequência.

O tipo de uma lista é dado por `List[<elemento>]`. Ou seja, uma lista de inteiros tem tipo `List[int]`.

# Listas

Listas são como conjuntos, porém com uma ordem definida e, possivelmente, com elementos repetidos. Em TLA+, essa mesma estrutura tem nome de sequência.

O tipo de uma lista é dado por `List[<elemento>]`. Ou seja, uma lista de inteiros tem tipo `List[int]`.

Criando listas:

```
1 [1, 2, 3]
2 // [1, 2, 3]
3
4 List(1, 2, 3)
5 // [1, 2, 3]
6
7 range(1, 4)
8 // [1, 2, 3]
```

# Acessando

```
1 val l = [1, 2, 3]
2
3 l[1]
4 // 2
5
6 l.head()
7 // 1
8
9 l.tail()
10 // [2, 3]
```

# Atualizando

```
1 val l = [1, 2, 3]
2
3 l.replaceAt(0, 5)
4 // [5, 2, 3]
5
6 l.concat([4, 5])
7 // [1, 2, 3, 4, 5]
8
9 l.append(4)
10 // [1, 2, 3, 4]
11
12 l
13 // [1, 2, 3]
```



# Filtrando listas

`slice` retorna uma nova lista com um intervalo de elementos da lista original.

```
1 [1, 2, 3].slice(0, 1)
2 // [1]
```

`select` é semelhante ao `filter` (de conjuntos).

```
1 [1, 2, 3, 4, 5].select(x => x > 3)
2 // [4, 5]
```

## foldl e foldr

Diferente do `fold` pra conjuntos, a operação de *fold* sobre listas respeita uma ordem específica. `foldl` (*fold left*) vai iterar da esquerda pra direita, enquanto `foldr` (*fold right*) vai iterar da direita pra esquerda.

## foldl e foldr

Diferente do `fold` pra conjuntos, a operação de *fold* sobre listas respeita uma ordem específica. `foldl` (*fold left*) vai iterar da esquerda pra direita, enquanto `foldr` (*fold right*) vai iterar da direita pra esquerda.

Atenção também para a ordem dos argumentos do operador dado como último argumento.

```
1 [1, 2, 3].foldl([], (acc, i) => acc.append(i))  
2 // [1, 2, 3]  
3  
4 [1, 2, 3].foldr([], (i, acc) => acc.append(i))  
5 // [3, 2, 1]
```

## Use índices para fazer um map

O operador `map` não funciona pra listas. Conseguimos reproduzir essa funcionalidade usando o operador `indices`, que retorna o índices de uma lista (isso é, 0 até  $length(l) - 1$ ).

```
1 val l = [1, 2, 3]
2 def f(x) = x + 1
3
4 l.indices().map(i => f(l[i]))
5 // Set(2, 3, 4)
```

Perceba que o resultado aqui é um conjunto. Para que o resultado seja uma lista, temos que usar `foldl` ou `foldr`.

# Outline

Introdução

Conjuntos

Maps

Tuplas

Records

Listas

**Tipos**

## Definindo tipos (*aliases*)

Nomes de tipos devem sempre iniciar com letra maiúscula.

```
1 type Idade = int
2
3 val a: Idade = 1
```

# Tipos soma

```
1 type Período = Manhã | Tarde | Noite
2
3 type EstadoLogin = Logado(str) | Deslogado
4
5 type Opcional[a] = Algum(a) | Nenhum
```

# Recursos

- Cheatsheet Quint
- Documentação dos builtins
- Spells - bibliotecas auxiliares
  - PS: Quer contribuir pra opensource? Esse é um ótimo local pra começar
- Manual do Quint



# Exercícios

- 1 Escreva um operador que recebe um conjunto de inteiros e retorna o maior valor.
- 2 Dado um conjunto de `records` do tipo `{ nome: str, idade: int }`, escreva um operador que recebe esse conjunto e retorna a média de idade.
- 3 Defina um valor que contenha todos os conjuntos possíveis com valores inteiros de 1 a 10, que contenham o número 5 ou o 6.
- 4 Escreva um operador que recebe uma lista e retorna o reverso dela.
- 5 Dado um conjunto de `records` do tipo `{ nome: str, idade: int }`, escreva um operador que recebe esse conjunto e retorna um mapa de nome pra idade.