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 *miniRuby*, uma linguagem de programação de brinquedo baseada em Ruby (https://pt.wikipedia.org/wiki/Ruby_%28linguagem_de_programa%C3%A7%C3%A3o%29). Ela possui suporte a tipos dinâmicos inteiros, strings e arranjos indexados.

2. Contextualização

A seguir é dado um exemplo de utilização da linguagem *miniRuby*. Primeiro, obtém-se o somatório da progressão aritmética de dois números. Depois, deve-se tentar adivinhar esse valor usando uma quantidade limitada de tentativas.

```
# Get a range.
puts 'Escolha dois numeros: ';
min = gets.to_i;
max = gets.to_i;

# Invert them if necessary.
min, max = max, min if min > max;

# Compute the sum.
sum = 0;
for e in min..max do
  sum = sum + e;
end

# Get how many chances?
puts 'Calculei o somatorio da progressão de ' + min.to_s + ' ate ' + max.to_s + '.';
print 'Com quantas chances você quer tentar acertar esse valor? ';
chances = gets.to_i;

# Attempts.
n = 0;
found = 0;
attempts = [];
while n < chances and found != 1 do
  n = n + 1;
  print '  Chance ' + n.to_s + ' de ' + chances.to_s + ': ';

  x = gets.to_i;
  if x === attempts then
    puts 'Voce ja tentou o valor ' + x.to_s;
    n = n - 1;
  else
    if x < sum
      puts 'O valor e maior que ' + x.to_s;
    elsif x > sum
      puts 'O valor e menor que ' + x.to_s;
    else
      puts 'Parabens, voce acertou.';
      found = 1;
    end
```

```
    attempts = attempts + [x];
  end
end

# Print loser.
puts 'Voce nao conseguiu. O valor correto era ' + sum.to_s + '.' if found != 1;

# Print all attempts.
i = 0;
while i < attempts.length do
  if i == 0 then
    print '  Voce tentou os seguintes valores: ';
  else
    print ', ';
  end

  print attempts[i].to_s;
  i = i + 1;
end
puts;
```

guess.mrb

A linguagem possui escopo global para as variáveis. A linguagem suporta três tipos primitivos: **inteiros** (números inteiros de 32 bits), **strings** (sequência de caracteres entre aspas simples) e **arranjos** (elementos separados por vírgula entre colchetes ou sequências usando os operadores **..** ou **...**). Caso uma variável seja usada sem ter seu valor atribuído previamente, deve-se considerar o valor padrão de string vazia (''). A linguagem não possui conversões implícitas, os valores precisão ser convertidos explicitamente pelos programadores através das funções **.to_i** (converte strings para inteiro) ou **.to_s** (converte inteiros ou arranjos para strings). Ou seja, operações aritméticas só funcionam com inteiros, enquanto concatenações (operador **+**) apenas entre strings ou apenas entre arranjos. Operadores relacionais funcionam apenas com inteiros, exceto igualdade (operador ==) e diferença (operador !=) que funcionam com strings. Arranjos são indexados a partir do índice 0 (zero) e podem ser acessados via colchetes (ex.: x[2]). É possível verificar se um elemento pertence a um arranjo (operador ===). A linguagem possui comentários de uma linha onde são ignorados qualquer sequência de caracteres após o símbolo # (hashtag). A linguagem possui as seguintes características:

1) **Comandos:**
   a. `if`: executar comandos se a expressão for verdadeira.
   b. `unless`: executar comandos se a expressão for falsa.
   c. `while`: repetir comandos enquanto a expressão for verdadeira.
   d. `until`: repetir comandos enquanto a expressão for falsa.
   e. `for`: repetir comandos para uma sequência de valores.
   f. `print/puts`: imprimir na tela. Terminam com ponto vírgula (;).
   g. **atribuição**: atribuir os valores das expressões do lado direito às expressões do lado esquerdo. As quantidades de ambos os lados devem bater e terminam com ponto vírgula (;). Ex.: *min, max = max, min;* (trocam-se seus valores).

2) **Constantes:**
   a. **Inteiro**: número formado por dígitos.
   b. **String:** uma sequência de caracteres entre aspas simples.
   c. **Arranjos:** elementos separados por vírgulas entre colchetes, ou gerados pelos operadores .. (dois pontos) ou ... (três pontos).
   d. **Lógico:** operações de comparações que obtém um valor lógico (não podem ser armazenados em variáveis).
3) **Valores:**
   a. Variáveis (começam com _ ou letras, seguidos de _, letras ou dígitos).
   b. Literais (inteiros, strings e arranjos).
4) **Operadores:**
   a. **Inteiro:** **+** (adição), **-** (subtração), ***** (multiplicação), **/** (divisão), **%** (resto), ** (exponenciação).
   b. **String: +** (concatenação)
   c. **Arranjos: +** (concatenação), **..** (faixa inclusivo, ex.: 3..5 gera [3,4,5]) e ... (faixa exclusivo, ex.: 3..5 gera [3, 4]).
   d. **Lógico: ==** (igual entre inteiros, strings), **!=** (diferença ente inteiros ou strings), **<** (menor entre inteiros), **>** (maior entre inteiros), **<=** (menor igual entre inteiros), **>=** (maior igual entre inteiros), **===** (contém inteiro ou string em um arranjo), **not** (negação).
   e. **Conector**: **and** (E lógico), **or** (OU lógico).
5) **Funções:**
   a. **gets:** ler uma linha do teclado, sem o caracter de nova linha (\n).
   b. **rand:** gerar um número inteiro aleatório.
   c. **.length:** obter o tamanho de um arranjo.
   d. **.to_i:** converter uma string para inteiro; se não for possível converter para 0 (zero).
   e. **.to_s:** converter um inteiro ou arranjo para string.

3. Gramática

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

```
<code>     ::= { <cmd> }
<cmd>      ::= <if> | <unless> | <while> | <until> | <for> | <output> | <assign>

<if>       ::= if <boolexpr> [ then ] <code> { elsif <boolexpr> [ then ] <code> } [ else <code> ] end
<unless>   ::= unless <boolexpr> [ then ] <code> [ else <code> ] end

<while>    ::= while <boolexpr> [ do ] <code> end
<until>    ::= until <boolexpr> [ do ] <code> end
<for>      ::= for <id> in <expr> [ do ] <code> end

<output>   ::= ( puts | print ) [ <expr> ] [ <post> ] ';'
<assign>   ::= <access> { ',' <access> } '=' <expr> { ',' <expr> } [ <post> ] ';'
<post>     ::= ( if | unless ) <boolexpr>
```

```
<boolexpr> ::= [ not ] <cmpexpr> [ (and | or) <boolexpr> ]
<cmpexpr>  ::= <expr> ( '==' | '!=' | '<' | '<=' | '>' | '>=' | '===' ) <expr>

<expr>     ::= <arith> [ ( '..' | '...' ) <arith> ]
<arith>    ::= <term> { ('+' | '-') <term> }
<term>     ::= <power> { ('*' | '/' | '%') <power> }
<power>    ::= <factor> { '**' <factor> }
<factor>   ::= [ '+' | '-' ] ( <const> | <input> | <access> ) [ <function> ]

<const>    ::= <integer> | <string> | <array>
<input>    ::= gets | rand
<array>    ::= '[' [ <expr> { ',' <expr> } ] ']'
<access>   ::= ( <id> | '(' <expr> ')' ) [ '[' <expr> ']' ]
<function> ::= '.' ( length | to_i | to_s )
```

4. Instruções

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

```
$ ./mrbi guess.mrb
Usage: ./mrbi [miniRuby file]
$ ./mrbi guess.mrb
Escolha dois numeros:
5
8
Calculei o somatorio da progressão de 5 ate 8.
Com quantas chances você quer tentar acertar esse valor? 3
  Chance 1 de 3: 20
O valor e maior que 20
  Chance 2 de 3: 30
O valor e menor que 30
  Chance 3 de 3: 25
O valor e maior que 25
Voce nao conseguiu. O valor correto era 26.
  Voce tentou os seguintes valores: 20, 30, 25
```

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:

```
$ ./mrbi erro.mrb
03: Lexema não esperado [;]
```

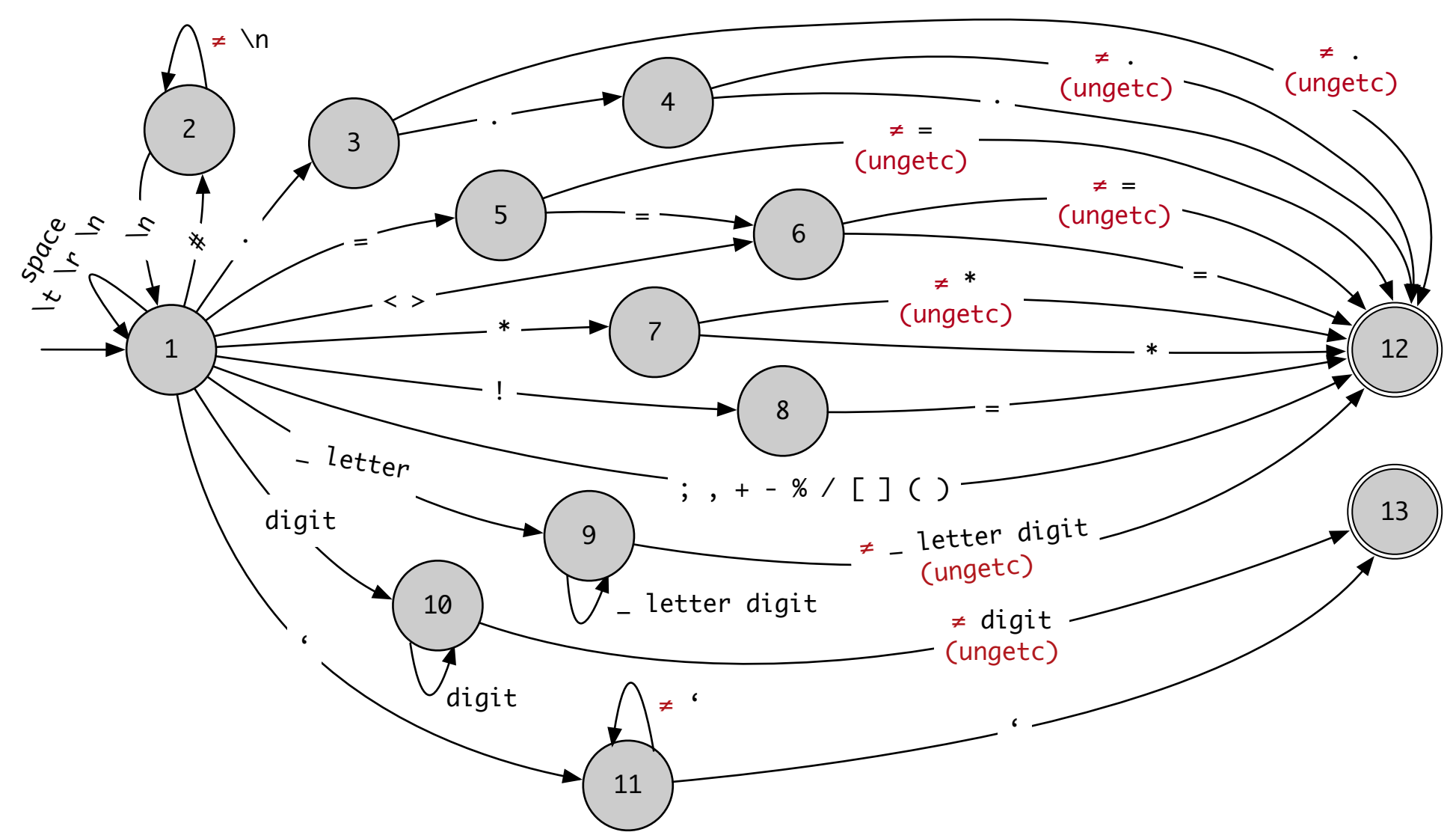
5. Avaliação

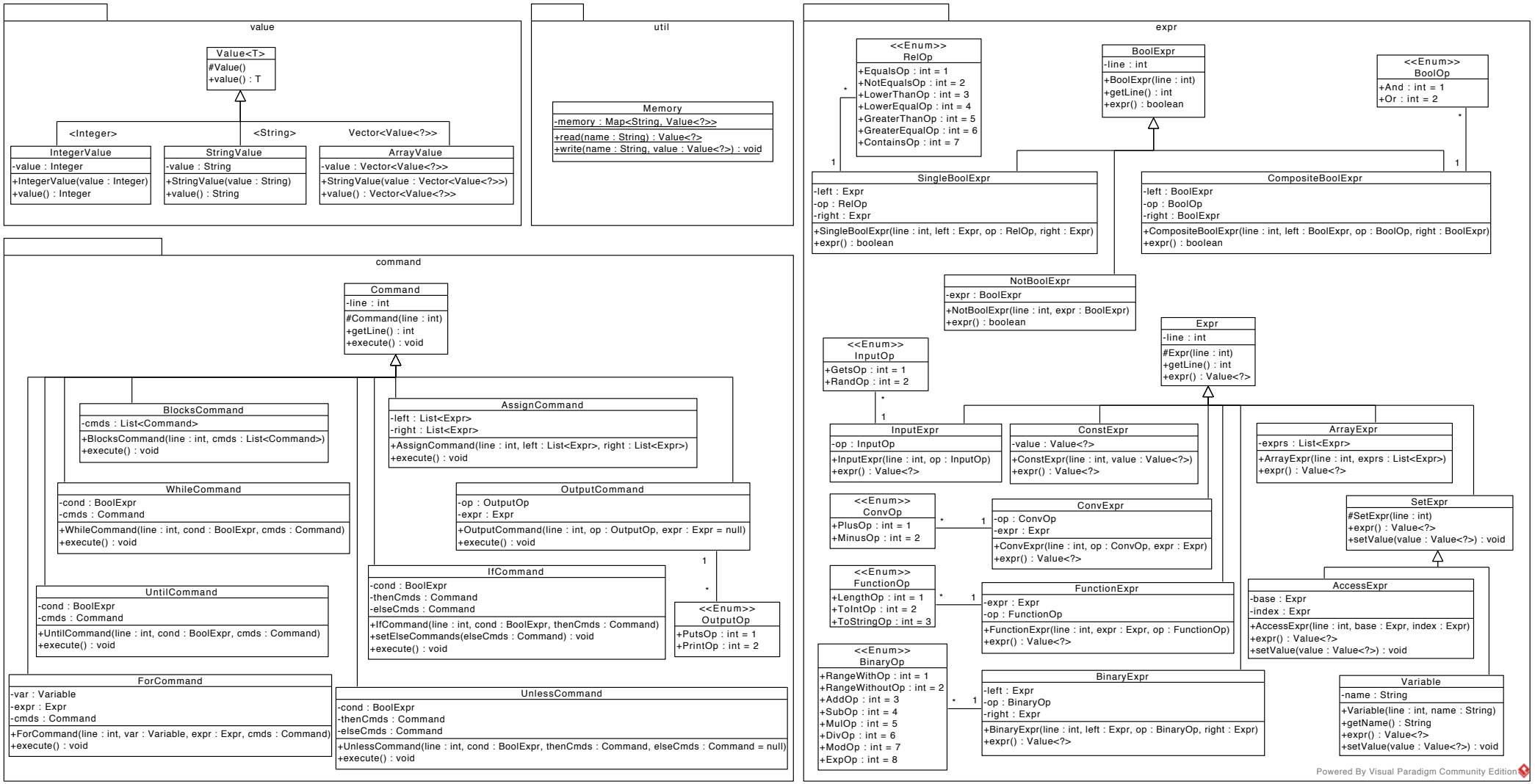
O trabalho deve ser feito em grupo de até dois alunos, sendo esse limite superior estrito. O trabalho será avaliado em 15 pontos, onde essa nota será multiplicada por um fator entre 0.0 e 1.0 para compor a nota de cada aluno individualmente. Esse fator poderá estar condicionado a apresentações presenciais 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 12/07/2021 (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 por apenas um dos integrantes da dupla.**


1
2
3
4
5
6
7
8
9
10
11
12
13
space
\t \r \n
\n
#
.
=
< >
*
!
– letter
digit
‘
; , + - % / [ ] ( )
≠ \n
≠ .
(ungetc)
≠ =
≠ *
≠ – letter digit
≠ digit
≠ ‘
_ letter digit
digit


value
Value<T>
#Value()
+value() : T
<Integer>
<String>
Vector<Value<?>>
IntegerValue
-value : Integer
+IntegerValue(value : Integer)
+value() : Integer
StringValue
-value : String
+StringValue(value : String)
+value() : String
ArrayValue
-value : Vector<Value<?>>
+StringValue(value : Vector<Value<?>>)
+value() : Vector<Value<?>>
util
Memory
-memory : Map<String, Value<?>>
+read(name : String) : Value<?>
+write(name : String, value : Value<?>) : void
expr
<<Enum>>
RelOp
+EqualsOp : int = 1
+NotEqualsOp : int = 2
+LowerThanOp : int = 3
+LowerEqualOp : int = 4
+GreaterThanOp : int = 5
+GreaterEqualOp : int = 6
+ContainsOp : int = 7
BoolExpr
-line : int
+BoolExpr(line : int)
+getLine() : int
+expr() : boolean
<<Enum>>
BoolOp
+And : int = 1
+Or : int = 2
SingleBoolExpr
-left : Expr
-op : RelOp
-right : Expr
+SingleBoolExpr(line : int, left : Expr, op : RelOp, right : Expr)
+expr() : boolean
CompositeBoolExpr
-left : BoolExpr
-op : BoolOp
-right : BoolExpr
+CompositeBoolExpr(line : int, left : BoolExpr, op : BoolOp, right : BoolExpr)
+expr() : boolean
NotBoolExpr
-expr : BoolExpr
+NotBoolExpr(line : int, expr : BoolExpr)
+expr() : boolean
Expr
-line : int
#Expr(line : int)
+getLine() : int
+expr() : Value<?>
<<Enum>>
InputOp
+GetsOp : int = 1
+RandOp : int = 2
InputExpr
-op : InputOp
+InputExpr(line : int, op : InputOp)
+expr() : Value<?>
ConstExpr
-value : Value<?>
+ConstExpr(line : int, value : Value<?>)
+expr() : Value<?>
ArrayExpr
-exprs : List<Expr>
+ArrayExpr(line : int, exprs : List<Expr>)
+expr() : Value<?>
<<Enum>>
ConvOp
+PlusOp : int = 1
+MinusOp : int = 2
ConvExpr
-op : ConvOp
-expr : Expr
+ConvExpr(line : int, op : ConvOp, expr : Expr)
+expr() : Value<?>
SetExpr
#SetExpr(line : int)
+expr() : Value<?>
+setValue(value : Value<?>) : void
<<Enum>>
FunctionOp
+LengthOp : int = 1
+ToIntOp : int = 2
+ToStringOp : int = 3
FunctionExpr
-expr : Expr
-op : FunctionOp
+FunctionExpr(line : int, expr : Expr, op : FunctionOp)
+expr() : Value<?>
AccessExpr
-base : Expr
-index : Expr
+AccessExpr(line : int, base : Expr, index : Expr)
+expr() : Value<?>
+setValue(value : Value<?>) : void
<<Enum>>
BinaryOp
+RangeWithOp : int = 1
+RangeWithoutOp : int = 2
+AddOp : int = 3
+SubOp : int = 4
+MulOp : int = 5
+DivOp : int = 6
+ModOp : int = 7
+ExpOp : int = 8
BinaryExpr
-left : Expr
-op : BinaryOp
-right : Expr
+BinaryExpr(line : int, left : Expr, op : BinaryOp, right : Expr)
+expr() : Value<?>
Variable
-name : String
+Variable(line : int, name : String)
+getName() : String
+expr() : Value<?>
+setValue(value : Value<?>) : void
command
Command
-line : int
#Command(line : int)
+getLine() : int
+execute() : void
BlocksCommand
-cmds : List<Command>
+BlocksCommand(line : int, cmds : List<Command>)
+execute() : void
AssignCommand
-left : List<Expr>
-right : List<Expr>
+AssignCommand(line : int, left : List<Expr>, right : List<Expr>)
+execute() : void
WhileCommand
-cond : BoolExpr
-cmds : Command
+WhileCommand(line : int, cond : BoolExpr, cmds : Command)
+execute() : void
OutputCommand
-op : OutputOp
-expr : Expr
+OutputCommand(line : int, op : OutputOp, expr : Expr = null)
+execute() : void
UntilCommand
-cond : BoolExpr
-cmds : Command
+UntilCommand(line : int, cond : BoolExpr, cmds : Command)
+execute() : void
IfCommand
-cond : BoolExpr
-thenCmds : Command
-elseCmds : Command
+IfCommand(line : int, cond : BoolExpr, thenCmds : Command)
+setElseCommands(elseCmds : Command) : void
+execute() : void
<<Enum>>
OutputOp
+PutsOp : int = 1
+PrintOp : int = 2
ForCommand
-var : Variable
-expr : Expr
-cmds : Command
+ForCommand(line : int, var : Variable, expr : Expr, cmds : Command)
+execute() : void
UnlessCommand
-cond : BoolExpr
-thenCmds : Command
-elseCmds : Command
+UnlessCommand(line : int, cond : BoolExpr, thenCmds : Command, elseCmds : Command = null)
+execute() : void
Powered By Visual Paradigm Community Edition