Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

Trabalho Prático I

1. Objetivo

O objetivo desse trabalho é desenvolver um interpretador para um subconjunto de uma linguagem de programação conhecida. Para isso foi criada *miniGroovy*, uma linguagem de programação de brinquedo baseada em *Groovy* (https://groovy-lang.org). Ela possui suporte a tipos dinâmicos lógicos, inteiros, strings, arranjos e mapas.

2. Contextualização

A seguir é dado um exemplo de utilização da linguagem *miniGroovy*. O recebe números do teclado e separa em positivos e negativos (caso existam).

```
// Preencher um arranjo com numeros lidos do teclado.
def arr = []
def n // recebe null por padrao
while (n != 0) {
  n = read('Entre com um numero inteiro (0 para sair): ') as Integer
  if (n != 0)
    arr += [n]
}

// Separar os numeros em positivos e negativos.
def map = [:]
for (def (i, e) = [0, size(arr)]; i < e; i += 1) {
  def tmp = arr[i]
  if (tmp < 0) {
    if (!map.neg)
      map.neg = []

    map.neg += [tmp]
  } else {
    if (!map.pos)
      map.pos = []

    map.pos += [tmp]
  }
}

// Imprimir os numeros separadamente.
foreach (def k in keys(map)) {
  def name = switch (k) {
    case 'pos' -> 'Positivos'
    case 'neg' -> 'Negativos'
  }

  println(name + ': ' + map[k])
}
```

numbers.mg

Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

A linguagem *miniGroovy* possui escopo global para as variáveis. As variáveis podem ser declaradas previamente através da palavra-reservada **def**, mas sua declaração antes de uso não é obrigatória. A linguagem suporta os seguintes tipos dinâmicos: nulo (**null**), lógico (**false**/**true**), numérico (inteiros), strings (sequência de caracteres entre aspas simples), arranjos e mapas (ambos entre colchetes). Variáveis declaradas mas não inicializadas ou variáveis usadas sem declaração devem ser consideradas como **null**. A linguagem possui conversões implícitas dependendo do operador utilizado. Se não for possível fazer a conversão deve-se gerar um erro tempo de execução. Operadores relacionais funcionam apenas com números, exceto os operadores de igualdade (**==**) e diferença (**!=**), que funcionam também com outros tipos. Já os operadores relacionais de contém (**in**) e não contém (**!in**) funcionam apenas com arranjos (elementos) ou mapas (chaves). Não devem ser feitas conversões implícitas de tipos em expressões condicionais; se os tipos forem diferentes a igualdade obtém falso. Em expressões lógicas, é falso os valores **null**, **false**, inteiro 0, string vazia, arranjos e mapas vazios.

Arranjos e mapas são as estruturas de dados mais importante da linguagem. Elas podem crescer dinamicamente, mas elementos nunca podem ser removidos delas. Arranjos são indexados por índices, começando pelo índice zero, enquanto mapas são indexados por nomes usando a sintaxe de colchetes (por exemplo: **array['1']** ou **mapa['one']**). Os mapas também podem ser acessados através de uma sintaxe alternativa, via nome da propriedade (por exemplo: **mapa.one**). Note que a propriedade tem que ter um nome válido (por exemplo: não é possível usar **mapa.1** ou **mapa.+**).

A linguagem possui comentários de uma linha, onde são ignorados qualquer sequência de caracteres após a sequência //. A linguagem possui as seguintes características:

1) **Comandos:**
   a. `if`: executar comandos se a expressão for verdadeira e executar opcionalmente outros comandos (se houverem) caso contrário.
   b. `while`: repetir comandos enquanto a expressão for verdadeira.
   c. `for`: repetir comandos com declaração, avaliação e incremento, onde cada uma destas partes é opcional.
   d. `foreach`: repetir comandos para cada item de um arranjo.
   e. `print/println`: imprimir na tela sem nova linha (*print*) ou com nova linha (*println*).
   f. **declaração:** variáveis podem ser declaradas com a palavra-reservada **def**, mas sua declaração antes de uso não é obrigatória. Existem duas sintaxes possíveis:

   tipo1: uma ou múltiplas declarações com um valor de inicialização opcional (**null** se não for definido).
   Ex.: **def a = 3** (*a* recebe *3*).
   def b (*b* recebe *null*).
   def c = 4, d, e = false (*c* recebe *4*, *d* recebe **null** e *e* recebe **false**).

tipo2: declaração de múltiplas variáveis entre parênteses do lado esquerdo com uma inicialização obrigatória de uma lista do lado direito. As quantidades de cada lado não precisam ser iguais: se o lado esquerdo for maior as variáveis extras recebem **null**, se o lado direito for maior os valores extras são ignorados:

Ex.: **def (a, b) = [3, false]** (*a* recebe *3*, *b* recebe **false**).
def (d, e, f) = [true, 6] (*d* recebe **true**, *e* recebe *6* e *f* recebe **null**).
def (g, h) = [7, 'M', false] (*g* recebe *7*, *h* recebe 'M').

g. **atribuição**: avaliar o valor de uma expressão do lado direito e opcionalmente atribuir ou atribuir e operar à uma expressão do lado esquerdo (se houver).
Ex.: *x = i + 1* (avaliação com atribuição).
y += x (avaliação com atribuição e operação).

2) Constantes:

a. **null**: valor nulo.
b. **Lógico:** valores **false** e **true**.
c. **Inteiro**: valores formados por números inteiros.
d. **String:** uma sequência de caracteres entre aspas simples.
e. **Arranjo**: sequência de valores entre colchetes separados por vírgula.
f. **Mapa:** sequência de valores separados por vírgula indexados por um índice nominal entre colchetes separados por dois pontos (:).

3) Valores:

a. Variáveis (começam com _, $ ou letras, seguidos de _, $, letras ou dígitos).
b. Literais (inteiros, strings e lógicos).
c. Dinâmicos (arranjos e mapas).

4) Operadores:

a. **Numéricos: +** (adição), **-** (subtração), ***** (multiplicação), **/** (divisão), **%** (resto), ** (exponenciação).
b. **String, Arranjo e Mapa: +** (concatenação).
c. **Lógico: ==** (igualdade), **!=** (diferença), **<** (menor, entre números), **>** (maior, entre números), **<=** (menor igual, entre números), **>=** (maior igual, entre números), **!** (negação), **in** (contém) e **!in** (não contém).
d. **Conector**: **&&** (E) e **||** (OU) (ambos usam curto-circuito).
e. **Pré-operador: -** (inverter sinal).
f. **Conversor de tipo:** operador **as** para conversões com as seguintes regras:
   - para **Boolean**: **null,** lógico **false,** inteiro **0**, arranjo e mapa vazios viram **false**; qualquer outro valor vira **true**.
   - para **Integer**: **null** vira 0; **false** vira 0 e **true** vira 1; inteiro é mantido; string deve ser convertida para inteiro, se falhar vira 0; qualquer outro tipo vira 0.
   - para **String**: todos os tipos, inclusive **null**, são convertidos para seu formato textual.

5) **Comutador:** expressão *switch* que obtém um valor condicionado a uma expressão dada; se não existir default retornar **null**.

Ex.: ***switch*** *(x) {*
null -> 'e nulo'
false -> 'é falso'
default -> 'não é nulo nem falso'
}

6) **Funções:**
   a. **read:** ler uma linha do teclado (sem nova linha \n) como string.
   b. **empty:** verificar se um arranjo, mapa ou string são vazios; para outros tipos deve gerar um erro em tempo de execução.
   c. **size:** contar a quantidade de elementos de arranjos e mapas; gerar um erro em tempo de execução para os outros tipos.
   d. **keys:** obter uma lista com todas as chaves do mapa; para outros tipos deve gerar um erro em tempo de execução.
   e. **values:** obter uma lista com todos os valores do mapa; para outros tipos deve gerar um erro em tempo de execução.

3. Gramática

A gramática da linguagem *miniGroovy* é dada a seguir no formato de Backus-Naur estendida (EBNF):

```
<code>       ::= { <cmd> }
<cmd>        ::= <decl> | <print> | <if> | <while> | <for> | <foreach> | <assign>

<decl>       ::= def ( <decl-type1> | <decl-type2> )
<decl-type1> ::= <name> [ '=' <expr> ] { ',' <name> [ '=' <expr> ] }
<decl-type2> ::= '(' <name> { ',' <name> } ')' '=' <expr>
<print>      ::= (print | println) '(' <expr> ')'
<if>         ::= if '(' <expr> ')' <body> [ else <body> ]
<while>      ::= while '(' <expr> ')' <body>
<for>        ::= for '(' [ ( <decl> | <assign> ) { ',' ( <decl> | <assign> ) } ] ';'
                    [ <expr> ] ';' [ <assign> { ',' <assign> } ] ')' <body>
<foreach>    ::= foreach '(' [ def ] <name> in <expr> ')' <body>
<body>       ::= <cmd> | '{' <code> '}'
<assign>     ::= <expr>  ( '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '**=') <expr>

<expr>       ::= <rel> { ('&&' | '||') <rel> }
<rel>        ::= <cast> [ ('<' | '>' | '<=' | '>=' | '==' | '!=' | in | '!in') <cast> ]
<cast>       ::= <arith> [ as ( Boolean | Integer | String) ]
<arith>      ::= <term> { ('+' | '-') <term> }
<term>       ::= <power> { ('*' | '/' | '%') <power> }
<power>      ::= <factor> { '**' <factor> }
<factor>     ::= [ '!' | '-' ] ( '(' <expr> ')' | <rvalue> )

<lvalue>     ::= <name> { '.' <name> | '[' <expr> ']' }
<rvalue>     ::= <const> | <function> | <switch> | <struct> | <lvalue>
```

```
<const>      ::= null | false | true | <number> | <text>
<function>   ::= (read | empty | size | keys | values) '(' <expr> ')'
<switch>     ::= switch '(' <expr> ')' '{' { case <expr> '->' <expr> } [ default '->' <expr> ] '}'
<struct>     ::= '[' [ ':' | <expr> { ',' <expr> } | <name> ':' <expr> { ',' <name> ':' <expr> } ] ']'
```

4. Instruções

Deve ser desenvolvido um interpretador em linha de comando que recebe um programa-fonte na linguagem *miniGroovy* como argumento e executa os comandos especificados pelo programa. Por exemplo, para o programa *numbers.mg* deve-se produzir uma saída semelhante a:

```
$ ./mgi numbers.mg
Usage: ./mrbi [miniGroovy file]
$ ./mgi numbers.mg
Entre com um numero inteiro (0 para sair): 9
Entre com um numero inteiro (0 para sair): -3
Entre com um numero inteiro (0 para sair): 7
Entre com um numero inteiro (0 para sair): 2
Entre com um numero inteiro (0 para sair): -5
Entre com um numero inteiro (0 para sair): 0

Negativos: [-3, -5]
Positivos: [9, 7, 2]
```

O programa deverá abortar sua execução, em caso de qualquer erro léxico, sintático ou semântico, indicando uma mensagem de erro. As mensagens são padronizadas indicando o número da linha (2 dígitos) onde ocorreram:

Tipo de Erro	**Mensagem**
Léxico	Lexema inválido [*lexema*]
	Fim de arquivo inesperado
Sintático	Lexema não esperado [*lexema*]
	Fim de arquivo inesperado
Semântico	Operação inválida

Exemplo de mensagem de erro:

```
$ ./mgi erro.mg
03: Lexema não esperado [;]
```

5. Avaliação

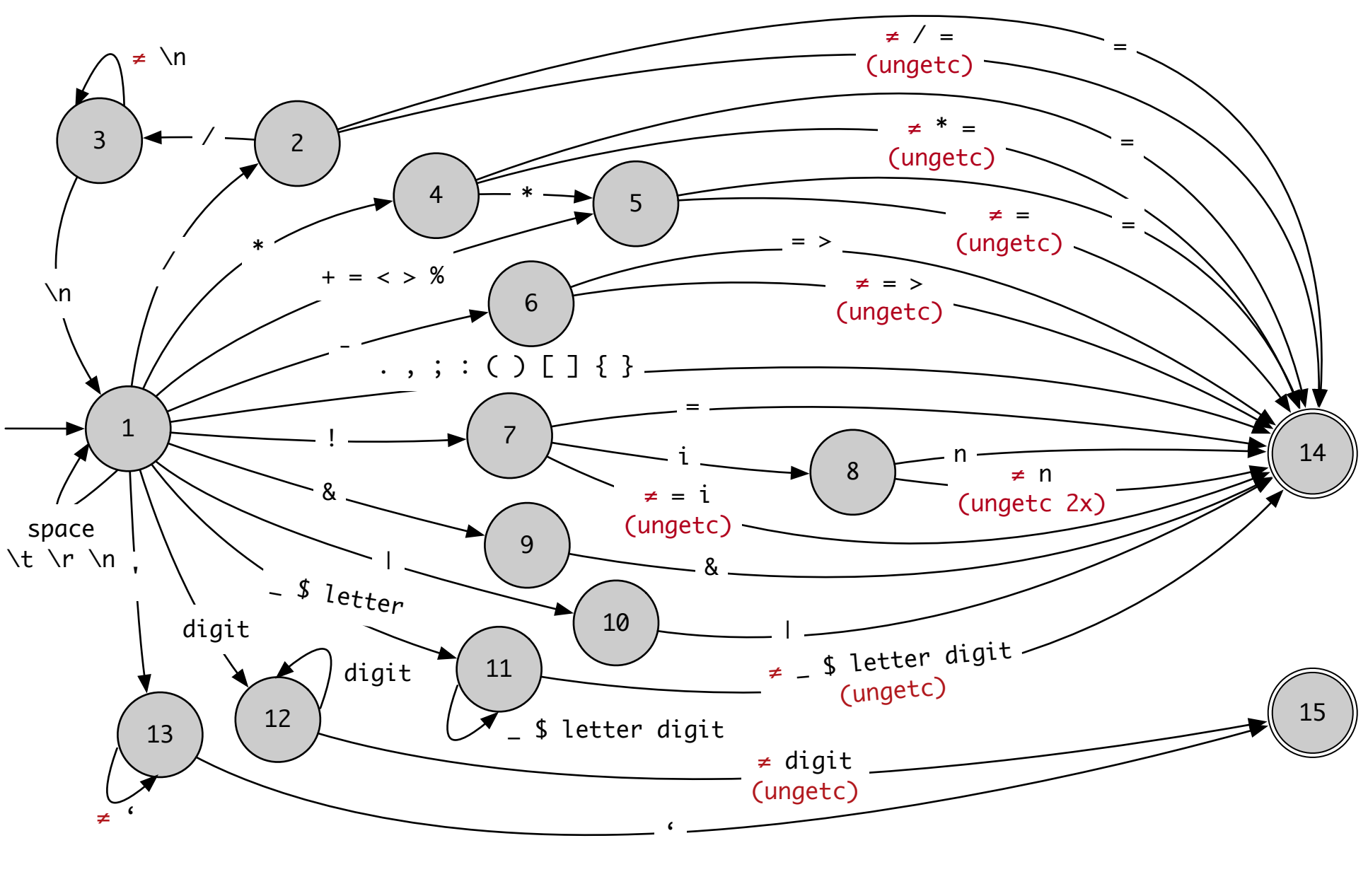
O trabalho deve ser feito em grupo de até dois alunos, sendo esse limite superior estrito. O trabalho será avaliado em 15 pontos, onde essa nota será multiplicada por um fator entre 0.0 e 1.0 para compor a nota de cada aluno individualmente. Esse fator poderá estar condicionado a apresentações presenciais a critério do professor. A avaliação é feita exclusivamente executando casos de testes criados pelo professor. Portanto, códigos que não compilam ou não funcionam serão avaliados com nota **ZERO**.

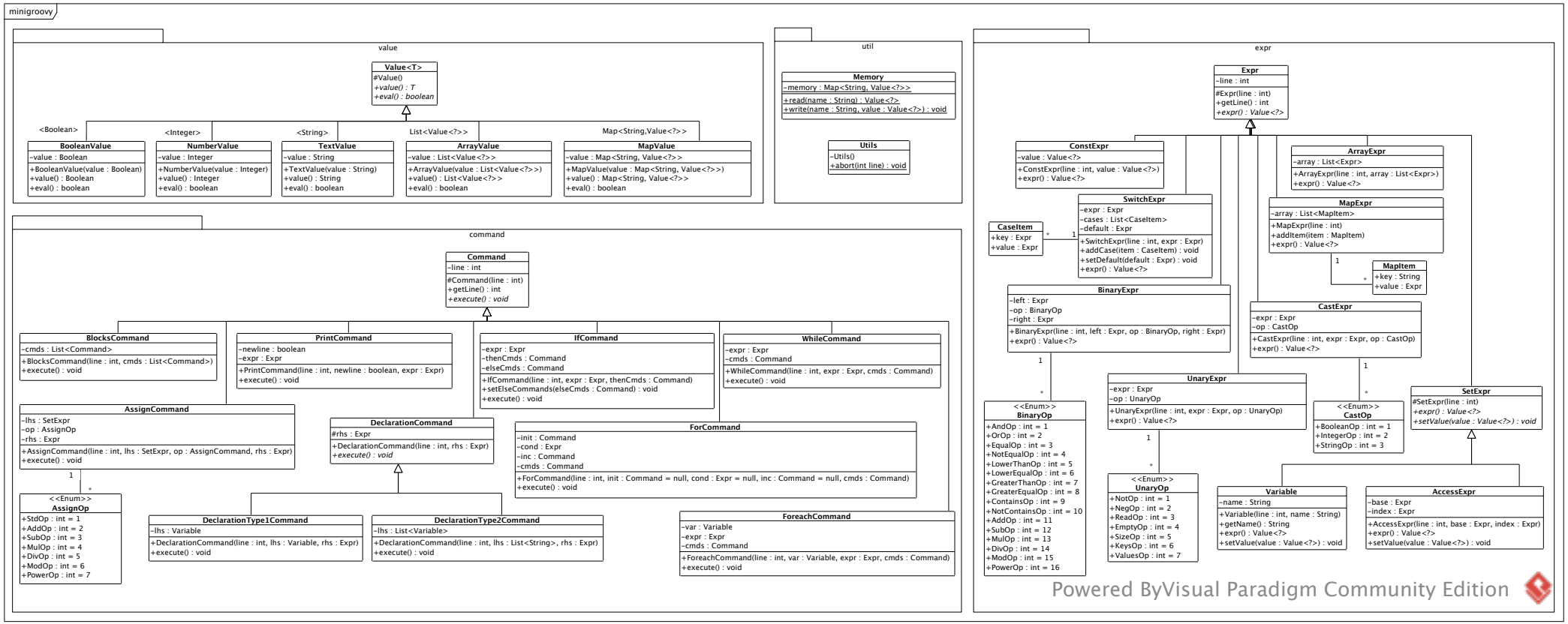
Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

Trabalhos copiados, parcialmente ou integralmente, serão avaliados com nota **ZERO**, sem direito a contestação. Você é responsável pela segurança de seu código, não podendo alegar que outro grupo o utilizou sem o seu consentimento.

6. Submissão

O trabalho deverá ser submetido até as 23:59 do dia 30/05/2022 (segunda-feira) via sistema acadêmico em pasta específica. Não serão aceitos, em hipótese alguma, trabalhos enviados por e-mail ou por quaisquer outras fontes. Para trabalhos feitos em dupla, deve-se criar um arquivo README na raiz do projeto com os nomes dos integrantes da dupla. **A submissão deverá ser feita apenas por um dos integrantes da dupla.**


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
≠ \n
/
\n
*
+ = < > %
-
. , ; : ( ) [ ] { }
!
&
|
space
\t \r \n
'
- $ letter
digit
digit
_ $ letter digit
≠ ‘
‘
≠ / =
(ungetc)
=
≠ * =
(ungetc)
=
≠ =
(ungetc)
=
= >
≠ = >
(ungetc)
=
i
n
≠ = i
(ungetc)
≠ n
(ungetc 2x)
&
|
≠ _ $ letter digit
(ungetc)
≠ digit
(ungetc)


minigroovy
value
Value<T>
#Value()
+value() : T
+eval() : boolean
<Boolean>
<Integer>
<String>
List<Value<?>>
Map<String,Value<?>>
BooleanValue
-value : Boolean
+BooleanValue(value : Boolean)
+value() : Boolean
+eval() : boolean
NumberValue
-value : Integer
+NumberValue(value : Integer)
+value() : Integer
+eval() : boolean
TextValue
-value : String
+TextValue(value : String)
+value() : String
+eval() : boolean
ArrayValue
-value : List<Value<?>>
+ArrayValue(value : List<Value<?>>)
+value() : List<Value<?>>
+eval() : boolean
MapValue
-value : Map<String, Value<?>>
+MapValue(value : Map<String, Value<?>>)
+value() : Map<String, Value<?>>
+eval() : boolean
util
Memory
-memory : Map<String, Value<?>>
+read(name : String) : Value<?>
+write(name : String, value : Value<?>) : void
Utils
-Utils()
+abort(int line) : void
command
Command
-line : int
#Command(line : int)
+getLine() : int
+execute() : void
BlocksCommand
-cmds : List<Command>
+BlocksCommand(line : int, cmds : List<Command>)
+execute() : void
PrintCommand
-newline : boolean
-expr : Expr
+PrintCommand(line : int, newline : boolean, expr : Expr)
+execute() : void
IfCommand
-expr : Expr
-thenCmds : Command
-elseCmds : Command
+IfCommand(line : int, expr : Expr, thenCmds : Command)
+setElseCommands(elseCmds : Command) : void
+execute() : void
WhileCommand
-expr : Expr
-cmds : Command
+WhileCommand(line : int, expr : Expr, cmds : Command)
+execute() : void
AssignCommand
-lhs : SetExpr
-op : AssignOp
-rhs : Expr
+AssignCommand(line : int, lhs : SetExpr, op : AssignCommand, rhs : Expr)
+execute() : void
1
*
<<Enum>>
AssignOp
+StdOp : int = 1
+AddOp : int = 2
+SubOp : int = 3
+MulOp : int = 4
+DivOp : int = 5
+ModOp : int = 6
+PowerOp : int = 7
DeclarationCommand
#rhs : Expr
+DeclarationCommand(line : int, rhs : Expr)
+execute() : void
ForCommand
-init : Command
-cond : Expr
-inc : Command
-cmds : Command
+ForCommand(line : int, init : Command = null, cond : Expr = null, inc : Command = null, cmds : Command)
+execute() : void
DeclarationType1Command
-lhs : Variable
+DeclarationCommand(line : int, lhs : Variable, rhs : Expr)
+execute() : void
DeclarationType2Command
-lhs : List<Variable>
+DeclarationCommand(line : int, lhs : List<String>, rhs : Expr)
+execute() : void
ForeachCommand
-var : Variable
-expr : Expr
-cmds : Command
+ForeachCommand(line : int, var : Variable, expr : Expr, cmds : Command)
+execute() : void
expr
Expr
-line : int
#Expr(line : int)
+getLine() : int
+expr() : Value<?>
ConstExpr
-value : Value<?>
+ConstExpr(line : int, value : Value<?>)
+expr() : Value<?>
ArrayExpr
-array : List<Expr>
+ArrayExpr(line : int, array : List<Expr>)
+expr() : Value<?>
SwitchExpr
-expr : Expr
-cases : List<CaseItem>
-default : Expr
+SwitchExpr(line : int, expr : Expr)
+addCase(item : CaseItem) : void
+setDefault(default : Expr) : void
+expr() : Value<?>
MapExpr
-array : List<MapItem>
+MapExpr(line : int)
+addItem(item : MapItem)
+expr() : Value<?>
CaseItem
+key : Expr
+value : Expr
MapItem
+key : String
+value : Expr
BinaryExpr
-left : Expr
-op : BinaryOp
-right : Expr
+BinaryExpr(line : int, left : Expr, op : BinaryOp, right : Expr)
+expr() : Value<?>
CastExpr
-expr : Expr
-op : CastOp
+CastExpr(line : int, expr : Expr, op : CastOp)
+expr() : Value<?>
<<Enum>>
BinaryOp
+AndOp : int = 1
+OrOp : int = 2
+EqualOp : int = 3
+NotEqualOp : int = 4
+LowerThanOp : int = 5
+LowerEqualOp : int = 6
+GreaterThanOp : int = 7
+GreaterEqualOp : int = 8
+ContainsOp : int = 9
+NotContainsOp : int = 10
+AddOp : int = 11
+SubOp : int = 12
+MulOp : int = 13
+DivOp : int = 14
+ModOp : int = 15
+PowerOp : int = 16
UnaryExpr
-expr : Expr
-op : UnaryOp
+UnaryExpr(line : int, expr : Expr, op : UnaryOp)
+expr() : Value<?>
<<Enum>>
CastOp
+BooleanOp : int = 1
+IntegerOp : int = 2
+StringOp : int = 3
SetExpr
#SetExpr(line : int)
+expr() : Value<?>
+setValue(value : Value<?>) : void
<<Enum>>
UnaryOp
+NotOp : int = 1
+NegOp : int = 2
+ReadOp : int = 3
+EmptyOp : int = 4
+SizeOp : int = 5
+KeysOp : int = 6
+ValuesOp : int = 7
Variable
-name : String
+Variable(line : int, name : String)
+getName() : String
+expr() : Value<?>
+setValue(value : Value<?>) : void
AccessExpr
-base : Expr
-index : Expr
+AccessExpr(line : int, base : Expr, index : Expr)
+expr() : Value<?>
+setValue(value : Value<?>) : void
Powered ByVisual Paradigm Community Edition