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 *miniDart*, uma linguagem de programação de brinquedo baseada em *Dart* (https://dart.dev). Ela possui suporte a tipos dinâmicos lógicos, inteiros, strings, listas e mapas, com suporte a segurança de nulidade.

2. Contextualização

A seguir é dado um exemplo de utilização da linguagem *miniDart*. O código sorteia uma quantidade de jogadas de dados e mostra quantas estavam na primeira metade (dados de 1 a 3), segunda metade (dados de 4 a 6) e todos (dados de 1 a 6).

```
// Ler uma quantidade de jogadas do teclado.
var? tmp = read('Entre com uma quantidade de jogadas de dados: ');
final var runs = toint(tmp ?? '0');
assert(runs > 0, 'Quantidade de jogadas invalida');

// Sortear rodadas aleatorias do dado.
var i = 0, r, freqs = {};
while (i++ < runs) {
  r = random(6) + 1;
  if (freqs[r] == null)
    freqs[r] = 1;
  else
    freqs[r]++;
}

// Sumarizar as frequencias obtidas em listas.
var side;
final var halve1 = [for (side in [1,2,3]) freqs[side] ?? 0];
final var halve2 = [for (side in [4,5,6]) freqs[side] ?? 0];
final var all = [...halve1, ...halve2];

// Imprimir os resultados.
print('Primeira metade: ' + tostr(halve1));
print('Segunda metade: ' + tostr(halve2));
print('Todos: ' + tostr(all));
```

dices.mdart

A linguagem *miniDart* possui escopo global para suas variáveis. Elas precisam ser declaradas através da palavra-reservada **var** antes de seu uso. Variáveis podem ter segurança de nulidade que impede que elas recebam valor **null**. Nesse caso, deve ser atribuído um valor diferente de **null** antes de usá-las. Para desabilitar essa proteção, variáveis devem ser anotadas explicitamente com o símbolo ponto de interrogação (**?**). Nesse caso, assume-se o valor **null** se elas não forem inicializadas. Variáveis também podem ser anotadas como constantes através da palavra-reservada **final**. Essa anotação impede que, uma vez inicializada, seu valor seja modificado. Variáveis só podem ser declaradas uma única vez.

Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

A linguagem suporta os seguintes tipos dinâmicos: nulo (**`null`**), lógico (**`false`**/**`true`**), numérico (inteiros), *strings* (sequência de caracteres entre aspas simples), listas (entre colchetes, onde os elementos são separados por vírgulas) e mapas (entre chaves, onde os elementos são pares de chave/valor separados por dois pontos e os pares são separados por vírgulas). Variáveis declaradas, mas não inicializadas explicitamente, devem receber valor **null**. Listas são indexadas por índices numéricos, começando pelo índice zero, enquanto mapas são indexados por expressões usando a sintaxe de colchetes (por exemplo: **lista[1]** ou **mapa['one']**). Acessos a índices fora da lista e acessos a chaves inexistentes em mapas devem retornar **`null`**. Listas e mapas podem crescer dinamicamente quando novos elementos são adicionados a eles, mas seus elementos nunca podem ser removidos.

A linguagem não possui conversões de tipos implícitas. Todos os operadores esperam operandos de mesmo tipo. Caso deseja-se fazer operações com tipos diferentes, esses devem ser explicitamente convertidos através de funções pré-definidas como **`tobool`**, **`tostr`** e **`toint`**. Expressões em comandos condicionais como **`assert`**, **`if`**, **`while`** e **`do-while`** só funcionam com tipos lógicos. Os operadores relacionais de igualdade (**`==`**) e diferença (**`!=`**) podem ser usados com todos os tipos, já os outros operadores relacionais (**`<`**, **`<=`**, **`>`** e **`>=`**) apenas com inteiros.

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. **declaração:** variáveis podem ser declaradas antes de seu uso através da palavra-reservada **var**. Variáveis podem ser anotadas sem segurança de nulidade, através do símbolo ponto de interrogação (**`?`**), e/ou anotadas como constantes através da palavra-reservada **`final`**.
   b. **`assert`**: verificar se a condição lógica é verdadeira (primeiro argumento), caso contrário exibir a mensagem "`assert: msg`", onde *`msg`* é "`not true`" ou o conteúdo do segundo argumento se houver.
   c. **`print`**: imprimir na tela um valor (argumento) com nova linha.
   d. **`if`**: executar comandos se a expressão for verdadeira e executar opcionalmente outros comandos (se houverem) caso contrário.
   e. **`while`**: enquanto a expressão for verdadeira repetir comandos.
   f. **`do-while`**: repetir comandos enquanto a expressão for verdadeira.
   g. **`for:`** repetir comandos para cada item de uma lista.
   h. **atribuição**: avaliar o valor de uma expressão do lado direito e opcionalmente atribuir à uma expressão do lado esquerdo (se houver).
      Ex.: *x = i + 1* (avaliação com atribuição).
      i++ (avaliação sem atribuição).

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. **Lista**: sequência de elementos entre colchetes separados por vírgula, onde os elementos podem ser expressões, uma coleção, **if** e **for**. Ex.: `[1, 'str', false, x+2]` (expressões)
   `[...ls]` (coleção: inclui todos os elementos da lista `ls`)
   `[if (cond) x]` (se condição então incluir x)
   `[if (cond) x else y]` (se condição então incluir x, senão y)
   `[for (x in l) x]` (incluir os elementos x de uma lista l).

f. **Mapa:** sequência de pares de chave/valor separados por vírgula entre chaves, onde os pares chave/valor são expressões separadas por dois pontos.

3) Valores:

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

4) Operadores:

a. **Numéricos: +** (adição), **-** (subtração), ***** (multiplicação), **/** (divisão), **%** (resto).
b. **String, lista e mapa: +** (concatenação).
c. **Lógico: ==** (igualdade), **!=** (diferença), **<** (menor, entre números), **>** (maior, entre números), **<=** (menor igual, entre números), **>=** (maior igual, entre números).
d. **Conector**: **&&** (E) e **||** (OU) (ambos usam curto-circuito).
e. **Se `null`**: ??
   Ex.: x ?? y (se x for **`null`**, avalia em y, caso contrário em x)
f. **Pré-operador: –** (inverter sinal), **!** (negação), **++** (pré-incremento) e **––** (pré-decremento).
g. **Pós-operador: ++** (pós-incremento) e **––** (pós-decremento).

5) Funções:

a. **`read`:** ler uma linha do teclado (sem nova linha - \n) como string. Se a linha for vazia, deve-se retornar **`null`**.
b. **`random`:** gerar número aleatório entre 0 e argumento (não inclusivo).
c. **`length`:** contar a quantidade de elementos de uma lista; 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.
f. **`tobool`:** converter qualquer tipo para lógico: **`null`,** lógico **`false`,** inteiro **`0`**, string, lista e mapa vazios viram **`false`**; qualquer outro valor vira **`true`**.
g. **`toint`**: converter qualquer tipo para inteiro: **`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`**.
h. **`tostr`**: converter qualquer tipo para seu formato textual, inclusive o valor **`null`**.

3. Gramática

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

```
<code>       ::= { <cmd> }
<cmd>        ::= <decl> | <print> | <assert> | <if> | <while> | <dowhile> | <for> | <assign>
<decl>       ::= [ final ] var [ '?' ] <name> [ '=' <expr> ] { ',' <name> [ '=' <expr> ] } ';'
<print>      ::= print '(' [ <expr> ] ')' ';'
<assert>     ::= assert '(' <expr> [ ',' <expr> ] ')' ';'

<if>         ::= if '(' <expr> ')' <body> [ else <body> ]
<while>      ::= while '(' <expr> ')' <body>
<dowhile>    ::= do <body> while '(' <expr> ')' ';'
<for>        ::= for '(' <name> in <expr> ')' <body>
<body>       ::= <cmd> | '{' <code> '}'

<assign>     ::= [ <expr> '=' ] <expr> ';'

<expr>       ::= <cond> [ '??' <cond> ]
<cond>       ::= <rel> { ( '&&' | '||' ) <rel> }
<rel>        ::= <arith> [ ( '<' | '>' | '<=' | '>=' | '==' | '!=' ) <arith> ]
<arith>      ::= <term> { ( '+' | '-' ) <term> }
<term>       ::= <prefix> { ( '*' | '/' | '%' ) <prefix> }
<prefix>     ::= [ '!' | '-' | '++' | '--' ] <factor>
<factor>     ::= ( '(' <expr> ')' | <rvalue> ) [ '++' | '--' ]

<rvalue>     ::= <const> | <function> | <lvalue> | <list> | <map>

<const>      ::= null | false | true | <number> | <text>
<function>   ::= ( read | random | length | keys | values | tobool | toint | tostr ) '(' <expr> ')'
<lvalue>     ::= <name> { '[' <expr> ']' }

<list>       ::= '[' [ <l-elem> { ',' <l-elem> } ] ']'
<l-elem>     ::= <l-single> | <l-spread> | <l-if> | <l-for>
<l-single>   ::= <expr>
<l-spread>   ::= '...' <expr>
<l-if>       ::= if '(' <expr> ')' <l-elem> [ else <l-elem> ]
<l-for>      ::= for '(' <name> in <expr> ')' <l-elem>

<map>        ::= '{' [ <m-elem> { ',' <m-elem> } ] '}'
<m-elem>     ::= <expr> ':' <expr>
```

4. Instruções

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

Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

```
$ ./mdi
Usage: ./mdi [miniDart file]
$ ./mdi dices.mdart
Entre com uma quantidade de jogadas de dados: 25
Primeira metade: [4, 4, 0]
Segunda metade: [3, 7, 7]
Todos: [4, 4, 0, 3, 7, 7]
```

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:

```
$ ./mdi erro.mdart
03: Lexema não esperado [;]
```

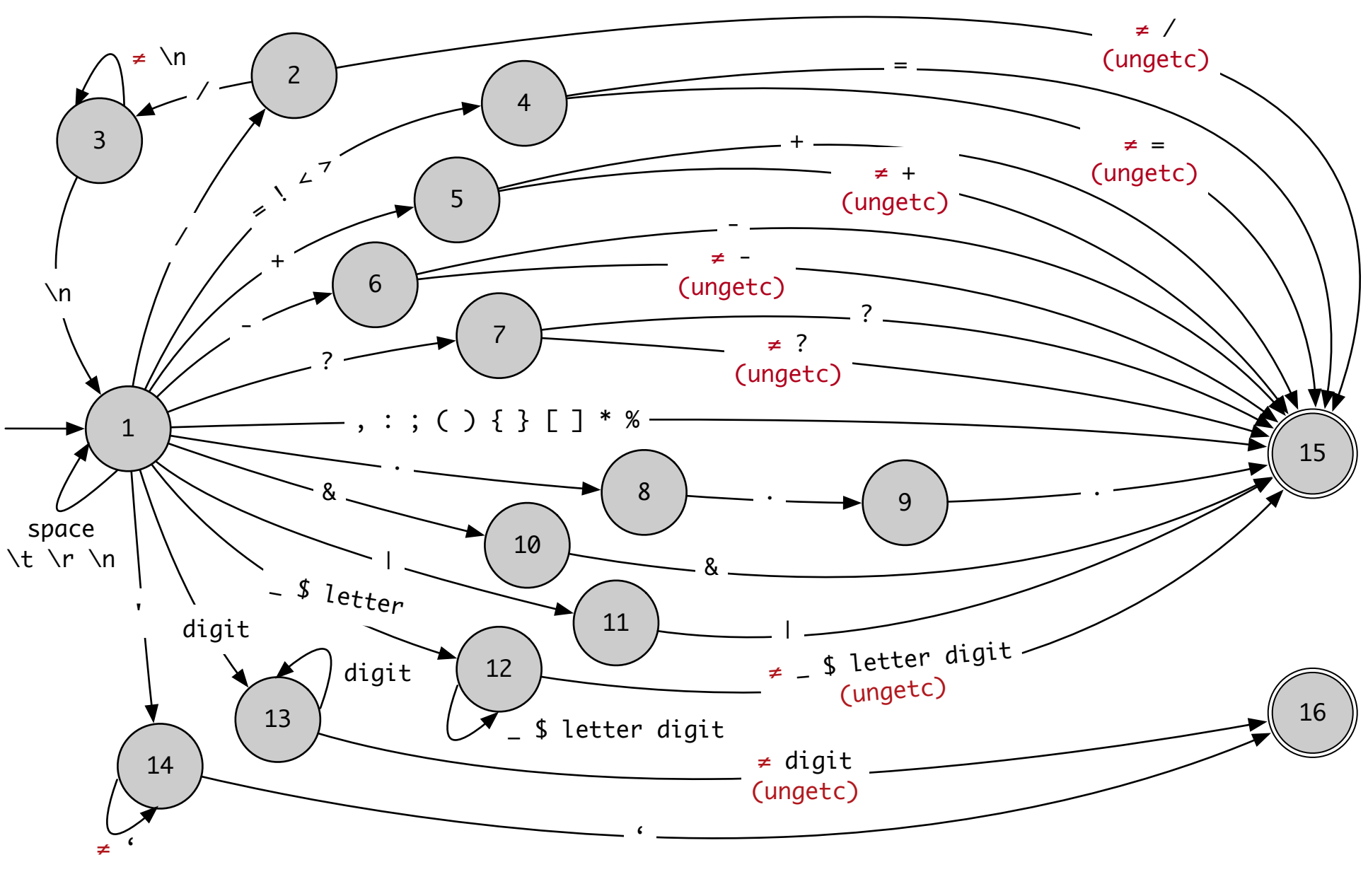
5. Avaliação

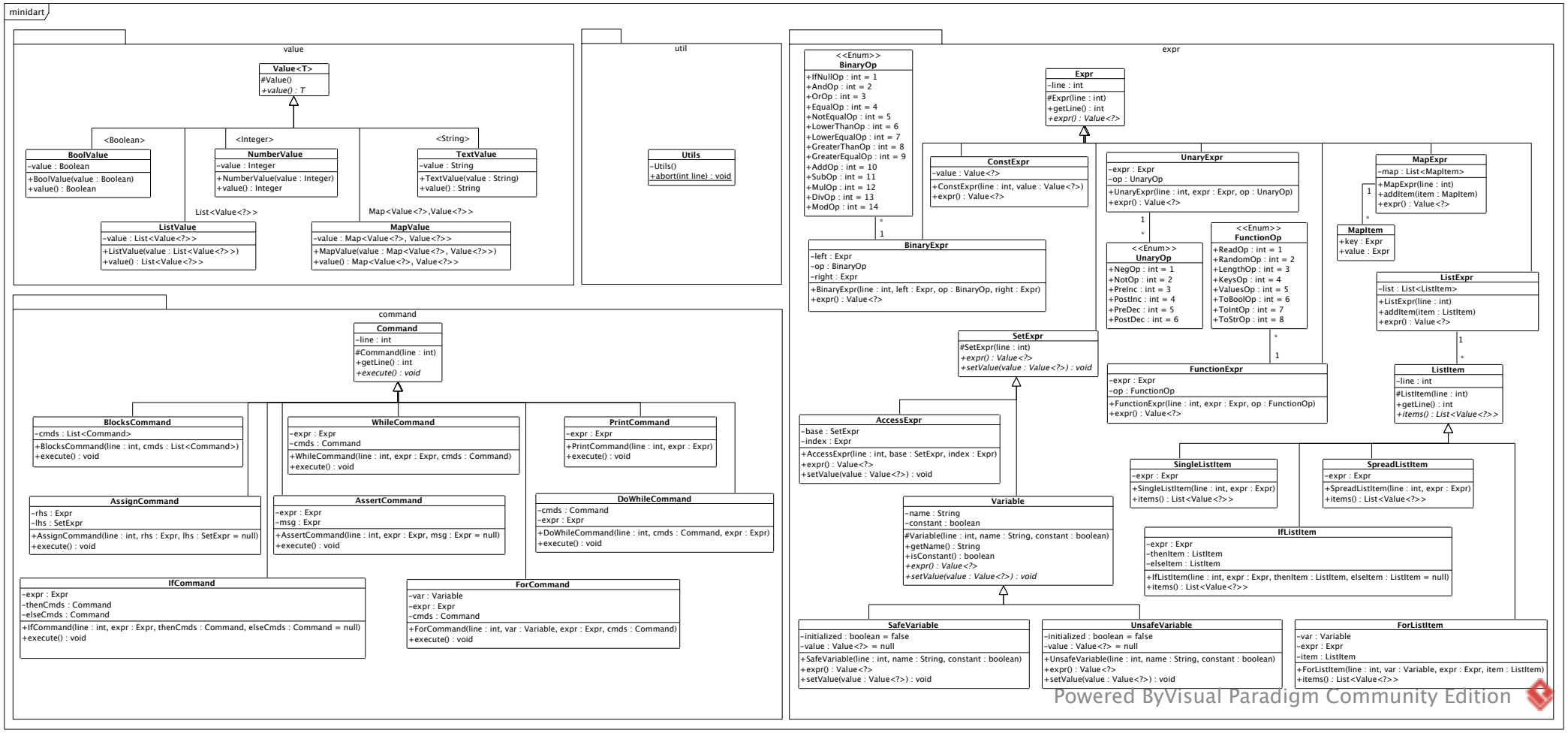
O trabalho deve ser feito em grupo de até dois alunos, sendo esse limite superior estrito. O trabalho será avaliado em 15 pontos, onde essa nota será multiplicada por um fator entre 0.0 e 1.0 para compor a nota de cada aluno individualmente. Esse fator poderá estar condicionado a apresentações presenciais 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**.

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 24/10/2022 (segunda-feira) exclusivamente via sistema acadêmico em pasta específica. **Não serão aceitos, em hipótese alguma, trabalhos enviados por e-mail ou por quaisquer outras fontes**. A submissão deverá incluir todo o código-fonte do programa desenvolvido e arquivos de apoio necessários em um arquivo compactado (zip ou rar). **Não serão consideradas submissões com links para hospedagens externas**. Para trabalhos feitos em dupla, deve-se incluir um arquivo README na raiz do projeto com os nomes dos integrantes da dupla. Nesse caso, a submissão deverá ser feita por apenas um deles.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
space
\t \r \n
\n
≠ \n
/
= ! < >
+
-
?
, : ; ( ) { } [ ] * %
.
&
|
' ‘
digit
_ $ letter
≠ /
(ungetc)
=
≠ =
(ungetc)
+
≠ +
(ungetc)
-
≠ -
(ungetc)
?
≠ ?
(ungetc)
&
|
≠ _ $ letter digit
(ungetc)
_ $ letter digit
≠ digit
(ungetc)
≠ ‘


minidart
value
Value<T>
#Value()
+value() : T
<Boolean>
<Integer>
<String>
BoolValue
-value : Boolean
+BoolValue(value : Boolean)
+value() : Boolean
NumberValue
-value : Integer
+NumberValue(value : Integer)
+value() : Integer
TextValue
-value : String
+TextValue(value : String)
+value() : String
List<Value<?>>
Map<Value<?>,Value<?>>
ListValue
-value : List<Value<?>>
+ListValue(value : List<Value<?>>)
+value() : List<Value<?>>
MapValue
-value : Map<Value<?>, Value<?>>
+MapValue(value : Map<Value<?>, Value<?>>)
+value() : Map<Value<?>, Value<?>>
util
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
WhileCommand
-expr : Expr
-cmds : Command
+WhileCommand(line : int, expr : Expr, cmds : Command)
+execute() : void
PrintCommand
-expr : Expr
+PrintCommand(line : int, expr : Expr)
+execute() : void
AssignCommand
-rhs : Expr
-lhs : SetExpr
+AssignCommand(line : int, rhs : Expr, lhs : SetExpr = null)
+execute() : void
AssertCommand
-expr : Expr
-msg : Expr
+AssertCommand(line : int, expr : Expr, msg : Expr = null)
+execute() : void
DoWhileCommand
-cmds : Command
-expr : Expr
+DoWhileCommand(line : int, cmds : Command, expr : Expr)
+execute() : void
IfCommand
-expr : Expr
-thenCmds : Command
-elseCmds : Command
+IfCommand(line : int, expr : Expr, thenCmds : Command, elseCmds : Command = null)
+execute() : void
ForCommand
-var : Variable
-expr : Expr
-cmds : Command
+ForCommand(line : int, var : Variable, expr : Expr, cmds : Command)
+execute() : void
expr
<<Enum>>
BinaryOp
+IfNullOp : int = 1
+AndOp : int = 2
+OrOp : int = 3
+EqualOp : int = 4
+NotEqualOp : int = 5
+LowerThanOp : int = 6
+LowerEqualOp : int = 7
+GreaterThanOp : int = 8
+GreaterEqualOp : int = 9
+AddOp : int = 10
+SubOp : int = 11
+MulOp : int = 12
+DivOp : int = 13
+ModOp : int = 14
Expr
-line : int
#Expr(line : int)
+getLine() : int
+expr() : Value<?>
ConstExpr
-value : Value<?>
+ConstExpr(line : int, value : Value<?>)
+expr() : Value<?>
UnaryExpr
-expr : Expr
-op : UnaryOp
+UnaryExpr(line : int, expr : Expr, op : UnaryOp)
+expr() : Value<?>
MapExpr
-map : List<MapItem>
+MapExpr(line : int)
+addItem(item : MapItem)
+expr() : Value<?>
MapItem
+key : Expr
+value : Expr
BinaryExpr
-left : Expr
-op : BinaryOp
-right : Expr
+BinaryExpr(line : int, left : Expr, op : BinaryOp, right : Expr)
+expr() : Value<?>
<<Enum>>
UnaryOp
+NegOp : int = 1
+NotOp : int = 2
+PreInc : int = 3
+PostInc : int = 4
+PreDec : int = 5
+PostDec : int = 6
<<Enum>>
FunctionOp
+ReadOp : int = 1
+RandomOp : int = 2
+LengthOp : int = 3
+KeysOp : int = 4
+ValuesOp : int = 5
+ToBoolOp : int = 6
+ToIntOp : int = 7
+ToStrOp : int = 8
ListExpr
-list : List<ListItem>
+ListExpr(line : int)
+addItem(item : ListItem)
+expr() : Value<?>
SetExpr
#SetExpr(line : int)
+expr() : Value<?>
+setValue(value : Value<?>) : void
FunctionExpr
-expr : Expr
-op : FunctionOp
+FunctionExpr(line : int, expr : Expr, op : FunctionOp)
+expr() : Value<?>
ListItem
-line : int
#ListItem(line : int)
+getLine() : int
+items() : List<Value<?>>
AccessExpr
-base : SetExpr
-index : Expr
+AccessExpr(line : int, base : SetExpr, index : Expr)
+expr() : Value<?>
+setValue(value : Value<?>) : void
SingleListItem
-expr : Expr
+SingleListItem(line : int, expr : Expr)
+items() : List<Value<?>>
SpreadListItem
-expr : Expr
+SpreadListItem(line : int, expr : Expr)
+items() : List<Value<?>>
Variable
-name : String
-constant : boolean
#Variable(line : int, name : String, constant : boolean)
+getName() : String
+isConstant() : boolean
+expr() : Value<?>
+setValue(value : Value<?>) : void
IfListItem
-expr : Expr
-thenItem : ListItem
-elseItem : ListItem
+IfListItem(line : int, expr : Expr, thenItem : ListItem, elseItem : ListItem = null)
+items() : List<Value<?>>
SafeVariable
-initialized : boolean = false
-value : Value<?> = null
+SafeVariable(line : int, name : String, constant : boolean)
+expr() : Value<?>
+setValue(value : Value<?>) : void
UnsafeVariable
-initialized : boolean = false
-value : Value<?> = null
+UnsafeVariable(line : int, name : String, constant : boolean)
+expr() : Value<?>
+setValue(value : Value<?>) : void
ForListItem
-var : Variable
-expr : Expr
-item : ListItem
+ForListItem(line : int, var : Variable, expr : Expr, item : ListItem)
+items() : List<Value<?>>
Powered ByVisual Paradigm Community Edition