

# Introdução ao TLA+

Aula para disciplina de Métodos Formais

Gabriela Moreira

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

08 de abril de 2024

# Conteúdo



Quint -> TLA+

# Outline



Quint -> TLA+

# Quint -> TLA+

Já aprendemos Quint, então vamos ver TLA+ pensando nas equivalências com Quint.

- O próprio Manual do Quint trás essas comparações entra TLA+ e Quint

# Tipos

TLA+ não tem tipos!



# Tipos

TLA+ não tem tipos!

- No TLC, erros de tipo serão detectados em runtime
  - Se seu modelo tiver `1 + "bla"` no sétimo estado da execução, o TLC só vai perceber o problema quando chegar nesse estado em sua exploração

# Tipos

TLA+ não tem tipos!

- No TLC, erros de tipo serão detectados em runtime
  - Se seu modelo tiver `1 + "bla"` no sétimo estado da execução, o TLC só vai perceber o problema quando chegar nesse estado em sua exploração
- No Apalache, é preciso traduzir o modelo para fórmulas SMT, que precisam ser tipadas
  - TLA+ para o Apalache é tipado
  - A linguagem em si não é tipada, mas o Apalache espera que os tipos sejam anotados nos comentários

## VARIABLES

```
\* @type: Int -> Int;
clock,
\* @type: Int -> (Int -> Int);
req,
\* @type: Int -> Set(Int);
ack,
```

- Mais informações no Manual do Apalache
  - Não vamos nos aprofundar nisso na disciplina!

# TLA+ REPL

- O TLA+ tem uma REPL que só funciona para expressões constantes
  - Não podemos usar ela para definir variáveis e avaliar transições
- Só está disponível na pre-release do TLA+, então
  - No VSCode, precisamos da extensão versão **Nightly**
  - No `.jar`, precisamos da versão 1.8.0



# TLA+ REPL

- O TLA+ tem uma REPL que só funciona para expressões constantes
  - Não podemos usar ela para definir variáveis e avaliar transições
- Só está disponível na pre-release do TLA+, então
  - No VSCode, precisamos da extensão versão **Nightly**
  - No `.jar`, precisamos da versão 1.8.0

Atenção para como faremos pra rodar ela nos computadores da UDESC:

- ① No VSCode, baixar a extensão: **TLA+ Nightly**
- ② Aperte F1 e escolha: TLA+: Run REPL in Terminal.

# TLA+ REPL em qualquer terminal

Opção 1 (com ou sem `sudo`, somente UNIX):

- <https://github.com/pmer/tla-bin>
- Instalar:

```
git clone https://github.com/pmer/tla-bin.git
cd tla-bin
sh download_or_update_tla.sh --nightly
sh install.sh ~/.local
# ou, se tiver sudo:
sudo install.sh
```

- Executar:

```
cd ~/.local/bin
./tlarepl
```

# TLA+ REPL em qualquer terminal II

Opção 2 (sem `sudo`):

① Baixar o `tla2tools.jar` versão 1.8.0. Duas opções:

- Do GitHub: <https://github.com/tlaplus/tlaplus/releases/download/v1.8.0/tla2tools.jar>
- Ou, se você já instalou a extensão do VSCode, esse arquivo já existe em `~/.vscode/extensions/alygin.vscode-tlaplus-nightly-<versao>/tools`

② Executar o `jar`:

```
java -cp tla2tools.jar tlc2.REPL
```

# TLA+ REPL em qualquer terminal II

Opção 2 (sem `sudo`):

- 1 Baixar o `tla2tools.jar` versão 1.8.0. Duas opções:
  - Do GitHub: <https://github.com/tlaplus/tlaplus/releases/download/v1.8.0/tla2tools.jar>
  - Ou, se você já instalou a extensão do VSCode, esse arquivo já existe em `~/.vscode/extensions/alygin.vscode-tlaplus-nightly-<versao>/tools/tla2tools.jar`
- 2 Executar o `jar`:  
`java -cp tla2tools.jar tlc2.REPL`

Opção 3 (com `sudo`):

- Seguir as instruções em <https://lamport.azurewebsites.net/tla/standalone-tools.html>
- Executar com `java tlc2.REPL`

# Constantes e variáveis

Em Quint:

```
const MY_CONST: int
```

```
var x: str
```

```
var y: bool
```

Em TLA+:

```
CONSTANT MY_CONST
```

```
VARIABLES x, y
```

Temos as palavras-chave: **CONSTANT**, **CONSTANTS**, **VARIABLE** e **VARIABLES**.

# Instanciando módulos

Lembram nos semáforos, quando tínhamos a constante **SEMAFOROS**, e instanciávamos o módulo com:

```
module semaforos_3 {  
  import semaforos(SEMAFOROS=Set(0, 1, 2)).*  
}
```

Em TLA+, usaríamos o **INSTANCE**:

```
INSTANCE semaforos WITH SEMAFOROS <- {0, 1 ,2}
```

# Instanciando módulos

Lembram nos semáforos, quando tínhamos a constante **SEMAFOROS**, e instanciávamos o módulo com:

```
module semaforos_3 {  
  import semaforos(SEMAFOROS=Set(0, 1, 2)).*  
}
```

Em TLA+, usaríamos o **INSTANCE**:

```
INSTANCE semaforos WITH SEMAFOROS <- {0, 1 ,2}
```

Inclusive, em TLA+ podemos atribuir **variáveis** nas instâncias também, o que não é permitido em Quint.

# Instanciando módulos

Lembram nos semáforos, quando tínhamos a constante **SEMAFOROS**, e instanciávamos o módulo com:

```
module semaforos_3 {  
  import semaforos(SEMAFOROS=Set(0, 1, 2)).*  
}
```

Em TLA+, usaríamos o **INSTANCE**:

```
INSTANCE semaforos WITH SEMAFOROS <- {0, 1 ,2}
```

Inclusive, em TLA+ podemos atribuir **variáveis** nas instâncias também, o que não é permitido em Quint.

PS: Constantes e Instâncias são um tanto complicadas. A utilização delas nos trabalhos da disciplina é totalmente opcional.



# Imports

Em Quint, temos os imports

```
import meu_modulo.*  
import meu_modulo.minha_definicao  
import meu_modulo as M
```

Em TLA+

```
EXTENDS meu_modulo
```

# Imports

Em Quint, temos os imports

```
import meu_modulo.*  
import meu_modulo.minha_definicao  
import meu_modulo as M
```

Em TLA+

```
EXTENDS meu_modulo
```

Inclusive, os interiores não são *built-in* em TLA+. Temos que importar o módulo de inteiros com

```
EXTENDS Integers
```

# Literais

- `false` em Quint é `FALSE` em TLA+
- `true` em Quint é `TRUE` em TLA+
- inteiros e strings são a mesma coisa
  - Divisão de inteiros é feita com `\div`

# Lambdas (Operadores Anônimos)

Em Quint, temos lambdas como o a seguir. Contudo (por hora), lambdas só podem ser usados como argumentos pra outros operadores, como para o `map` e `fold`:

```
my_set.map(x => x + 1)
my_set.fold(0, (acc, i) => acc + i)
```

Em TLA+, temos lambdas, de forma geral, como:

```
LAMBDA x: x + 1
LAMBDA x, y: x + y
```

# LET ... IN ...

Em Quint, podemos declarar varios operadores seguidos de uma expressão:

```
pure val a = {  
  pure val b = 1  
  pure val c = b + 1  
  c + 1  
}
```

Em TLA+, fazemos o semelhante com:

```
a == LET b == 1  
      c == b + 1  
      IN c + 1
```

# LET ... IN ...

Em Quint, podemos declarar varios operadores seguidos de uma expressão:

```
pure val a = {
  pure val b = 1
  pure val c = b + 1
  c + 1
}
```

Em TLA+, fazemos o semelhante com:

```
a == LET b == 1
      c == b + 1
      IN c + 1
```

Percebam que usamos duplo = (==) para definições. Para o predicado de igualdade, usamos um único =, diferente de linguagens de programação. Basicamente, o oposto de Quint.

# Conjunção e Disjunção

## Conjunção em Quint:

```
pure val pred = a and b
action conj = all {
  A,
  B,
  C,
}
```

## Conjunção em TLA+:

```
pred == a /\ b
conj ==
  /\ A
  /\ B
  /\ C
```

## Disjunção em Quint:

```
pure val pred = a or b
action disj = any {
  A,
  B,
  C,
}
```

## Disjunção em TLA+:

```
pred == a \/ b
conj ==
  \/ A
  \/ B
  \/ C
```

# Condicional

Em Quint:

```
pure def f(x) = if (x == 0) 10 else 20
```

Em TLA+:

```
f(x) == IF x = 0 THEN 10 ELSE 20
```



# Sets!

Em Quint:

```
Set (1, 2, 3)
```

Em TLA+:

```
{1, 2, 3}
```

# Operadores sobre sets

Existe e para todo:

```
\E x \in S: P  \* S.exists(x => P)
```

```
\A x \in S: P  \* S.forall(x => P)
```

# Operadores sobre sets

Existe e para todo:

```
\E x \in S: P  \* S.exists(x => P)
\A x \in S: P  \* S.forall(x => P)
```

map e filter:

```
{ e: x \in S } \* S.map(x => e)
{ x \in S: P } \* S.filter(x => P)
```

# Operadores sobre sets II

Predicados:

```
e \in S \* e.in(S) ou S.contains(e)
S \union T \* S.union(T)
S \intersect T \* S.intersect(T)
S \ T \* S.exclude(T)
S \subseteq T \* S.subseteq(T)
```

# Operadores sobre sets II

Predicados:

```
e \in S \* e.in(S) ou S.contains(e)
S \union T \* S.union(T)
S \intersect T \* S.intersect(T)
S \ T \* S.exclude(T)
S \subseq T \* S.subseq(T)
```

Outros operadores:

```
SUBSET S \* S.powerset()
UNION S \* S.flatten()
Cardinality(S) \* S.size()
a..b \* a.to(b)
```

PS: Para usar `Cardinality`, precisamos fazer `EXTENDS FiniteSets`

# Não-determinismo

Em Quint:

```
nondet name = my_set.oneOf()  
x' = name
```

Em TLA+, é apenas um *exists* normal:

```
\E name \in my_set: x' = name
```

# Não-determinismo

Em Quint:

```
nondet name = my_set.oneOf()  
x' = name
```

Em TLA+, é apenas um *exists* normal:

```
\E name \in my_set: x' = name
```

Lembrando que o equivalente ao *exists* (`my_set.exists(name => x' = name)`) não é permitido em Quint, pois não podemos usar **ações** como argumentos do *exists*.

# Exercícios Sets

Resolva usando os equivalentes a `map` e `filter` na REPL:

- 1 Dado um conjunto de números, retorne um conjunto do quadrado desses números.

```
LET quadrado(S) == resolucao IN quadrado({1, 2, 3, 4})
```

- 2 Dado um conjunto de números, retorne um conjunto apenas com os números pares.

```
LET pares(S) == resolucao IN pares({1, 2, 3, 4})
```



# Maps

- Chamados funções em TLA+, mas podemos continuar chamando de mapas para não confundir.
- Contudo, a perspectiva aqui é a de funções. Não temos uma boa forma de expressar um mapa que começa vazio e vai crescendo conforme o sistema evolui.
  - Geralmente inicializamos o mapa com as chaves já definidas, e algum valor inicial.
  - Isso é uma boa prática para Quint também!

# Maps - construtor

Em Quint:

```
S.mapBy(x => e)
```

Em TLA+:

```
[ x \in S | -> e ]
```

# Maps - construtor

Em Quint:

```
S.mapBy(x => e)
```

Em TLA+:

```
[ x \in S |-> e ]
```

Por exemplo, criando uma estrutura para guardar o saldo no banco de cada pessoa:

```
[ pessoa \in { "alice", "bob", "charlie" } |-> 0 ]
```

# Maps - construtor

Em Quint:

```
S.mapBy(x => e)
```

Em TLA+:

```
[ x \in S |-> e ]
```

Por exemplo, criando uma estrutura para guardar o saldo no banco de cada pessoa:

```
[ pessoa \in { "alice", "bob", "charlie" } |-> 0 ]
```

Se eu ainda não souber quem são as pessoas, aí sim preciso criar um mapa vazio:

```
[ pessoa \in {} |-> 0 ]
```

# Maps - construtor como em Quint

O equivalente a:

```
Map(k_1 -> v_1, k_2 -> v_2, k_3 -> v_3)
```

seria:

```
[ x \in { a: <<a, b>> \in S } |-> (CHOOSE p \in S: p  
  [1] = x)[2]]
```

# Maps - construtor como em Quint

O equivalente a:

```
Map(k_1 -> v_1, k_2 -> v_2, k_3 -> v_3)
```

seria:

```
[ x \in { a: <<a, b>> \in S } |-> (CHOOSE p \in S: p  
  [1] = x) [2]]
```

O **CHOOSE** é um operador um tanto complicado

- Ele parece não determinístico, mas é completamente determinístico
- Vamos evitar ele por agora. Talvez voltamos nisso no final da disciplina.

# Maps - acesso

Para acessar uma chave  $e$  de um mapa  $f$ :

```
f[e]  \* f.get(e)
```

# Maps - acesso

Para acessar uma chave **e** de um mapa **f**:

```
f[e] \* f.get(e)
```

Um exemplo na REPL.

- PS: A REPL de TLA+ imprime somente os valores de um mapa quando imprime um mapa.

```
(tla+) [ x \in {1, 2} |-> x + 1 ]
\* <<2, 3>>
(tla+) LET m == [ x \in {1, 2} |-> x + 1 ] IN m[1]
\* 2
```



# Operadores sobre Maps

Obtendo o conjunto com as chaves:

```
DOMAIN f \* f.keys()
```

# Operadores sobre Maps

Obtendo o conjunto com as chaves:

```
DOMAIN f \* f.keys()
```

Obtendo todos os mapas possíveis:

```
[ S -> T ] \* setOfMaps(S, T)
```

# Operadores sobre Maps

Obtendo o conjunto com as chaves:

```
DOMAIN f \* f.keys()
```

Obtendo todos os mapas possíveis:

```
[ S -> T ] \* setOfMaps(S, T)
```

Atualizando e adicionando valores:

```
[f EXCEPT ![e1] = e2] \* f.set(e1, e2)
[f EXCEPT ![e1] = e2, ![e3] = e4]
\* f.set(e1, e2).set(e3, e4)
[f EXCEPT ![e1] = @ + y]
\* f.setBy(e1, (old => old + y))
(k :> v) @@ f \* f.put(k, v)
```

# Records

Construtor:

```
[ f_1 |-> e_1, ..., f_n |-> e_n ]  
\* { f_1: e_1, ..., f_n: e_n }
```

# Records

Construtor:

```
[ f_1 |-> e_1, ..., f_n |-> e_n ]  
\* { f_1: e_1, ..., f_n: e_n }
```

Acesso, idêntico ao Quint:

```
r.meu_campo \* r.meu_campo
```

# Records

Construtor:

```
[ f_1 |-> e_1, ..., f_n |-> e_n ]
/* { f_1: e_1, ..., f_n: e_n }
```

Acesso, idêntico ao Quint:

```
r.meu_campo /* r.meu_campo
```

Atualização:

```
[r EXCEPT !.f = e]
/* r.with("f", e) ou { ...r, f: e }
[r EXCEPT !.f1 = e1, !fN = eN] /* N campos
```

# Records II

Obtendo todos os possíveis records:

```
[ f_1: S_1, ..., f_n: S_n ]  
\* tuples(S_1, ..., S_n).map(((a_1, ..., a_n)) => {  
    f_1: a_1, ..., f_n: a_n })
```

# Records II

Obtendo todos os possíveis records:

```
[ f_1: S_1, ..., f_n: S_n ]  
\* tuples(S_1, ..., S_n).map(((a_1, ..., a_n)) => {  
    f_1: a_1, ..., f_n: a_n })
```

Obtendo os nomes dos campos:

```
DOMAIN r \* r.fieldNames()
```



# Listas (ou Sequências)

Construtor:

```
<<e_1, ..., e_n>> \* [ e_1, ..., e_n ]
```

# Listas (ou Sequências)

Construtor:

```
<<e_1, ..., e_n>> \* [ e_1, ..., e_n ]
```

Acesso, sendo que os índices iniciam em 1:

```
s[i] \* 1[i - 1]
```

# Listas (ou Sequências)

Construtor:

```
<<e_1, ..., e_n>> \* [ e_1, ..., e_n ]
```

Acesso, sendo que os índices iniciam em 1:

```
s[i] \* l[i - 1]
```

Atualização em um índice:

```
[ s EXCEPT ![i] = e ] \* l.replaceAt(i - 1, e)
```

# Listas (ou Sequências)

Construtor:

```
<<e_1, ..., e_n>> \* [ e_1, ..., e_n ]
```

Acesso, sendo que os índices iniciam em 1:

```
s[i] \* l[i - 1]
```

Atualização em um índice:

```
[ s EXCEPT ![i] = e ] \* l.replaceAt(i - 1, e)
```

Adicionando elementos:

```
Append(s, e) \* l.append(e)  
l \circ t \* l.concat(t)
```

# Listas II

Outros operadores:

```
Head(l)  \* l.head()
```

```
Tail(l)  \* l.tail()
```

```
Len(s)   \* l.length()
```

```
DOMAIN i \* l.indices().map(i => i - 1)
```

```
SubSeq(lst, start, end) \* l.slice(start - 1, end)
```

```
SelectSeq(s, Test) \* select(l, Test)
```

# Tuplas

Já que não temos tipos em TLA+, tuplas são nada mais do que uma lista.

- elementos podem ter tipos distintos em ambas (heterogenidade).

# Tuplas

Já que não temos tipos em TLA+, tuplas são nada mais do que uma lista.

- elementos podem ter tipos distintos em ambas (heterogenidade).

Construtor:

```
<< e_1, ..., e_n >> \* (e_1, ..., e_n)
```

# Tuplas

Já que não temos tipos em TLA+, tuplas são nada mais do que uma lista.

- elementos podem ter tipos distintos em ambas (heterogenidade).

Construtor:

```
<< e_1, ..., e_n >> \* (e_1, ..., e_n)
```

Acesso:

```
t[1], t[2], ..., t[50] \* t._1, t._2, ..., t._50
```



# Tuplas

Já que não temos tipos em TLA+, tuplas são nada mais do que uma lista.

- elementos podem ter tipos distintos em ambas (heterogenidade).

Construtor:

```
<< e_1, ..., e_n >> \* (e_1, ..., e_n)
```

Acesso:

```
t[1], t[2], ..., t[50] \* t._1, t._2, ..., t._50
```

Obtendo todas as possíveis tuplas:

```
S_1 \X S_2 \X ... \X S_n \* tuples(S_1, S_2, ..., S_n)
```

# Unchanged

TLA+ fornece um operador para o caso especial onde uma variável se mantém com o mesmo valor em uma ação:

# Unchanged

TLA+ fornece um operador para o caso especial onde uma variável se mantém com o mesmo valor em uma ação:

Ao invés de escrevermos:

```
MinhaAcao ==  
  /\ a' = a  
  /\ b' = b
```

Podemos escrever:

```
MinhaAcao ==  
  UNCHANGED << a, b >>
```

# Folds

Não consegui descobrir um jeito de fazer **EXTENDS** pela REPL. Então, vamos usar o VSCode com a funcionalidade de avaliação:

- Selecione o texto de uma **expressão**
- Aperte F1 e selecione TLA+: Evaluate selected expression

# Folds

Não consegui descobrir um jeito de fazer **EXTENDS** pela REPL. Então, vamos usar o VSCode com a funcionalidade de avaliação:

- Selecione o texto de uma **expressão**
- Aperte F1 e selecione TLA+: Evaluate selected expression

Para usar o fold, precisamos de:

- **EXTENDS FiniteSetsExt** para **FoldSet**
- **EXTENDS SequencesExt** para **FoldSeq**

# Folds

Não consegui descobrir um jeito de fazer **EXTENDS** pela REPL. Então, vamos usar o VSCode com a funcionalidade de avaliação:

- Selecione o texto de uma **expressão**
- Aperte F1 e selecione TLA+: Evaluate selected expression

Para usar o fold, precisamos de:

- **EXTENDS FiniteSetsExt** para **FoldSet**
- **EXTENDS SequencesExt** para **FoldSeq**

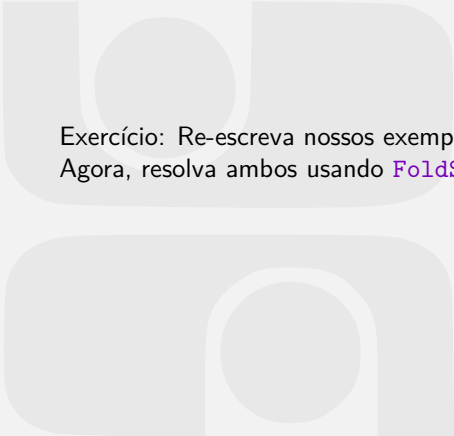
Em Quint:

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

Em TLA+:

```
FoldSet(LAMBDA i, acc : acc + i, 0, S)
```

# Exercícios Fold



Exercício: Re-escreva nossos exemplos anteriores usando **FoldSet**.  
Agora, resolva ambos usando **FoldSet**.

# Exercícios TLA+

- 1 Escreva um operador que recebe um conjunto e retorna a média dos valores.
- 2 Dado um conjunto de `records` como `[ nome |-> "Gabriela", idade |-> 25 ]`, escreva um operador que recebe esse conjunto e retorna a diferença de idade entre o mais velho e o mais novo.
- 3 Defina um valor que contenha todos os conjuntos possíveis com valores inteiros de 1 a 10, com tamanho maior que 2 e menor que 5.
- 4 Escreva um operador que calcule o fatorial de um número. Recursão não é permitida.
- 5 Escreva um operador que recebe uma lista e retorna um mapa onde as chaves são os elementos da lista, e os valores são inteiros representando a quantidade de ocorrências daquele elemento na lista.



# Exercícios TLA+

- 1 Escreva um operador que recebe um conjunto e retorna a média dos valores.
- 2 Dado um conjunto de `records` como `[ nome |-> "Gabriela", idade |-> 25 ]`, escreva um operador que recebe esse conjunto e retorna a diferença de idade entre o mais velho e o mais novo.
- 3 Defina um valor que contenha todos os conjuntos possíveis com valores inteiros de 1 a 10, com tamanho maior que 2 e menor que 5.
- 4 Escreva um operador que calcule o fatorial de um número. Recursão não é permitida.
- 5 Escreva um operador que recebe uma lista e retorna um mapa onde as chaves são os elementos da lista, e os valores são inteiros representando a quantidade de ocorrências daquele elemento na lista.

Dica: você vai precisar dos módulos importados pela expressão:

```
EXTENDS FiniteSets , FiniteSetsExt , Integers ,  
          SequencesExt
```