Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

Trabalho Prático I

1. Objetivo

O objetivo desse trabalho é desenvolver um interpretador para uma nova linguagem de programação que é um subconjunto de PHP, chamada de miniPHP. Ela possui suporte a tipos inteiros, strings e arranjos indexados. Além disso, suporta o conceito de variável-variável e permite iterar sobre arranjos através do comando `foreach`.

2. Contextualização

A seguir é dado um exemplo de utilização da linguagem que usa arranjos indexado por strings, `foreach` para navegação no mesmo e variável-variável:

```
/* Best Megadeth band formation */
$band = array("David Mustaine" => "guitars",
              "David Ellefson" => "bass",
              "Martin Friedman" => "guitars",
              "Nick Menza" => "drums");

echo "Members:\n";
foreach ($band as $member => $instrument) {
    echo "  " . $member . ": " . $instrument . "\n";
    $$instrument++;
}

echo "Stats:\n";
echo "  Guitars: " . $guitars . "\n";
echo "  Bass: " . $bass . "\n";
echo "  Drums: " . $drums . "\n";
```

mega.mphp

A linguagem possui escopo global para as variáveis. Operações aritméticas devem converter strings para inteiros (usar 0 se não for possível). Concatenação deve converter inteiros para string. Variáveis usadas sem ter valor previamente assinalado deve-se usar string vazia (“”) por padrão. A linguagem possui somente comentários de múltiplas linhas onde são ignorados qualquer sequência de caracteres entre /* e */. A linguagem possui as seguintes características:

1) **Comandos:**
   a. `if`: executar comandos baseado em expressões condicionais.
   b. `while`: repetir comandos enquanto a expressão for verdadeira.
   c. `foreach`: iterar sobre um arranjo.
   d. `echo`: imprimir na tela.
2) **Constantes:**
   a. **Inteiro**: sequência de dígitos.
   b. **String:** uma sequência de caracteres entre aspas duplas.
   c. **Lógico:** operações de comparações que obtém um valor lógico (não podem ser armazenados).

3) **Valores:**
   a. Variáveis (ex.: **$x**).
   b. Variável-variável (ex.: **$$x, $$$y**).
   c. Constantes (inteiro e string somente).
   d. **read**: ler do teclado.
4) **Operadores:**
   a. **Inteiro:** **+** (adição), **-** (subtração), ***** (multiplicação), **/** (divisão), **%** (resto inteiro).
   b. **String:** . (concatenação).
   c. **Lógico:** **==** (igual), **!=** (diferença), **<** (menor), **>** (maior), **<=** (menor igual), **>=** (maior igual), **!** (negação).
   d. **Conector**: **and** (E lógico), **or** (OU lógico).
   e. **Atribuição:** **=** (atribuir), **+=** (incrementar e atribuir), **-=** (decrementar e atribuir), ***=** (multiplicar e atribuir), **/=** (dividir e atribuir), **%=** (resto da divisão e atribuir), **.=** (concatenar e atribuir).
   f. **Pré e pós-fixados:** ++ (incremento), -- (decremento)

3. Gramática

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

```
<code>      ::= { <statement> }
<statement> ::= <if> | <while> | <foreach> | <echo> | <assign>

<if>        ::= if '(' <boolexpr> ')' '{' <code> '}'
                { elseif '(' <boolexpr> ')' '{' <code> '}' } [ else '{' <code> '}' ]
<while>     ::= while '(' <boolexpr> ')' '{' <code> '}'
<foreach>   ::= foreach '(' <expr> as <var> [ '=>' <var> ] ')' '{' <code> '}'
<echo>      ::= echo <expr> ';'
<assign>    ::= <value> [ ('=' | '+=' | '-=' | '.=' | '*=' | '/=' | '%=') <expr> ] ';'

<boolexpr>  ::= [ '!' ] <cmpexpr> [ (and | or) <boolexpr> ]
<cmpexpr>   ::= <expr> ('==' | '!=' | '<' | '>' | '<=' | '>=') <expr>

<expr>      ::= <term> { ('+' | '-' | '.') <term> }
<term>      ::= <factor> { ('*' | '/' | '%') <factor> }
<factor>    ::= <number> | <string> | <array> | <read> | <value>

<array>     ::= array '(' [ <expr> '=>' <expr> { ',' <expr> '=>' <expr> } ] ')'
<read>      ::= read <expr>
<value>     ::= [ ('++' | '--') ] <access> | <access> [ ('++' | '--') ]
<access>    ::= ( <varvar> | '(' <expr> ')' ) [ '[' <expr> ']' ]
<varvar>    ::= '$' <varvar> | <var>
```

Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

4. Instruções

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

```
$ ./mphpi
Usage: ./mphpi [miniPHP File]
$ ./mphpi mega.mphp
Members:
  David Mustaine: guitars
  David Ellefson: bass
  Martin Friedman: guitars
  Nick Menza: drums
Stats:
  Guitars: 2
  Bass: 1
  Drums: 1
```

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:

```
$ ./mphpi erro.mphp
03: Lexema não esperado [;]
```

5. Avaliação

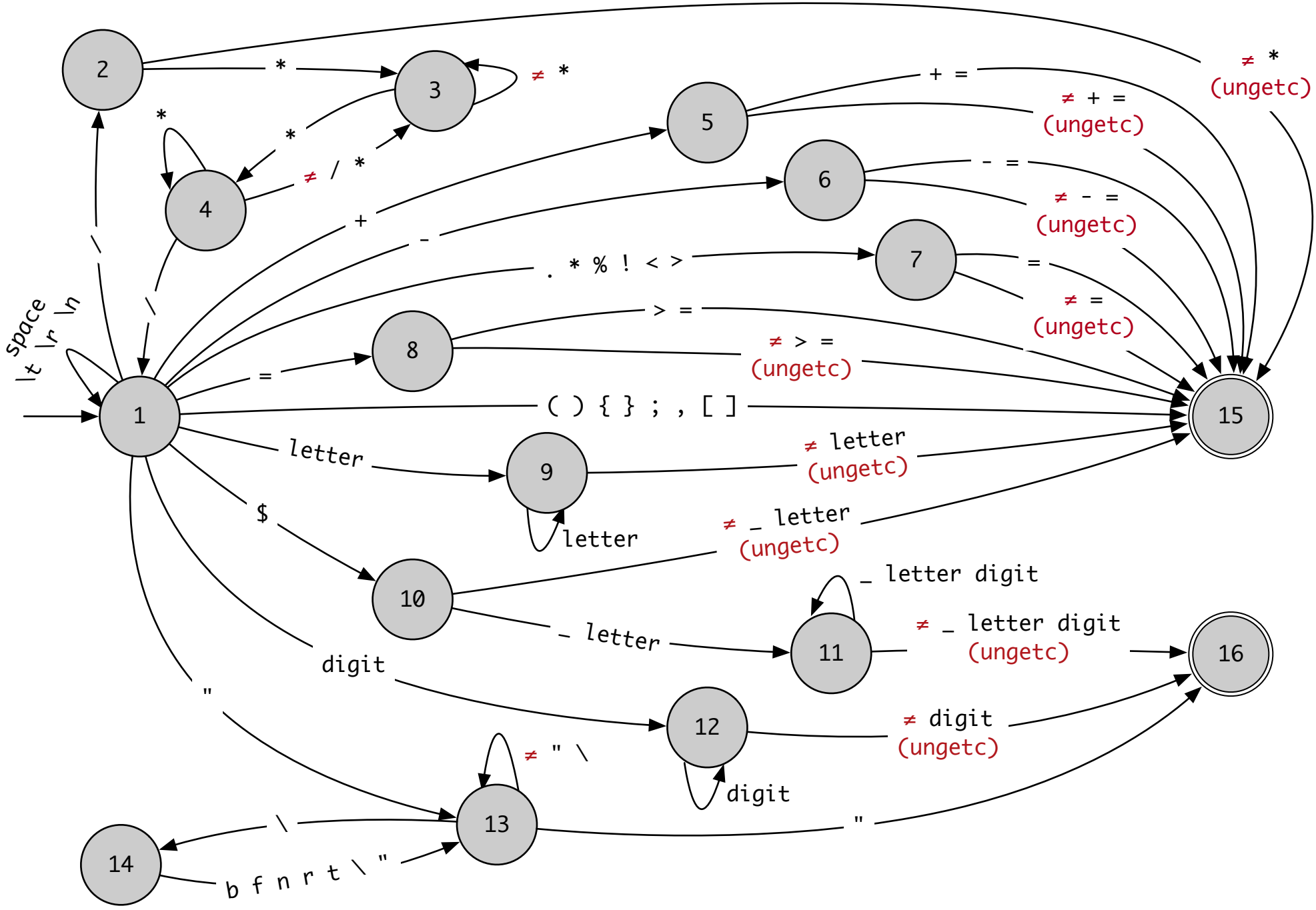
O trabalho deve ser feito em grupo de até dois alunos, sendo esse limite superior estrito. O trabalho será avaliado em 15 pontos, onde essa nota será multiplicada por um fator entre 0.0 e 1.0 para compor a nota de cada aluno individualmente. Esse fator poderá estar condicionado a apresentações presenciais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

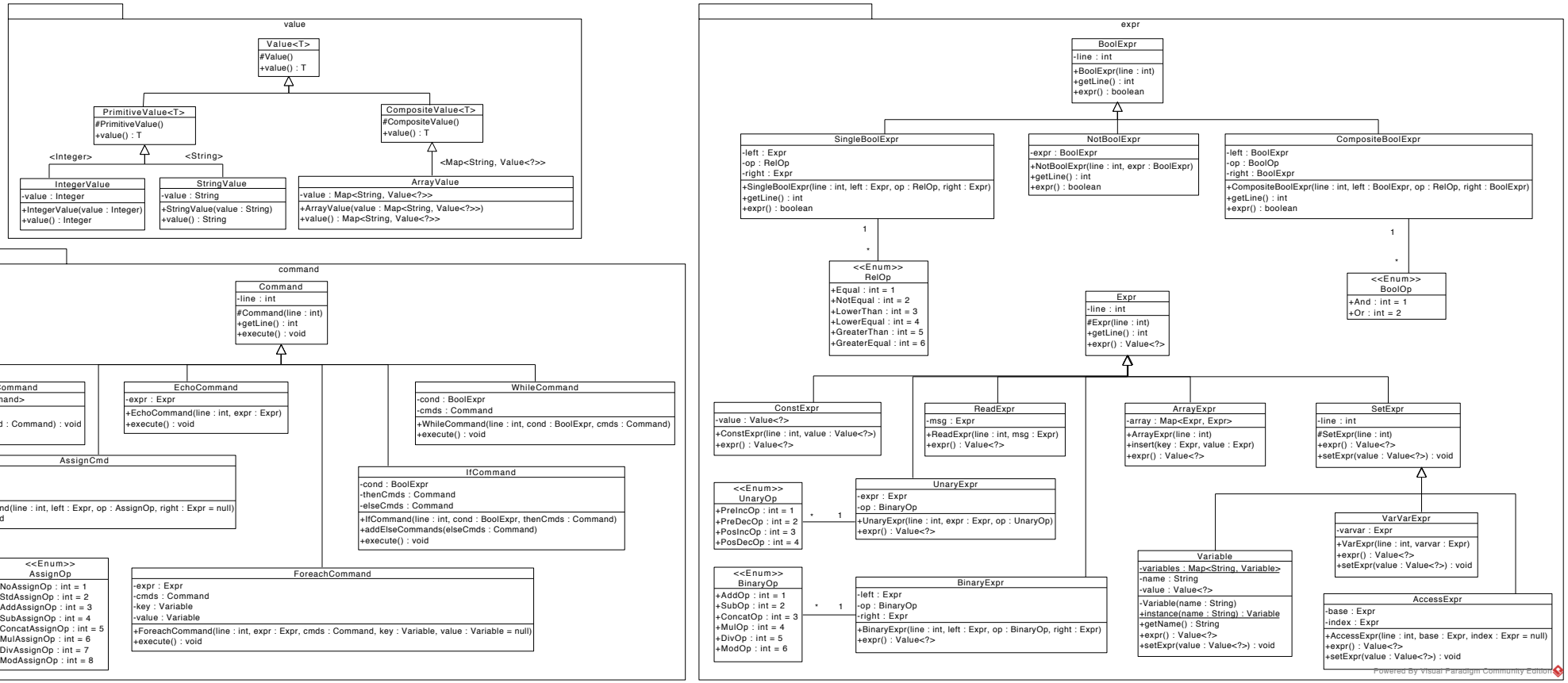
Trabalhos copiados, parcialmente ou integralmente, serão avaliados com nota **ZERO**, sem direito a contestação. Você é responsável pela segurança de seu código, não podendo alegar que outro grupo o utilizou sem o seu consentimento.

Laboratório de Linguagens de Programação
Prof. Andrei Rimsa Álvares

6. Submissão

O trabalho deverá ser submetido até as 23:59 do dia 12/10/2020 (segunda-feira) via sistema acadêmico em pasta específica. Não serão aceitos, em hipótese alguma, trabalhos enviados por e-mail ou por quaisquer outras fontes.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
space
\t \r \n
/
*
≠ *
≠ / *
+
=
≠ + =
(ungetc)
≠ *
(ungetc)
-
≠ - =
(ungetc)
. * % ! < >
=
≠ =
(ungetc)
> =
≠ > =
(ungetc)
( ) { } ; , [ ]
letter
≠ letter
(ungetc)
$
_ letter digit
≠ _ letter
(ungetc)
digit
≠ digit
(ungetc)
"
≠ " \
\
b f n r t \ "


value
Value<T>
#Value()
+value() : T
PrimitiveValue<T>
#PrimitiveValue()
+value() : T
CompositeValue<T>
#CompositeValue()
+value() : T
<Integer>
<String>
<Map<String, Value<?>>
IntegerValue
-value : Integer
+IntegerValue(value : Integer)
+value() : Integer
StringValue
-value : String
+StringValue(value : String)
+value() : String
ArrayValue
-value : Map<String, Value<?>>
+ArrayValue(value : Map<String, Value<?>>)
+value() : Map<String, Value<?>>
command
Command
-line : int
#Command(line : int)
+getLine() : int
+execute() : void
BlocksCommand
-cmds : List<Command>
+BlocksCommand()
+addCommand(cmd : Command) : void
+execute() : void
EchoCommand
-expr : Expr
+EchoCommand(line : int, expr : Expr)
+execute() : void
WhileCommand
-cond : BoolExpr
-cmds : Command
+WhileCommand(line : int, cond : BoolExpr, cmds : Command)
+execute() : void
AssignCmd
-left : Expr
-op : AssignOp
-right : Expr
+AssignCommand(line : int, left : Expr, op : AssignOp, right : Expr = null)
+execute() : void
IfCommand
-cond : BoolExpr
-thenCmds : Command
-elseCmds : Command
+IfCommand(line : int, cond : BoolExpr, thenCmds : Command)
+addElseCommands(elseCmds : Command)
+execute() : void
1
*
<<Enum>>
AssignOp
+NoAssignOp : int = 1
+StdAssignOp : int = 2
+AddAssignOp : int = 3
+SubAssignOp : int = 4
+ConcatAssignOp : int = 5
+MulAssignOp : int = 6
+DivAssignOp : int = 7
+ModAssignOp : int = 8
ForeachCommand
-expr : Expr
-cmds : Command
-key : Variable
-value : Variable
+ForeachCommand(line : int, expr : Expr, cmds : Command, key : Variable, value : Variable = null)
+execute() : void
expr
BoolExpr
-line : int
+BoolExpr(line : int)
+getLine() : int
+expr() : boolean
SingleBoolExpr
-left : Expr
-op : RelOp
-right : Expr
+SingleBoolExpr(line : int, left : Expr, op : RelOp, right : Expr)
+getLine() : int
+expr() : boolean
NotBoolExpr
-expr : BoolExpr
+NotBoolExpr(line : int, expr : BoolExpr)
+getLine() : int
+expr() : boolean
CompositeBoolExpr
-left : BoolExpr
-op : BoolOp
-right : BoolExpr
+CompositeBoolExpr(line : int, left : BoolExpr, op : RelOp, right : BoolExpr)
+getLine() : int
+expr() : boolean
<<Enum>>
RelOp
+Equal : int = 1
+NotEqual : int = 2
+LowerThan : int = 3
+LowerEqual : int = 4
+GreaterThan : int = 5
+GreaterEqual : int = 6
<<Enum>>
BoolOp
+And : int = 1
+Or : int = 2
Expr
-line : int
#Expr(line : int)
+getLine() : int
+expr() : Value<?>
ConstExpr
-value : Value<?>
+ConstExpr(line : int, value : Value<?>)
+expr() : Value<?>
ReadExpr
-msg : Expr
+ReadExpr(line : int, msg : Expr)
+expr() : Value<?>
ArrayExpr
-array : Map<Expr, Expr>
+ArrayExpr(line : int)
+insert(key : Expr, value : Expr)
+expr() : Value<?>
SetExpr
-line : int
#SetExpr(line : int)
+expr() : Value<?>
+setExpr(value : Value<?>) : void
<<Enum>>
UnaryOp
+PreIncOp : int = 1
+PreDecOp : int = 2
+PosIncOp : int = 3
+PosDecOp : int = 4
UnaryExpr
-expr : Expr
-op : BinaryOp
+UnaryExpr(line : int, expr : Expr, op : UnaryOp)
+expr() : Value<?>
<<Enum>>
BinaryOp
+AddOp : int = 1
+SubOp : int = 2
+ConcatOp : int = 3
+MulOp : int = 4
+DivOp : int = 5
+ModOp : int = 6
BinaryExpr
-left : Expr
-op : BinaryOp
-right : Expr
+BinaryExpr(line : int, left : Expr, op : BinaryOp, right : Expr)
+expr() : Value<?>
Variable
-variables : Map<String, Variable>
-name : String
-value : Value<?>
-Variable(name : String)
+instance(name : String) : Variable
+getName() : String
+expr() : Value<?>
+setExpr(value : Value<?>) : void
VarVarExpr
-varvar : Expr
+VarExpr(line : int, varvar : Expr)
+expr() : Value<?>
+setExpr(value : Value<?>) : void
AccessExpr
-base : Expr
-index : Expr
+AccessExpr(line : int, base : Expr, index : Expr = null)
+expr() : Value<?>
+setExpr(value : Value<?>) : void
Powered By Visual Paradigm Community Edition