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 embute ideias de programação orientada a objetos: *SmallOO*. Ela possui quatro (4) tipos que podem ser indexados: inteiros, string, objetos e funções. Objetos possuem propriedades que podem armazenar qualquer um dos tipos disponíveis. Funções possuem retorno e podem ser chamadas com qualquer quantidade de argumentos.

2. Contextualização

A seguir é dado um exemplo de utilização da linguagem que simula um objeto que mantém informações de um arranjo de inteiros:

```
/* iniciar o array com nenhum elemento e tamanho 0. */
array.size = 0;

/* adicionar um novo elemento ao arranjo. */
array.add = function {
  system.set(self, ("element" + self.size), args.arg1);
  array.size = array.size + 1;
}

/* obter um elemento através de seu índice. */
array.get = function {
  return system.get(self, ("element" + args.arg1));
}

/* imprimir o arranjo. */
array.print = function {
  system.print("[ ");
  i = 0;
  while (i < self.size) {
    system.print(system.get(self, ("element" + i)) + " ");
    i = i + 1;
  }
  system.println("]");
}

/* entrar c/ numeros no arranjo */
n = system.input("Entre c/ a qtd de numeros: ");
c = 0;
while (c < n) {
  x = system.random(0, 9);
  array.add(x);
  c = c + 1;
}

/* criar uma copia do arranjo. */
array2 = array;

/* adicionar mais um numero aleatorio a ele. */
y = system.random(0, 9);
array2.add(y);
```

```
/* adicionar o método ordenar no novo arranjo utilizando o metodo bolha. */
array2.sort = function {
  a = 0;
  while (a < self.size) {
    b = a + 1;
    while (b < self.size) {
      x = system.get(self, ("element" + a));
      y = system.get(self, ("element" + b));
      if (y < x) {
        system.set(self, ("element" + a), y);
        system.set(self, ("element" + b), x);
      }
      b = b + 1;
    }
    a = a + 1;
  }
}

/* imprimir o arranjo original */
array.print();

/* ordenar o novo arranjo e depois imprimir. */
array2.sort();
array2.print();
```

array.soo

Um programa em *SmallOO* possui variáveis e propriedades de objetos que conseguem armazenar tipos primitivos (inteiros), outros objetos e funções. Todo tipo é atribuído ou passado por parâmetros por **cópia**. A linguagem possui escopo global para as variáveis. Funções dentro de objetos podem referenciar suas próprias propriedades (atributos ou métodos) através da palavra reservada **self**, e acessar os parâmetros passados na chamada através da palavra reservada **args**. Existe um objeto especial chamado **system** que possui as funções especiais:

- `print(msg)`: imprime `msg` (opcional) na tela, onde `msg` deve ser inteiro ou string.
- `println(msg)`: similar a função `print`, mas com nova linha.
- `read(msg)`: imprime `msg` (opcional) como `print` e retorna um número inteiro ou string obtido do teclado.
- `random(n, m)`: retorna um número aleatório inteiro entre `n` e `m` (incluso), onde `n` e `m` são inteiros e `n` deve ser maior que `m`.
- `get(obj, name)`: obtém o valor da propriedade `name` do objeto `obj`, onde `obj` deve ser um objeto e `name` uma string. Pode retornar qualquer um dos quatro tipos.
- `set(obj, name, value)`: define valor da propriedade `name` do objeto `obj` com o valor `value`, onde `obj` deve ser um objeto, `name` uma string e `value` qualquer um dos quatro tipos.
- `abort(msg)`: imprime `msg` (opcional) como `print` e aborta a execução.
- `type(x)`: retorna o tipo de `x` como uma string.
- `length(str)`: retorna o tamanho da string `str`, zero caso contrário.
- `substring(str, i, f)`: retorna a substring de str a partir de i de tamanho f.

Operações aritméticas só podem ser feitas em inteiros, a não ser o operador + que pode ser usado para concatenar duas strings. A linguagem possui 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. `atribuição`: guardar um valor em uma variável.
   d. `chamada:` chamada de funções.
2) **Valores:**
   a. **Inteiro**: sequência de dígitos.
   b. **String:** uma sequência de caracteres entre aspas duplas.
   c. **Objetos:** armazenam propriedades que podem ser atributos (inteiros, strings e outros objetos) e métodos (funções) .
   d. **Funções:** procedimentos que podem tomar opcionalmente qualquer quantidade de argumentos e retornam um valor. Caso não seja especificado deve-se retornar 0.
   e. **Lógico:** operações de comparações que obtém um valor lógico (não podem ser armazenados).
3) **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**: & (E lógico), | (OU lógico).

3. Gramática

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

```
<code>      ::= { <statement> }
<statement> ::= <if> | <while> | <cmd>
<if>        ::= if '(' <boolexpr> ')' '{' <code> '}' [ else '{' <code> '}' ]
<while>     ::= while '(' <boolexpr> ')' '{' <code> '}'
<cmd>       ::= <access> ( <assign> | <call> ) ';'
<access>    ::= <var> { '.' <name> }
<assign>    ::= '=' <rhs>
<call>      ::= '(' [ <rhs> { ',' <rhs> } ] ')'

<boolexpr>  ::= [ '!' ] <cmpexpr> [ ('&' | '|') <boolexpr> ]
<cmpexpr>   ::= <expr> <relop> <expr>
<relop>     ::= '==' | '!=' | '<' | '>' | '<=' | '>='
```

```
<rhs>       ::= <function> | <expr>
<function>  ::= function '{' <code> [ return <rhs> ';' ] '}'

<expr>      ::= <term> { ('+' | '-') <term> }
<term>      ::= <factor> { ('*' | '/' | '%') <factor> }
<factor>    ::= <number> | <string> | <access> [ <call> ] | '(' <expr> ')'

<var>       ::= system | self | args | <name>
```

4. Instruções

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

```
$ ./sooi
Usage: ./sooi [SmallOO File]
$ ./sooi array.soo
Entre c/ a qtd de numeros: 5
[ 1, 9, 3, 7, 2 ]
[ 1, 2, 3, 6, 7, 9 ]
```

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:

```
$ ./sooi erro.msh
03: Lexema não esperado [;]
```

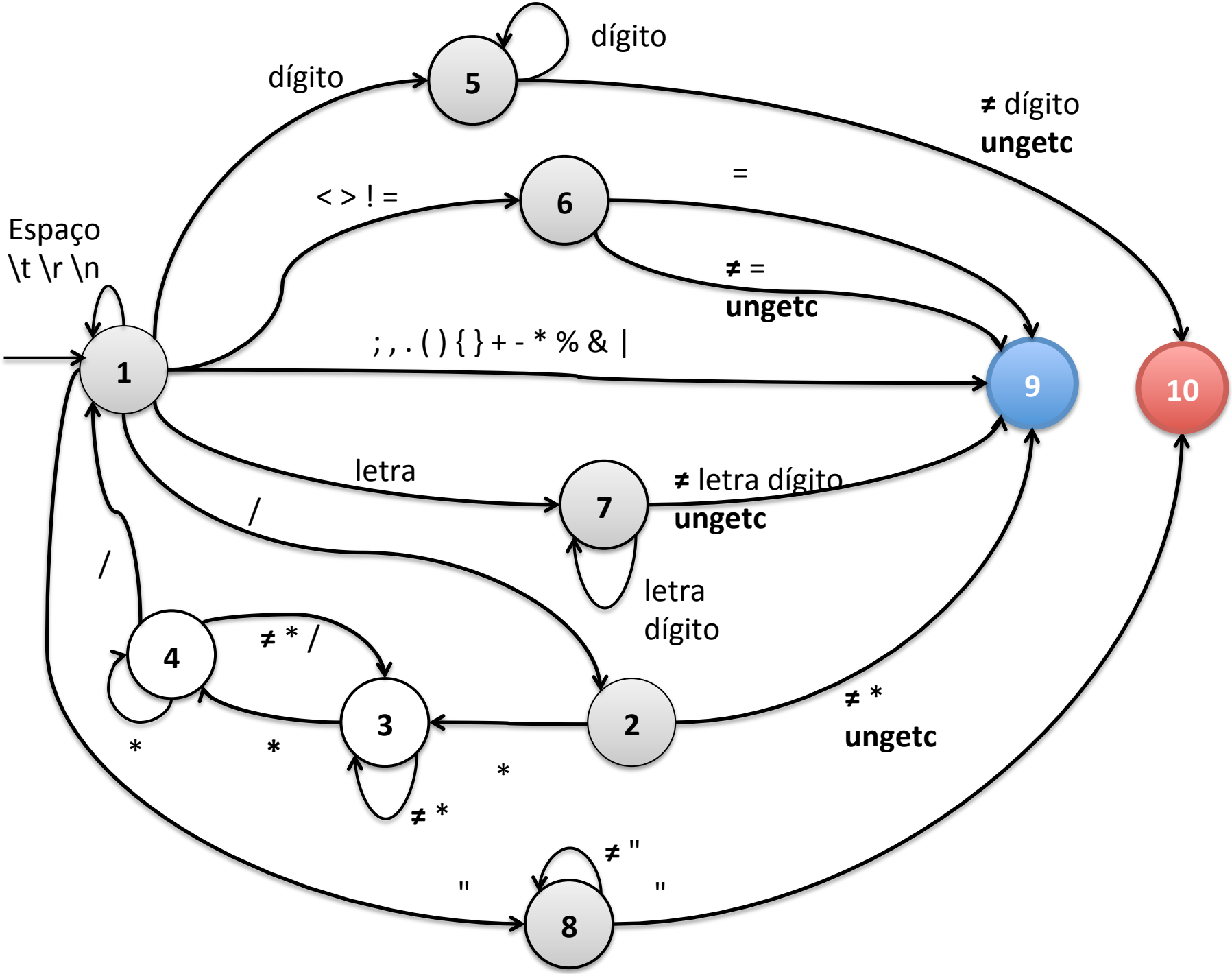
5. Avaliação

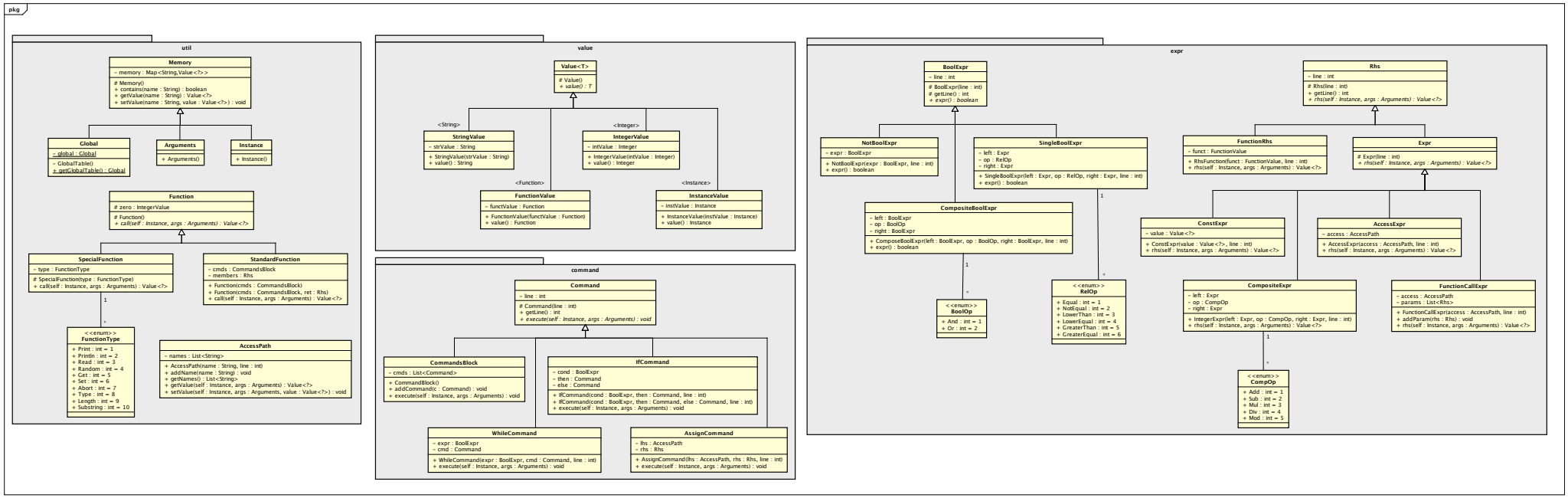
O trabalho deve ser feito em grupo de até dois alunos, sendo esse limite superior estrito. O trabalho será avaliado em 15 pontos, onde essa nota será multiplicada por um fator entre 0.0 e 1.0 para compor a nota de cada aluno individualmente. Esse fator poderá estar condicionado a apresentações presenciais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

6. Submissão

O trabalho deverá ser submetido até as 23:55 do dia 23/04/2018 (segunda-feira) via sistema acadêmico em pasta específica. Não serão aceitos, em hipótese alguma, trabalhos enviados por e-mail ou por quaisquer outras fontes.


dígito
dígito
5
≠ dígito
ungetc
=
< > ! =
6
Espaço
\t \r \n
≠ =
ungetc
; , . ( ) { } + - * % & |
1
9
10
letra
7
≠ letra dígito
ungetc
/
/
letra
dígito
4
≠ * /
3
2
≠ *
ungetc
*
*
*
≠ *
≠ "
"
"
8


pkg
util
Memory
- memory : Map<String,Value<?>>
Memory()
+ contains(name : String) : boolean
+ getValue(name : String) : Value<?>
+ setValue(name : String, value : Value<?>) : void
Global
- global : Global
- GlobalTable()
+ getGlobalTable() : Global
Arguments
+ Arguments()
Instance
+ Instance()
Function
zero : IntegerValue
Function()
+ call(self : Instance, args : Arguments) : Value<?>
SpecialFunction
- type : FunctionType
SpecialFunction(type : FunctionType)
+ call(self : Instance, args : Arguments) : Value<?>
StandardFunction
- cmds : CommandsBlock
- members : Rhs
+ Function(cmds : CommandsBlock)
+ Function(cmds : CommandsBlock, ret : Rhs)
+ call(self : Instance, args : Arguments) : Value<?>
1
*
<<enum>>
FunctionType
+ Print : int = 1
+ Println : int = 2
+ Read : int = 3
+ Random : int = 4
+ Get : int = 5
+ Set : int = 6
+ Abort : int = 7
+ Type : int = 8
+ Length : int = 9
+ Substring : int = 10
AccessPath
- names : List<String>
+ AccessPath(name : String, line : int)
+ addName(name : String) : void
+ getNames() : List<String>
+ getValue(self : Instance, args : Arguments) : Value<?>
+ setValue(self : Instance, args : Arguments, value : Value<?>) : void
value
Value<T>
Value()
+ value() : T
<String>
<Integer>
<Function>
<Instance>
StringValue
- strValue : String
+ StringValue(strValue : String)
+ value() : String
IntegerValue
- intValue : Integer
+ IntegerValue(intValue : Integer)
+ value() : Integer
FunctionValue
- functValue : Function
+ FunctionValue(functValue : Function)
+ value() : Function
InstanceValue
- instValue : Instance
+ InstanceValue(instValue : Instance)
+ value() : Instance
command
Command
- line : int
Command(line : int)
+ getLine() : int
+ execute(self : Instance, args : Arguments) : void
CommandsBlock
- cmds : List<Command>
+ CommandBlock()
+ addCommand(c : Command) : void
+ execute(self : Instance, args : Arguments) : void
IfCommand
- cond : BoolExpr
- then : Command
- else : Command
+ IfCommand(cond : BoolExpr, then : Command, line : int)
+ IfCommand(cond : BoolExpr, then : Command, else : Command, line : int)
+ execute(self : Instance, args : Arguments) : void
WhileCommand
- expr : BoolExpr
- cmd : Command
+ WhileCommand(expr : BoolExpr, cmd : Command, line : int)
+ execute(self : Instance, args : Arguments) : void
AssignCommand
- lhs : AccessPath
- rhs : Rhs
+ AssignCommand(lhs : AccessPath, rhs : Rhs, line : int)
+ execute(self : Instance, args : Arguments) : void
expr
BoolExpr
- line : int
BoolExpr(line : int)
getLine() : int
+ expr() : boolean
NotBoolExpr
- expr : BoolExpr
+ NotBoolExpr(expr : BoolExpr, line : int)
+ expr() : boolean
SingleBoolExpr
- left : Expr
- op : RelOp
- right : Expr
+ SingleBoolExpr(left : Expr, op : RelOp, right : Expr, line : int)
+ expr() : boolean
CompositeBoolExpr
- left : BoolExpr
- op : BoolOp
- right : BoolExpr
+ ComposeBoolExpr(left : BoolExpr, op : BoolOp, right : BoolExpr, line : int)
+ expr() : boolean
<<enum>>
BoolOp
+ And : int = 1
+ Or : int = 2
<<enum>>
RelOp
+ Equal : int = 1
+ NotEqual : int = 2
+ LowerThan : int = 3
+ LowerEqual : int = 4
+ GreaterThan : int = 5
+ GreaterEqual : int = 6
Rhs
- line : int
Rhs(line : int)
+ getLine() : int
+ rhs(self : Instance, args : Arguments) : Value<?>
FunctionRhs
- funct : FunctionValue
+ RhsFunction(funct : FunctionValue, line : int)
+ rhs(self : Instance, args : Arguments) : Value<?>
Expr
Expr(line : int)
+ rhs(self : Instance, args : Arguments) : Value<?>
ConstExpr
- value : Value<?>
+ ConstExpr(value : Value<?>, line : int)
+ rhs(self : Instance, args : Arguments) : Value<?>
AccessExpr
- access : AccessPath
+ AccessExpr(access : AccessPath, line : int)
+ rhs(self : Instance, args : Arguments) : Value<?>
CompositeExpr
- left : Expr
- op : CompOp
- right : Expr
+ IntegerExpr(left : Expr, op : CompOp, right : Expr, line : int)
+ rhs(self : Instance, args : Arguments) : Value<?>
FunctionCallExpr
- access : AccessPath
- params : List<Rhs>
+ FunctionCallExpr(access : AccessPath, line : int)
+ addParam(rhs : Rhs) : void
+ rhs(self : Instance, args : Arguments) : Value<?>
<<enum>>
CompOp
+ Add : int = 1
+ Sub : int = 2
+ Mul : int = 3
+ Div : int = 4
+ Mod : int = 5