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          ::= 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[Expression[Int]] )
}

unaryOpExpression -> '!' simpleExpression{
       assert( simpleExpression.isInstanceOf[Expression[Boolean]]
}
```

```
unaryOpExpression -> simpleExpression

expressionMult -> unaryOpExpression

expressionMult -> unaryOpExpression '/' expressionMult

expressionMult -> unaryOpExpression '*' expressionMult

expressionMult -> unaryOpExpression '%' expressionMult

expressionOp -> expressionSum

expressionOp -> expressionSum valueComparators expressionMult

expression -> expressionOp

expression -> expressionOp '&&' expression

expression -> expressionOp '||' expression

assignment -> location '=' expression

arguments -> expression

arguments -> expression {',' expression}

methodCall -> ident '(' arguments ')'

returnStatement -> 'return'

returnStatement -> 'return' expression ';'

whileStatement -> 'while' '(' expression ')' block

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

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

```
statement -> ifStatement

statement -> whileStatement

statement -> returnStatement

statement -> methodCall ';'
```