## Attribute Grammar

### Attributes

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

|  |  |
| --- | --- |
| Name | Description |
| primitiveOrVoid(type) | True if type is primitive type (int, float, char) or void. |
| primitiveType(type) | True if type is primitive type (int, float, char). |
| hasProperty(fieldAccess) | True if struct definition has the property it´s trying to access. |
| checkArgumentTypes(expression\*); | True if function definition arguments and passed arguments are same size and same types. |

### 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.type)  for(param p: params){  primitiveType(p);  } |  |
| attrDefinition → name:string type | |  |  |
| read:statement → expression | | primitiveType(expression);  expression.lvalue = true; |  |
| print:statement → expression\* | | for(expression e : expression\*){  primitiveType(e);  } |  |
| println:statement → expression\* | | for(expression e : expression\*){  primitiveType(e);  } |  |
| printsp:statement → expression\* | | for(expression e : expression\*){  primitiveType(e);  } |  |
| return:statement → expression? | if(expression instanceof VoidType) {  returnValue.getExpression().isPresent();  } else {  returnValue.getExpression().isEmpty();  else  !areTypesEqual(returnValue.getExpression().get().getExpressionType(),functionReturnType .type);  } | |  |
| assignment:statement → left:expression right:expression | | primitiveType(left);  primitiveType(right);  left.lvalue == true; |  |
| while:statement → expression statement\* | | whileValue.expression.type == IntType; |  |
| ifelse:statement → cond:expression tr:statement\* fs:statement\* | | cond.type == IntType; |  |
| functionCallStatement:statement → name:string expression\* | | checkArgumentTypes(expression\*); |  |
| intLiteral:expression → intValue:int | |  | intLiteral.type = IntType;  intLiteral.lvalue = false; |
| floatLiteral:expression → floatValue:float | |  | floatLiteral.type = FloatType;  floatLiteral.lvalue = false; |
| charLiteral:expression → name:string | |  | charLiteral.type = CharLiteral;  charLiteral.lvalue = false; |
| arrayAccess:expression → expr1:expression expr2:expression | | expr1.type == ArrayType;  expr2.type == IntType; | arrayAccess.lvalue=true; |
| fieldAccess:expression → expr:expression name:string | | expr.type == StructType;  hasProperty(fieldAccess); | fieldAccess.lvalue=true; |
| not:expression → expression | | expression.type == IntType; | Not.lvalue = false;  Not.type = IntType; |
| logic:expression → left:expression operator:string right:expression | | left.type == IntType;  right.type == IntType; | logic.lvalue = false;  logic.type = IntType |
| arithmetic:expression → left:expression operator:string right:expression | | left.type == IntType || FloatType;  right == IntType || FloatType;  left.type == right.type; | Arithmetic.lvalue = false; |
| variable:expression → name:string | |  | variable.type= variable.varDefinition.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){  paramDefinition.type = param.type  } | functionCallExpression.type = 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):  
⇒ ⇔ ≠ ∅ ∈ ∉ ∪ ∩ ⊂ ⊄ ∑ ∃ ∀