KDecaf

Carlos López

July 26, 2010

Gramática

Keywords

```
class  struct true   false  void   if     else
while  return int    char   boolean
```

Producciones

```
program           ::= 'class' 'Program' '{' {declaration} '}'
declaration       ::= varDeclaration
                    | structDeclaration
                    | methodDeclaration
varDeclaration    ::= varType ident '[' numericLit ']' ';'
                    | varType ident ';'
                    | structVarDecl
                    | structDeclaration ident ';'
                    | structDeclaration ident '[' numericLit ']'
                      ';'
structVarDecl     ::= 'struct' ident ident ';'
structDeclaration ::= 'struct' ident {varDeclarations}
varType           ::= primitiveType
                    | 'void'
primitiveType     ::= 'int'
                    | 'char'
                    | 'boolean'
methodDeclaration ::= varType ident parameterList block
parameterList     ::= '(' parameter [ ',' parameter ] ')'
```

```
parameter          ::= primitiveType ident
                     | primitiveType ident '[' ']'
block              ::= '{' {varDeclaration} {statement} '}'
statement          ::= ifStatement
                     | whileStatement
                     | returnStatement
                     | methodCall ';'
                     | block
                     | assignment
                     | expression ';'
ifStatement        ::= 'if' '(' expression ')' block 'else' block
                     | 'if' '(' expression ')' block
whileStatement     ::= 'while' '(' expression ')' block
returnStatement    ::= 'return' [expression] ';'
methodCall         ::= ident '(' arguments ')'
arguments          ::= expression [ ',' expression ]
assignment         ::= location '=' expression
expression         ::= expressionOp ['&&' | '||' expression ]
expressionOp       ::= expressionSum [valueComparators
    expressionSum]
valueComparators   ::= '<='|'<'|'>'|'>='|'=='|'!='
expressionSum      ::= expressionMult [ '+'|'-' expressionSum ]
expressionMult     ::= unaryOpExpression ['/'|'*','%'
    expressionMult]
unaryOpExpression  ::= '-' simpleExpression
                     | '!' simpleExpression
                     | simpleExpression
simpleExpression   ::= literal
                     | '(' expression ')'
                     | methodCall
                     | location
literal            ::= numericLit
                     | charLit
                     | 'false'
                     | 'true'
location           ::= ident '[' expression ']' ['.' location]
                     | ident ['.' location]
```

Sistema de tipos

Axiomas

Cualquier literal es de tipo varType

```
literal -> numericLit
{ literal.type = int;
  literal.value = numericLit.lexema }
literal -> charLit
{ literal.type = char;
  literal.value = charLit.lexema.charAt(0) }
literal -> 'false' | 'true'
{ literal.type = boolean;
  literal.value = lexema.toBoolean }
```

Reglas semánticas

Identificadores

```
location -> ident
//simple location
{ assert(exists(ident));
  location = lookUp(ident); }

//array location
location -> ident '[' expression ']'
{ assert(lookUp(ident).isInstanceOf[KArray]);
  assert(expression.isInstanceOf[int]);
  location = lookUp(ident).getUnderlyingType }

//array location with member
location -> ident '[' expression ']' '.' location2
{
  assert(lookUp(ident).isInstanceOf[KArray]);
  assert(expression.isInstanceOf[int]);
  val s = lookup(ident).getUnderlyingType
  assert(s.isInstanceOf[Struct])
  val = varDeclarations = s.varDeclarations
  assert(varDeclarations.exists(_.id == location2.literal));
```

```
  location = T.find(_.id == location2).get }

//simple location with member
location -> ident '.' location2 {
  assert(exists(ident);
  val s = lookup(ident).getUnderlyingType
  assert(s.isInstanceOf[Struct])
  val = varDeclarations = s.varDeclarations
  assert(varDeclarations.exists(_.id == location2.literal));
  location = T.find(_.id == location2).get
}

simpleExpression -> literal {
       literal
}

//parenthesis expression
simpleExpression -> '(' expression ')' {
       expression
}

///method call expression
simpleExpression -> methodCall {
       methodCall
}

//location expression
simpleExpression -> location {
       location
}

unaryOpExpression -> '-' simpleExpression{
       assert(
          simpleExpression.isInstanceOf[ExpressionOperation[Int]] )
       unaryOpExpression:ExpressionOperation[Int] =
          ExpressionMult(int(-1),simpleExpression)
}

unaryOpExpression -> '!' simpleExpression{
```

```
        assert(
            simpleExpression.isInstanceOf[ExpressionOperation[Boolean]]
        unaryOpExpression:ExpressionOperation[Boolean] =
            NotExpression(simpleExpression)
}

unaryOpExpression -> simpleExpression{
        unaryOpExpression = simpleExpression
}

expressionMult -> unaryOpExpression{
        unaryOpExpression
}

expressionMult -> unaryOpExpression '/' expressionMult2{
        assert( unaryOpExpression.isInstanceOf[UnaryOperation[Int]]
            && expressionMult2.isInstanceOf[BinaryOperation[Int]])
        expressionMult = ExpressionDiv( unaryOpExpression,
            expressionMult)
}

expressionMult -> unaryOpExpression '*' expressionMult2{
        assert( unaryOpExpression.isInstanceOf[UnaryOperation[Int]]
            && expressionMult2.isInstanceOf[BinaryOperation[Int]])
        expressionMult = ExpressionMult( unaryOpExpression,
            expressionMult)
}

expressionMult -> unaryOpExpression '\%' expressionMult2{
        assert( unaryOpExpression.isInstanceOf[UnaryOperation])
        assert(
            unaryOpExpression.asInstanceOf[UnaryOperation].getUnderlyingType
            == Int )
        assert( expressionMult2.isInstanceOf[BinaryOperation])
        assert(
            expressionMult2.asInstanceOf[BinaryOperation].getUnderlyingType
            == Int)
        expressionMult = ExpressionMod( unaryOpExpression,
            expressionMult)
}
```

```
expressionOp -> expressionSum{
        expressionSum
}

expressionOp -> expressionSum valueComparators expressionMult{
        assert ( expressionSum.isInstanceOf[ExpressionOperation] )
        assert ( expressionSum.getUnderlyingType == "Int")
        assert ( expressionMult.isInstanceOf[ExpressionOperation])
        valueComparators match{
            case "<=" =>
                ExpressionLessOrEquals(expressionSum,expressionMult)
            case "<" => ExpressionLess(expressionSum,expressionMult)
            case ">" =>
                ExpressionGreater(expressionSum,expressionMult)
            case ">=" =>
                ExpressionGreaterOrEquals(expressionSum,expressionMult)
        case "==" => ExpressionEquals(expressionSum,expressionMult)
            case "!=" =>
                ExpressionNotEquals(expressionSum,expressionMult)
    }
}

expression -> expressionOp{
        expressionOp
}

expression -> expressionOp '&&' expression{
        assert( expressionOp.isInstanceOf[ExpressionOperation] )
        assert( expressionOp.getUnderlyingType == "Boolean" )
        assert( expression.getUnderlyingType == "Boolean" )
        expressionAnd(expressionOp,expression)
}

expression -> expressionOp '||' expression{
        assert( expressionOp.isInstanceOf[ExpressionOperation] )
        assert( expressionOp.getUnderlyingType == "Boolean" )
        assert( expression.getUnderlyingType == "Boolean" )
        expressionOr(expressionOp,expression)
}
```

```
assignment -> location '=' expression{
        assert ( location.getUnderlyingType ==
            expression.getUnderlyingType )
        Assignment(locatio,expression)
}

arguments -> expression

arguments -> expression {',' expression}

methodCall -> ident '(' arguments ')'{
        val methodDeclaration = lookUp(ident)
        assert ( methodDeclaration.isInstanceOf[MethodDeclaration] )
        assert ( methodDeclaration.parameters.size ==
            arguments.size )
        methodDeclaration.parameters.ZipWithIndex.foreach{
                parameterWithIndex => {
                        val index = parameterWithIndex._2
                        val parameter:Parameter = parameterWithIndex
                        assert( parameter.getUnderlyingType ==
                            arguments(index) )
                }
        }
}

returnStatement -> 'return' {
        ReturnStatament(None)
}

returnStatement -> 'return' expression ';'{
        ReturnStatement(expression)
}

whileStatement -> 'while' '(' expression ')' block{
        assert ( expression.getUnderlyingType == "Boolean" )
        WhileStatement(expression,block)
}

ifStatement -> 'if' '(' expression ')' block {
```

```
        assert ( expression.getUnderlyingType == "Boolean" )
        IfStatement(expression,block)
}

ifStatement -> 'if' '(' expression ')' block 'else' block2{
        assert ( expression.getUnderlyingType == "Boolean")
        IfStatement( expression, block, block2)
}

statement -> ifStatement{
        ifStatement
}

statement -> whileStatement{
        whileStatement
}

statement -> returnStatement{
        whileStatement
}

statement -> methodCall ';'{
        methodCall
}

statement -> block {
        block
}

statement -> assignment{
        assignment
}

block -> '{' varDeclarations statements '}'{
        Block(varDeclarations,statements)
}

parameter -> primitiveType ident{
        PrimitiveTypeParameter(primitiveType,ident)
```

```
}

parameter -> primitiveType ident '[' ']'{
       PrimitiveArrayParameter(primitiveType,ident)
}

primitiveType -> 'int' {
       int(0)
}

primitiveType -> 'char' {
       char(0)
}

primitiveType -> 'boolean'{
       boolean(0)
}

structDeclaration -> 'struct' ident varDeclarations{
       assert( !table.contains(ident) )
       Struct(ident,varDeclarations)
}

structVarDecl -> 'struct' ident ident2 ';'{
       assert( !table.contains(ident2) )
       assert( table.contains(ident) )
       VarDeclaration(struct(ident),ident2)
}

varDeclaration -> varType ident '[' numericLit ']' ';'{
       assert( !table.contains(ident) )
       VarDeclaration(ident,KArray(varType,numericLit))
}

varDeclaration -> varType ident ';'{
       assert ( !table.contains(ident) )
       VarDeclaration(varType,ident)
}
```

```
varDecalration -> structVarDecl{
        structVarDecl
}

varDeclaration -> structDeclaration ident ';'{
        assert ( !table.contains(ident))
        VarDeclaration(structDeclaration,ident)
}

declaration -> varDeclaration{
        varDeclaration
}

declaration -> structDeclaration{
        structDeclaration
}

program -> 'class' 'Program' '{' declarations '}' {
        Program("Program",declarations)
}
```

Algunas reglas semánticas programadas

```
package parsing

import lexical.KDecafLexer
import scala.util.parsing.combinator.{syntactical,PackratParsers}
import syntactical.{StandardTokenParsers}
import scala.util.parsing.input.CharArrayReader.EofCh
import ast._

/**
 * A Parser for the Decaf language
 *
 * @author Carlos Lopez
 * @version 1.0
 * @since 1.0
 */
```

```
class KDecafParser extends StandardTokenParsers with
    PackratParsers{
  override val lexical = new KDecafLexer

  lazy val program:PackratParser[Program] = "class" ~> ident ~
     declarations ^^ { case programName~declarations =>
     Program(programName,declarations)}

  lazy val declarations:PackratParser[List[Declaration]] = "{" ~>
     rep(declaration) <~ "}"

  lazy val declaration:PackratParser[Declaration] = varDeclaration
     | structDeclaration | methodDeclaration

  lazy val varDeclarations:PackratParser[List[VarDeclaration]] =
     "{" ~> rep(varDeclaration) <~ "}"

  lazy val varDeclaration:PackratParser[VarDeclaration] =
     varArrayDeclaration | varType ~ ident <~ ";" ^^ {
    case varType~id => VarDeclaration(varType,id)
  } | structVarDeclaration | structConstructorVarDeclaration

  lazy val
     structConstructorVarDeclaration:PackratParser[VarDeclaration]
     = structDeclaration ~ ident <~ ";" ^^ {
    case structDeclaration~id =>
       VarDeclaration(structDeclaration.value,id)
  } | structDeclaration ~ ident ~ arraySizeDeclaration ^^ {
    case structDeclaration~id~arraySie =>
       VarDeclaration(structDeclaration.value,id)
  }

  lazy val structVarDeclaration:PackratParser[VarDeclaration] =
     "struct" ~> ident ~ ident <~ ";" ^^ {
    case structName ~ id => VarDeclaration(struct(structName),id)
  }

  lazy val varArrayDeclaration:PackratParser[VarDeclaration] =
     varType ~ ident ~ arraySizeDeclaration ^^ {
    case varType~id~arraySize =>
```

```
    VarDeclaration(
      KArray(Array.fill(arraySize)(varType)) //varType has its
          value set to default value
      ,id
    )
}

lazy val arraySizeDeclaration:PackratParser[Int] = "[" ~>
    numericLit <~ "]" <~ ";" ^^ (_.toInt)

lazy val structDeclaration:PackratParser[StructDeclaration] =
    "struct" ~> ident ~ varDeclarations ^^ {
  case structName~varDeclarations =>
      StructDeclaration(structName, Struct(varDeclarations))
}

lazy val varType:PackratParser[VarType[_]] = primitiveType |
    "void" ^^ {
  _ => void({})
}

lazy val primitiveType:PackratParser[PrimitiveType[_]] = "int" ^^
    { _ => int(0) } | "char" ^^ { _ => char(' ')} | "boolean" ^^
    {_ => boolean(false)}

lazy val methodDeclaration:PackratParser[MethodDeclaration] =
    varType ~ ident ~ parameterList ~ block ^^ {
  case methodType~name~parameters~codeBlock =>
      MethodDeclaration(methodType,name,parameters,codeBlock)
}

lazy val parameterList:PackratParser[List[Parameter]] = {
  "(" ~> repsep(parameter,",") <~ ")"
}

lazy val parameter:PackratParser[Parameter] = primitiveType ~
    ident ^^ {
  case pType~name => PrimitiveTypeParameter(pType,name)
} | primitiveType ~ ident <~ "[" <~ "]" ^^ {
  case pType~name => PrimitiveArrayParameter(pType,name)
```

```
}

lazy val block:PackratParser[Block] = "{" ~> rep(varDeclaration)
    ~ statements <~ "}" ^^ {
  case varDeclarations ~ statements =>
     Block(varDeclarations,statements)
}

lazy val statements:PackratParser[List[Statement]] =rep(statement)

lazy val statement:PackratParser[Statement] = ifStatement |
    whileStatement | returnStatement | methodCall <~ ";" | block
    | assignment | expression <~ ";"

lazy val parenthesisExpression:PackratParser[Expression] = "(" ~>
    expression <~ ")"

lazy val arguments:PackratParser[List[Expression]] = "(" ~>
    repsep(expression,",") <~ ")"

lazy val ifStatement:PackratParser[IfStatement] = ifElseStatement
    | "if" ~> parenthesisExpression ~ block ^^ {
  case expr ~ ifBlock => IfStatement(expr,ifBlock)
}

lazy val ifElseStatement:PackratParser[IfStatement] = "if" ~>
    parenthesisExpression ~ block ~ "else" ~ block ^^ {
  case expr ~ ifBlock ~ "else" ~ elseBlock =>
     IfStatement(expr,ifBlock,Some(elseBlock))
}

lazy val whileStatement:PackratParser[WhileStatement] = "while"
    ~> parenthesisExpression ~ block ^^ {
  case expression_ ~ block => WhileStatement(expression_,block)
}

lazy val returnStatement:PackratParser[ReturnStatement] =
    "return" ~> opt(expression) <~ ";" ^^ { ReturnStatement(_) }
```

```
lazy val methodCall:PackratParser[MethodCall] = ident ~ arguments
    ^^ {
  case id~args => MethodCall(id,args)
}

lazy val assignment:PackratParser[Assignment] = location ~ "=" ~
    expression ^^ {
  case loc ~ "=" ~ expr => Assignment(loc,expr)
}

lazy val expression:PackratParser[Expression] =
    expressionOperation ~ opt(
  ("&&"|"||") ~ expression
) ^^ {
  case exp ~ None => exp
  case exp1 ~ Some(expressionWithOperator) =>
      expressionWithOperator match{
    case "&&" ~ exp2 => ExpressionAnd(exp1,exp2)
    case "||" ~ exp2 => ExpressionOr(exp1,exp2)
  }
}

lazy val expressionOperation:PackratParser[Expression] =
    expressionSum ~ opt(
  ("<="|"<"|">"|">="|"=="|"!=") ~ expressionSum
) ^^ {
  case exp ~ None => exp
  case exp1 ~ Some(comparedExpression) => comparedExpression
      match{
    case compareSymbol ~ exp2 => compareSymbol match{
      case "<=" => ExpressionLessOrEquals(exp1,exp2)
      case "<" => ExpressionLess(exp1,exp2)
      case ">" => ExpressionGreater(exp1,exp2)
      case ">=" => ExpressionGreaterOrEquals(exp1,exp2)
      case "==" => ExpressionEquals(exp1,exp2)
      case "!=" => ExpressionNotEquals(exp1,exp2)
    }
  }
}
```

```
lazy val expressionSum:PackratParser[Expression] = expressionMult
    ~ opt(
  ("+"|"-") ~ expressionSum
) ^^ {
  case exp ~ None => exp
  case exp1 ~ Some(exp2WithOperator) => exp2WithOperator match{
    case "+" ~ exp2 => ExpressionAdd(exp1,exp2)
    case "-" ~ exp2 => ExpressionSub(exp1,exp2)
  }
}

lazy val expressionMult:PackratParser[Expression] =
    unaryOperationExpression ~ opt(
  ("*"|"/"|"%") ~ expressionMult
) ^^ {
  case exp ~ None => exp
  case exp1 ~ Some(exp2WithOperator) => exp2WithOperator match{
    case "*" ~ exp2 => ExpressionMult(exp1,exp2)
    case "%" ~ exp2 => ExpressionMod(exp1,exp2)
    case "/" ~ exp2 => ExpressionDiv(exp1,exp2)
  }
}

lazy val unaryOperationExpression :PackratParser[Expression] =
    "-" ~> simpleExpression ^^ { NegativeExpression(_)} | "!" ~>
    simpleExpression ^^ { NotExpression(_)} | simpleExpression

//expression without operations
lazy val simpleExpression:PackratParser[Expression] = literal |
    "(" ~> expression <~ ")" | methodCall | location

lazy val literal:PackratParser[PrimitiveType[_]] = numericLit ^^
    { lit =>int(lit.toInt)} | charLit | "true" ^^ { lit =>
    boolean(lit.toBoolean)} | "false" ^^ { lit =>
    boolean(lit.toBoolean)}

lazy val charLit:PackratParser[char] = elem("char", x => {
    x.isInstanceOf[lexical.CharLit] }) ^^ {c =>
    char(c.chars.charAt(0))}
```

```
  lazy val location:PackratParser[Location] = arrayLocation | ident
      ~ optionalLocation ^^ {
    case id~optLocation => SimpleLocation(id,optLocation)
  }

  lazy val optionalLocation:PackratParser[Option[Location]] =
      opt("." ~> location)

  lazy val arrayLocation:PackratParser[Location] = ident ~
      arrayLocationExpression ~ optionalLocation ^^ {
    case id~exp~optLocation => ArrayLocation(id,exp,optLocation)
  }

  lazy val arrayLocationExpression:PackratParser[Expression] = "["
      ~> expression <~ "]"

  def parseTokens[T <: lexical.Scanner](tokens:T) = program(tokens)

  def parse(s:String) = {
    val tokens = new lexical.Scanner(s)
    val packratReader = new PackratReader(tokens)
    program(packratReader)
  }
}
```

Nodos del árbol sintáctico abstracto: Sistema de tipos con sus subtipos

```
package parsing.ast

/**
 * ASTs class nodes
 *
 * @author Carlos Lopez
 * @version 1.0
 * @since 1.0
 */

sealed trait KDecafAST extends Product{
```

```
  override def toString = getClass.getName
  implicit def s(s:String):KDecafAST = StringWrapper(s)
  val children: List[KDecafAST]


}

case class StringWrapper(val s:String) extends KDecafAST{
  val children = Nil
  override def toString = s
}

case class Program(val name:String, val
    declarations:List[Declaration]) extends KDecafAST{
  val children = declarations
}

abstract class Declaration extends KDecafAST

case class VarDeclaration(val varType:VarType[_], val id:String)
    extends Declaration{
  val children = List(s("name: "+id),varType)
}

case class StructDeclaration(val name:String, val value:Struct)
    extends Declaration{
  val children = List(s(name),value)
}

case class MethodDeclaration(val methodType:VarType[_],val
    name:String,val parameters:List[Parameter],val codeBlock:Block)
    extends Declaration{
  val children:List[KDecafAST] =
      List(methodType,s(name))++parameters:+codeBlock
}

trait VarType[+T] extends Expression with KDecafAST{
  val value:T
  override val children:List[KDecafAST] = List(value.toString)
}
```

```
abstract class PrimitiveType[+T] extends VarType[T]

//basic types
case class int(val value:Int) extends PrimitiveType[Int]

case class char(val value:Char) extends PrimitiveType[Char]

case class boolean(val value:Boolean) extends
    PrimitiveType[Boolean]

case class struct(val value:String) extends VarType[String]{ //the
    name of the struct
}

case class void(val value:Unit) extends VarType[Unit]

trait TypeConstructor[+T] extends VarType[T]

case class KArray[U <: VarType[_]](val value:Array[U]) extends
    TypeConstructor[Array[U]]{
  override val children = value.toList
}

case class Struct(val value:List[VarDeclaration]) extends
    TypeConstructor[List[VarDeclaration]]{
  override val children = value
}

abstract class Parameter extends KDecafAST{
  val varType:PrimitiveType[_]
  val name:String
  override val children:List[KDecafAST] = List(varType,name)
}

case class PrimitiveTypeParameter(val varType:PrimitiveType[_],
    val name:String) extends Parameter

case class PrimitiveArrayParameter(val varType:PrimitiveType[_],
    val name:String) extends Parameter
```

```
case class Block(val varDeclarations:List[VarDeclaration], val
    statements:List[Statement]) extends Statement{
  override val children:List[KDecafAST] =
      varDeclarations++statements
}

abstract class Statement extends KDecafAST{
  val children:List[KDecafAST]
}

trait ConditionStatement extends Statement{
  val expression:Expression
  val codeBlock:Block
}

case class IfStatement(val expression:Expression,val
    codeBlock:Block, val elseBlock:Option[Block] = None) extends
    ConditionStatement{
  val children:List[KDecafAST] = elseBlock match{
    case Some(elseBlock) => List(expression,codeBlock,elseBlock)
    case _ => List(expression,codeBlock)
  }
}

case class WhileStatement(val expression:Expression,val
    codeBlock:Block) extends ConditionStatement{
  val children:List[KDecafAST] = List(expression,codeBlock)
}

case class MethodCall(val name:String,val
    arguments:List[Expression]) extends Expression {
  val children:List[KDecafAST] = List(s(name))++arguments
}

case class ReturnStatement(val expression:Option[Expression])
    extends Statement{
  val children:List[KDecafAST] = expression match{
    case Some(exp) => List(exp)
    case _ => Nil
```

```
  }
}

case class Assignment(val location:Location,val
    expression:Expression) extends Statement{
  val children:List[KDecafAST] = List(location,expression)
}

abstract class Location extends Expression

case class SimpleLocation(val name:String, val
    optionalMember:Option[Location] = None) extends Location{
  val children:List[KDecafAST] = optionalMember match{
    case Some(member) => List(s(name),member)
    case _ => List(s(name))
  }
}

case class ArrayLocation(val name:String, val index:Expression,
    val optionalMember:Option[Location] = None) extends Location{
  val children:List[KDecafAST] = optionalMember match{
    case Some(member) => List(s(name),index,member)
    case _ => List(s(name))
  }
}

abstract class Expression extends Statement

abstract class Operator[T]{
  val lexeme:T
}

case class ArithmeticOperator(val lexeme:Char) extends
    Operator[Char]

case class InequalityOperator(val lexeme:String) extends
    Operator[String]

case class EqualityOperator(val lexeme:String) extends
    Operator[String]
```

```
case class ConditionalOperator(val lexeme:String) extends
    Operator[String]

trait ExpressionOperation extends Expression

trait BinaryOperation[+T] extends ExpressionOperation{
  val exp1:Expression
  val exp2:Expression

  val children:List[Expression] = List(exp1,exp2)
}

trait UnaryOperation extends ExpressionOperation{
  val exp:Expression
  val children:List[Expression] = List(exp)
}

case class ExpressionAdd(val exp1:Expression,val exp2:Expression)
    extends BinaryOperation[Int]
case class ExpressionSub(val exp1:Expression,val exp2:Expression)
    extends BinaryOperation[Int]
case class ExpressionMult(val exp1:Expression, val
    exp2:Expression) extends BinaryOperation[Int]
case class ExpressionDiv(val exp1:Expression, val exp2:Expression)
    extends BinaryOperation[Int]
case class ExpressionMod(val exp1:Expression, val exp2:Expression)
    extends BinaryOperation[Int]

case class ExpressionAnd(val exp1:Expression, val exp2:Expression)
    extends BinaryOperation[Boolean]
case class ExpressionOr(val exp1:Expression, val exp2:Expression)
    extends BinaryOperation[Boolean]

case class ExpressionLessOrEquals(val exp1:Expression, val
    exp2:Expression) extends BinaryOperation[Boolean]
case class ExpressionLess(val exp1:Expression, val
    exp2:Expression) extends BinaryOperation[Boolean]
case class ExpressionGreater(val exp1:Expression, val
    exp2:Expression) extends BinaryOperation[Boolean]
```

```
case class ExpressionGreaterOrEquals(val exp1:Expression, val
    exp2:Expression) extends BinaryOperation[Boolean]
case class ExpressionEquals(val exp1:Expression, val
    exp2:Expression) extends BinaryOperation[Boolean]
case class ExpressionNotEquals(val exp1:Expression, val
    exp2:Expression) extends BinaryOperation[Boolean]


case class NegativeExpression(val exp:Expression) extends
    UnaryOperation
case class NotExpression(val exp:Expression) extends UnaryOperation
```