```
grammar Grammar;

import Tokenizer;

@header {
      import compiler.ast.*;
      import compiler.ast.type.*;
      import compiler.ast.statement.*;
      import compiler.ast.expression.*;
      import compiler.ast.definition.*;
}

 // PROGRAM

program returns [Program ast]
      : 'class' IDENT ';' globalSection createSection
functionDefinitions+=functionDefinition* 'end' run EOF
      {$ast = new Program($IDENT, $globalSection.list, $createSection.list,
$functionDefinitions, $run.ast); }
      ;

// GLOBAL

globalSection returns [List<Definition> list = new ArrayList<Definition>()]
  : ('global' (globalTypesSection { $list.addAll($globalTypesSection.list); }
globalVarsSection { $list.addAll($globalVarsSection.list); })) ?
  ;

globalTypesSection returns [List<StructDefinition> list = new
ArrayList<StructDefinition>()]
  : ('types' (typeDefinition { $list.add($typeDefinition.ast); } )*)?
  ;
```

```
typeDefinition returns [StructDefinition ast]
  : 'deftuple' IDENT 'as' fieldDefinitions 'end' { $ast = new
StructDefinition($IDENT, $fieldDefinitions.list); }
  ;


fieldDefinitions returns [List<FieldDefinition> list = new
ArrayList<FieldDefinition>()]
  : (s=singleFieldDefinition ';' { $list.add($s.ast); }
    | m=multipleFieldDefinition ';' { $list.addAll($m.list); }
    )*
  ;


singleFieldDefinition returns [FieldDefinition ast]
  : IDENT ':' type { $ast = new FieldDefinition($IDENT.getText(), $type.ast); }
  ;


multipleFieldDefinition returns [List<FieldDefinition> list = new
ArrayList<FieldDefinition>()]
  : ids+=IDENT (',' ids+=IDENT)* ':' type
      {
        for (Token id: $ids) {
          $list.add(new FieldDefinition(id.getText(), $type.ast));
        }
      }
  ;


globalVarsSection returns [List<VarDefinition> list = new
ArrayList<VarDefinition>()]
  : ('vars' varDefinitions { $list = $varDefinitions.list; })?
  ;


// CREATE
```

```
createSection returns [List<FunctionCreation> list = new
ArrayList<FunctionCreation>()]
  : 'create' functionCreations { $list = $functionCreations.list; }
  ;


functionCreations returns [List<FunctionCreation> list = new
ArrayList<FunctionCreation>()]
  : (IDENT { $list.add(new FunctionCreation($IDENT)); } ';')*
  ;


functionDefinition returns [FunctionDefinition ast]
  : 'feature' IDENT parameters ':' type 'is' localVarsSection 'do'
statements+=statement* 'end'
  { $ast = new FunctionDefinition($IDENT, $parameters.list, $type.ast,
$localVarsSection.list, $statements); }
  | 'feature' IDENT parameters 'is' localVarsSection 'do' statements+=statement*
'end'
  { $ast = new FunctionDefinition($IDENT, $parameters.list, null,
$localVarsSection.list, $statements); }
  ;


localVarsSection returns [List<VarDefinition> list = new
ArrayList<VarDefinition>()]
  : ('local' varDefinitions { $list = $varDefinitions.list; } )?
  ;


parameters returns [List<VarDefinition> list = new ArrayList<VarDefinition>()]
  : ('(' ident1=IDENT ':' type1=type { $list.add(new
VarDefinition($ident1.getText(), $type1.ast)); }
    (',' ident2=IDENT ':' type2=type { $list.add(new
VarDefinition($ident2.getText(), $type2.ast)); } )* ')')?
  ;


// VarDefinition
```

```
varDefinitions returns [List<VarDefinition> list = new ArrayList<VarDefinition>()]
  : (s=singleVarDefinition ';' { $list.add($s.ast); }
    | m=multiVarDefinition ';' { $list.addAll($m.list); }
    )*
  ;

singleVarDefinition returns [VarDefinition ast]
  : IDENT ':' type { $ast = new VarDefinition($IDENT.getText(), $type.ast); }
  ;

multiVarDefinition returns [List<VarDefinition> list = new
ArrayList<VarDefinition>()]
  : ids+=IDENT (',' ids+=IDENT)* ':' type
      {
        for (Token id: $ids) {
          $list.add(new VarDefinition(id.getText(), $type.ast));
        }
      }
  ;

// Run

run returns [Run ast]
  : 'run' IDENT '(' arguments ')' ';' { $ast = new Run($IDENT, $arguments.list); }
  ;

// Expression

expression returns [Expression ast]
  : IDENT { $ast = new Variable($IDENT); }
      | INT_LITERAL { $ast = new IntLiteral($INT_LITERAL); }
      | REAL_LITERAL { $ast = new RealLiteral($REAL_LITERAL); }
```

```
      | CHAR_LITERAL { $ast = new CharLiteral($CHAR_LITERAL); }

      | IDENT '(' arguments ')' { $ast = new FunctionCallExpression($IDENT,
$arguments.list); }

      | left=expression '[' right=expression ']' { $ast = new
ArrayAccess($left.ast, $right.ast); }

      | expr=expression '.' IDENT { $ast = new StructAccess($expr.ast, $IDENT); }

  | operator='-' expr=expression { $ast = new ArithmeticUnary($operator,
$expr.ast); }

      | '(' expr=expression ')' { $ast = $expr.ast; }

      | operator='not' expr=expression { $ast = new LogicUnary($operator,
$expr.ast); }

      | 'to' '<' type '>' '(' expr=expression ')' { $ast = new Cast($type.ast,
$expr.ast); }

      | left=expression operator=('*'|'/'|'%') right=expression { $ast = new
ArithmeticBinary($left.ast, $operator, $right.ast); }

      | left=expression operator=('+'|'-') right=expression { $ast = new
ArithmeticBinary($left.ast, $operator, $right.ast); }

      | left=expression operator=('<'|'>'|'<='|'>=') right=expression { $ast = new
RelationalBinary($left.ast, $operator, $right.ast); }

      | left=expression operator=('<>'|'=') right=expression { $ast = new
RelationalBinary($left.ast, $operator, $right.ast); }

      | left=expression operator='and' right=expression { $ast = new
LogicBinary($left.ast, $operator, $right.ast); }

      | left=expression operator='or' right=expression { $ast = new
LogicBinary($left.ast, $operator, $right.ast); }

      ;


arguments returns [List<Expression> list = new ArrayList<Expression>()]

  : (expr1=expression { $list.add($expr1.ast); } (',' expr2=expression
{ $list.add($expr2.ast); } )*)?

  ;


expressions returns [List<Expression> list = new ArrayList<Expression>()]

  : (expr1=expression { $list.add($expr1.ast); } (',' expr2=expression
{ $list.add($expr2.ast); })*)?

  ;
```

```
// Statement


statement returns [Statement ast]

  : 'print' expressions ';' { $ast = new Print($expressions.list); }

  | 'println' expressions ';' { $ast = new Println($expressions.list); }

      | 'read' expressions ';' { $ast = new Read($expressions.list); }

      | IDENT '(' arguments ')' ';' { $ast = new FunctionCallStatement($IDENT,
$arguments.list); }

      | left=expression ':=' right=expression ';' { $ast = new
Assignment($left.ast, $right.ast); }

      | 'if' '(' expr=expression ')' '{' ifStatements+=statement* '}' 'else' '{'
elseStatements+=statement* '}'  { $ast = new Conditional($expr.ast, $ifStatements,
$elseStatements); }

      | 'if' '(' expr=expression ')' '{' ifStatements+=statement* '}' { $ast = new
Conditional($expr.ast, $ifStatements, null); }

      | fromClause 'until' expr=expression 'loop' loopStatements+=statement* 'end'
{ $ast = new Loop($fromClause.list, $expr.ast, $loopStatements); }

      | 'return' expr=expression ';' { $ast = new Return($expr.ast); }

      | 'return' ';' { $ast = new Return(null); }

      ;


fromClause returns [List<Statement> list = new ArrayList<Statement>()]

  : ('from'  (expr1=expression ':=' expr2=expression ';' { $list.add(new
Assignment($expr1.ast, $expr2.ast)); } )*)?

  ;


// Type


type returns [Type ast]

      : 'INTEGER' { $ast = new IntType(); $ast.updatePositions($ctx.start); }

      | 'DOUBLE' { $ast = new RealType(); $ast.updatePositions($ctx.start); }

      | 'CHARACTER' { $ast = new CharType(); $ast.updatePositions($ctx.start); }

      | '[' INT_LITERAL ']' type { $ast = new ArrayType($INT_LITERAL, $type.ast);
$ast.updatePositions($ctx.start); }
```

```
| IDENT { $ast = new StructType($IDENT); $ast.updatePositions($ctx.start); }
;
```