Attribute Grammar – *TypeChecking* – Ignacio Fernández Suárez (UO294177)

Attributes

Symbol	Attribute Name	Java Type	Inherited/Synthesized	Description
Expression	type	DataType	Synthesized	Define el tipo que tiene una expresión
Expression	lvalue	Boolean	Synthesized	Indica si es o no modificable
Stmt	feature	Feature	Synthesized	La sentencia guarda una referencia a la feature a la que pertenece
Stmt	returnable	Boolean	Synthesized	Indica si la sentencia es o no retornable
DoBlock	returnable	Boolean	Synthesized	Indica si en el doBlock retorna algo o no
Feature	returntype	DataType	Synthesized	Define el tipo que retorna la feature
Feature	constructor	Boolean	Synthesized	Indica si la feature es o no constructora

Rules

Node	Predicates	Semantic Functions
program → classDef global? create feature* runInvocation		
classDef → name:string		
runInvocation → procedure	**Comprobamos que la feature invocada en 'run' sea un constructor** runInvocation.procedure.invocation != NULL	

readStmt:stmt → expression*	**Comprobamos que la expresión sea un lvalue**
	for(Expression exp : readStmt .expressions) { exp.lvalue == TRUE }
	
	Comprobamos que la expresión sea de tipo simple
	for(Expression exp : readStmt .expressions) { isSimpleType(exp.type) == TRUE}	

printStmt:stmt → expression* format:string	**Comprobamos que cada expresión sea de tipo simple** for (Expression exp:printStmt.expressions) { isSimpleType(exp.type) == TRUE }	
assignStmt:stmt → assignment		

ifStmt:stmt → condition:expression ifStmts:stmt* elseStmts:stmt*	**Comprobamos que la condición sea de tipo 'integer'**
	ifStmt.condition.type *instanceof* IntegerType == TRUE
		Si alguna sentencia de la parte if o else retorna algo entonces lo marcamos como retornable
		for(Stmt s : ifStmt.ifstmts) {
		if(s.isReturnable()) { ifReturns = TRUE } }
		for(Stmt s : ifStmt.elsestmts) {
		if(s.isReturnable()) { ifReturns = TRUE } }

fromStmt:stmt → declarations:assignment* condition:expression stmts:stmt*	**Comprobamos que la condición del until sea de tipo 'integer'**
	fromStmt.condition.type*instanceof* IntegerType == TRUE
		Si alguna sentencia devuelve algo la marcamos como retornable
		for(Stmt s : fromStmt.stmts) {
		if(s.isReturnable()) { loopReturns=TRUE}}

procedureStmt:stmt → procedure		

returnStmt:stmt → returnInvoc	**Comprobamos que la feature no sea de tipo void**
	if(returnStmt.returnInvoc.expression.
	isPresent()){featureReturnType instanceof VoidType = FALSE}
	
	Comprobamos que si devuelve un tipo sea del mismo tipo que el tipo de retorno de la feature
	if(returnStmt.returnInvoc.expression.
	isPresent()){ if (!(featureReturnType instanceof VoidType)) {
	areSameType(featureReturnType, returnedType) == TRUE
	}}
	
	Si no hay expresión en el return, comprobamos que la feature sea de tipo void
		Lo definimos como retornable
		returnStmt.returnable = TRUE

	if(!returnStmt.returnInvoc.expression. isPresent()){featureReturnType instanceof VoidType == TRUE}	
assignment → left:expression right:expression	**Comprobamos que el lado izquierdo sea lvalue** assignment.left.lvalue == TRUE **Comprobamos que el lado izquierdo sea de tipo simple** isSimpleType(assignment.left.type) == TRUE **Comprobamos que ambos tipos sean iguales** areSameType (assignment.left.type, assignment.right.type) == TRUE	
intLiteral:expression → value:string		**Definimos el tipo** intLiteral.type = new IntegerType() **Indicamos que no es modificable** intLiteral.lvalue = FALSE
realLiteral:expression → value:string		**Definimos el tipo** realLiteral. type = new DoubleType() **Indicamos que no es modificable** realLiteral. lvalue = FALSE
charLiteral:expression → value:string		**Definimos el tipo** charLiteral. type = new CharacterType() **Indicamos que no es modificable** charLiteral. lvalue = FALSE
variable:expression → name:string		**Inferimos el tipo desde definition** variable.type = variable.definition.type **Indicamos que es modificable** variable.lvalue = TRUE
procedureExpression:expression → procedure	**Comprobamos que feature usada como expresión no sea Void**	**Inferimos el tipo**

	resultType instanceof VoidType == FALSE	resultType = procedureExpression.procedure. invocation.returntype **Definimos el tipo** procedureExpression.type = resultType **Indicamos que no es modificable** procedureExpression.lvalue = FALSE
arrayExpression:expression → array:expression index:expression	**Comprobamos que efectivamente sea de tipo array** arrayExpression.array.type *instanceof* ArrayType == TRUE **Comprobamos si el índice es de tipo 'integer'** arrayExpression.index.type *instanceof* IntegerType == TRUE	**Definimos el tipo** resultType = ((ArrayType) arrayBaseType).dataType **Indicamos que es o no modificable** isLvalue = arrayExpression.array.lvalue
structExpression:expression → struct:expression field:string	**Comprobamos que la base sea de tipo struct** structExpression.struct.type instanceof StructType == TRUE **Comprobamos que el campo exista en la definición de la deftuple** getField(((StructType) structBaseType).deftuple, structExpression.field) != NULL	**Inferimos el tipo** structExpression.type = fieldDefinition.type **Indicamos que es o no modificable** isLvalue = structExpression.struct.lvalue
minusExpression:expression → expression	**Comprobamos que la expresión sea de tipo 'integer' o 'double'** (expType instanceof IntegerType || expType instanceof DoubleType) == TRUE	**Inferimos el tipo** minusExpression.type = expType **Indicamos que no es modificable** minusExpression.lvalue = FALSE
notExpression:expression → expression	**Comprobamos que la expresión sea de tipo 'integer'** exprType instanceof IntegerType == TRUE	**Inferimos el tipo** notExpression.type = new IntegerType() **Indicamos que no es modificable** notExpression. lvalue = FALSE

cast:expression → dataType expression	**Comprobamos que el cast cumpla las reglas permitidas** isValidCast(cast.expression.type, cast.dataType) == TRUE	**Inferimos el tipo** cast.type = cast.dataType **Indicamos que no es modificable** cast. lvalue = FALSE
arithmeticExpression:expression → left:expression operator:string right:expression	**Comprobamos que ambas expresiones sean del mismo tipo** areSameType(leftType, rightType) == TRUE **Comprobamos si es 'mod', ya que en mod ambos tipos deben ser 'integer'** if (operator.equals("mod")){ leftType instanceof IntegerType && rightType instanceof IntegerType	**Inferimos el tipo** arithmeticExpression.type = resultType **Indicamos que no es modificable** arithmeticExpression. lvalue = FALSE
comparisonExpression:expression → left:expression operator:string right:expression	**Comprobamos que los operandos deben ser del mismo tipo, pudiendo ser 'integer' o 'double'** areSameType(leftType, rightType) && (leftType instanceof IntegerType || leftType instanceof DoubleType) == TRUE	**Inferimos el tipo** comparisonExpression.type = resultType **Indicamos que no es modificable** comparisonExpression. lvalue = FALSE
logicExpression:expression → left:expression operator:string right:expression	**Comprobamos que ambas expresiones sean de tipo 'integer'** leftType instanceof IntegerType && rightType instanceof IntegerType	**Inferimos el tipo** logicExpression.type = resultType **Indicamos que no es modificable** logicExpression.lvalue = FALSE
procedure → name:string expression*	**Comprobamos el mismo número de argugemntos que de parámetros** procedure.expressions.size() == procedure.invocation.params.size() **Comprobamos que los argumentos sean del mismo tipo** for(int i=0; i< arguments.size(); i++){ areSameType(arguments.get(i).getType(), parameters.get(i).getType())}	
integerType:dataType → ε		

doubleType:dataType → ε		
characterType:dataType → ε		
structType:dataType → name:string		
arrayType:dataType → size:string dataType	**Comprobamos que el tamaño sea un entero literal positivo** isPositiveIntegerLiteral(arrayType.size)	
voidType:dataType → ε		
errorType:dataType → ε		
create → idents:string*		
feature → name:string params:varDefinition* dataType? localBlock? doBlock	**Comprobamos que si existe tipo de retorno este sea simple** isSimpleType(calculatedReturnType) == TRUE **Comprobamos que los tipos de los parámetros sean simples** for (VarDefinition p : feature.params){ isSimpleType(p.type) } **Comprobamos que si el tipo de retorno es distinto de 'void' haya por lo menos un return** calculatedReturnType instanceof VoidType == FALSE	**Establecemos su tipo de retorno** feature.returntype = calculatedReturnType
returnInvoc → expression?		
localBlock → varDefinition*		

doBlock → stmt*		**Si alguna sentencia es retornable, lo marcamos como retornable**
		for(Stmt s : doBlock.stmts) {
		if (s.isReturnable()) {
		blockReturns = TRUE }}

global → globalTypes? varsTypes?		

globalTypes → deftuple*		
varsTypes → varDefinition*		
deftuple → name:string field*		
field → name:string type:dataType		
varDefinition → name:string type:dataType		

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

Auxiliary Functions

Name	Devuelve	Description
isSimpleType(DataType type)	boolean	Comprueba que el tipo es simple
areSameType(DataType typeA, DataType typeB)	boolean	Comprueba si dos tipos son del mismo tipo
isValidCast(DataType origin, DataType destination	boolean	Comprueba las reglas del cast
isPositiveIntegerLiteral(String size)	boolean	Comprueba si un string puede parsearse a integer
getField(Deftuple def, String name)	Field	Devuelve el campo de una deftuple dado el nombre del mismo