## Attribute Grammar

### Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbol | Attribute Name | Java Type | Inherited/Synthesized | Description |
| Expression | expressionType | Type | Synthesized | Type of the expression |
| expression | LValue | Boolean | Synthesized | True if the expression can appear to the left of an assignment |
|  |  |  |  |  |

### Rules

|  |  |  |  |
| --- | --- | --- | --- |
| Node | | Predicates | Semantic Functions |
| program → definition\* | |  |  |
| varDefinition:definition → name:string type | |  |  |
| structDefinition:definition → name:string attrDefinition\* | |  |  |
| functionDefinition:definition → name:string params:varDefinition\* type? definitions:varDefinition\* statement\* | | primitiveOrVoid(functionDefinition.getType())  for(param : params){  primitiveType(param);  } |  |
| attrDefinition → name:string type | |  |  |
| read:statement → expression | | primitiveType(read.getExpression().getExpressionType();  read.getExpression().isLValue(); |  |
| print:statement → expression\* | | for(expression : print.getExpressions()){  primitiveType(param);  } |  |
| println:statement → expression\* | | for(expression : print.getExpressions()){  primitiveType(param);  } |  |
| printsp:statement → expression\* | | for(expression : print.getExpressions()){  primitiveType(param);  } |  |
| return:statement → expression? | if(functionReturnType.type instanceof VoidType) {  returnValue.getExpression().isPresent();  } else {  returnValue.getExpression().isEmpty();  else  !areTypesEqual(returnValue.getExpression().get().getExpressionType(),functionReturnType .type);  } | |  |
| assignment:statement → left:expression right:expression | | primitiveType(assignment.getLeft().getExpressionType();  primitiveType(assignment.getRight().getExpressionType(); |  |
| while:statement → expression statement\* | | isInt(whileValue.getExpression()); |  |
| ifelse:statement → cond:expression tr:statement\* fs:statement\* | | isInt(ifelse.getCond()); |  |
| functionCallStatement:statement → name:string expression\* | | chechArgumentTypes(functionCallStatement.params); |  |
| intLiteral:expression → intValue:int | |  | intLiteral.expressionType = IntType;  intLiteral.lvalue = false; |
| floatLiteral:expression → floatValue:float | |  | floatLiteral.expressionType = FloatType;  floatLiteral.lvalue = false; |
| charLiteral:expression → name:string | |  | charLiteral.expressionType = CharLiteral;  charLiteral.lvalue = false; |
| arrayAccess:expression → expr1:expression expr2:expression | |  |  |
| fieldAccess:expression → expr:expression name:string | |  |  |
| not:expression → expression | |  |  |
| logic:expression → left:expression operator:string right:expression | |  |  |
| arithmetic:expression → left:expression operator:string right:expression | |  |  |
| variable:expression → name:string | |  | variable.expressionType= variable.type;  variable.lvalue = true; |
| cast:expression → type expression | |  | Cast.expressionType= cast.type;  Cast.lvalue = false; |
| functionCallExpression:expression → name:string expression\* | | paramDefinitions.size() == params.size();  for(paramDefinition, param){  AreTypesEqual(paramDefinition,param );  } | functionCallExpression.expressionType = functionCallExpression.functionDefinition.type;  functionCallExpression.lvalue = false; |
| intType:type → ε | |  |  |
| floatType:type → ε | |  |  |
| charType:type → ε | |  |  |
| arrayType:type → intValue:int type | |  |  |
| structType:type → name:string | |  |  |
| voidType:type → ε | |  |  |
| errorType:type → msg:string | |  |  |

Operators samples (cut & paste if needed):  
⇒ ⇔ ≠ ∅ ∈ ∉ ∪ ∩ ⊂ ⊄ ∑ ∃ ∀