Attribute Grammar

Attributes

Symbol	Attribute Name	Java Type	Inherited/ Synthesized	Description
expression	type	Type	synthesized	Expression Type
expression	lvalue	boolean	synthesized	Is Modifiable
statement	function	FunctionDefinition	synthesized	Function of the stmnt
funcDef	hasReturn	boolean	synthesized	Has a return stmnt
structAccess	fieldDefinition	fieldDefinition	synthesized	Field related to the access

Rules

Node	Predicates	Semantic Functions
program → name:string definition* functionCreation* functionDefinition* run		
functionCreation → name:string		
varDefinition:definition → name:string type		
structDefinition:definition → name:string fieldDefinition*		
fieldDefinition:definition → name:string type		
functionDefinition:definition → name:string parameters:varDefinition* type? locals:varDefinition* statement*	isPrimitive(funcDef.params) isPrimitive(funcDef.locals) funcDef.hasReturn()	
print:statement → expression*	isPrimitive(print.expressions)	
println:statement → expression*	isPrimitive(println.expressions)	
read:statement → expression*	isPrimitive(read.expressions)	

functionCallStatement:statement → name:string expression*	For i in funcCall.expressions.length:
	  sameType(
	  funcCall.function.params,
	  funcCall.expressions
	)	

assignment:statement → left:expression right:expression	isPrimitive(assignment.left()) sameType(assignment.left, assignment.right) assignment.left.isLValue	

conditional:statement → expression ifStatements:statement* elseStatements:statement*	isInt(conditional.getExpression)	cond.ifStmnts.forEach(
		  stmnt.function = cond.function
		)
		cond.elseStmnts.forEach(
		  smnt.function = cond.function
		)

loop:statement → fromStatements:statement* expression loopStatements:statement*	isInt(loop.getExpression)	loop.loopStmtns.forEach(stmnt.function = loop.function)

return:statement → expression?	return.expression.empty ?
	  return.function isVoid :
	  sameType(return.function, return.expr)	

run → name:string expression*	For i in return.expressions.length: sameType(return.expression, return.function.param)	
intType:type → ε		
realType:type → ε		
charType:type → ε		
arrayType:type → intValue:int type		
structType:type → name:string		
voidType:type → ε		
variable:expression → name:string		Lvalue = true type = variable.varDef.type
intLiteral:expression → intValue:int		Lvalue = false type = IntType
realLiteral:expression → floatValue:float		Lvalue = false type = RealType
charLiteral:expression → name:string		Lvalue = false type = CharType
functionCallExpression:expression → name:string expression*	For i in .funcCall.length: sameType(funcCall.expression, funcCall.function.param)	Lvalue=false type = funcCall.function.type
structAccess:expression → expr:expression name:string	isStruct(structAccess.expr) stringFound == false	Lvalue = true type = fieldDefinitions[found].type fieldDefinition = fieldDefinitions[found]
arrayAccess:expression → left:expression right:expression	isAccesible(left) isInt(right)	Lvalue = true type = getType del array o de la expresion dependiendo de lo que sea left
cast:expression → castType:type expression	!sameType(castType, expression)	Lvalue = false type = cast.castType

arithmeticBinary:expression → left:expression operator:string right:expression	If op == “mod”
	  isInt(left)
	  isInt(right)
	else
	  isInt(left) || isDouble(left)
	  isInt(right) || isDouble(right)
		Lvalue = false
		type = left.type

arithmeticUnary:expression → operator:string expr:expression	isInt(expr) || isDouble(expr)	Lvalue = false type = expr.typ
logicBinary:expression → left:expression operator:string right:expression	isInt(left) isInt(right)	Lvalue = false type = IntType

logicUnary:expression → operator:string expr:expression	isInt(expr)	Lvalue = false type = IntType
relationalBinary:expression → left:expression operator:string right:expression	isInt(left) || isDouble(left) isInt(right) || isDouble(right)	Lvalue = false type = IntType

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