Paula Suárez Prieto, UO269745
Grupo de prácticas PL-02
Curso 2023-2024

Attribute Grammar - TypeChecking

Attributes

Symbol	Attribute Name	Java Type	Inherited/Synthesized	Description
Expression	lValue	Boolean	Synthesized	Define el tipo de la expresión, puede ser directo en las constantes o sintetizado en el resto de las expresiones
Expression	type	Type	Synthesized	Determina el tipo de una expresión.
FeatureDefinition	hasReturn	Boolean	Synthesized	Una `feature` que no sea de tipo `void` debería de tener al menos una sentencia `return`.
Sentence	owner	FunctionDefinition	Inherited	Enlace a la función en la que se encuentra la sentencia

Atributos de la fase anterior (Fase de identificación)

Algunos de estos atributos también se usan en esta especificación.

Symbol	Attribute Name	Java Type	Inherited/Synthesized	Description
VarDefinition	scope	ENUM { GLOBAL, LOCAL, PARAMETER }	Inherited	Registro del ámbito en el que se ha definido la variable: - Global: La variable se ha definido en el bloque `global` → `vars` - Local: La variable se ha definido en el bloque `local` dentro de un `feature` - Parameter: La variable es un parámetro de una función
Variable	definition	VarDefinition	Synthesized	Enlace a la definición de esta variable

FunctionDefinition	builder	boolean	Synthesized	True si se ha definido la función previamente en el bloque `create` (constructor). Se usará para verificar que la función puede ser llamada en la llamada `run`.
FunctionCallSent	definition	FunctionDefinition	Synthesized	Enlace a la definición de la función
FunctionCallExpr	definition	FunctionDefinition	Synthesized	Enlace a la definición de la función
FieldDefinition	fieldOwner	StructType	Inherited	Enlace a la struct en la que se define el campo
StructType	definition	StructDefinition	Synthesized	Enlace a la definición de la Struct
RunCall	definition	FunctionDefinition	Synthesized	Enlace a la definición de la función

Rules

Node	Predicates	Semantic Functions
program → name:string types:structDefinition* vars:varDefinition* builders:functionBuilder* features:functionDefinition* runCall:runCall		
runCall → name:string args:expression*	args.size == runCall.definition.params checkArgs(args, runCall.definition.params)	
structDefinition → name:structType fields:fieldDefinition*		
functionDefinition → name:string params:varDefinition* returnType:type? vars:varDefinition* sentences:sentence*	returnType ≠ ∅ AND returnType ≠ VoidType then { isPrimitive(returnType)}	sentences.forEach(s -> s.owner = functionDefinition)

	params.forEach(p ->
	  isPrimitive(p.tipo)
	
	returnType ≠ ∅ AND returnType ≠ VoidType then {
	  i functionDefinition.hasReturn == TRUE
	}
		functionDefinition.hasReturn = FALSE *
		
		* *Aclaración: hasReturn = FALSE se asigna antes de visitar a los hijos, el predicado hasReturn==TRUE se comprueba después de visitarlos*

fieldDefinition → name:string tipo:type		
varDefinition → name:string tipo:type		
functionBuilder → name:string		
functionCallSent:sentence → name:string args:expression*	args.size() == definition.params.size() checkArgs(args, definition.params)	
assignment:sentence → left:expression right:expression	left.lvalue == TRUE isPrimitive(left.type) checkSameType(left.type, right.type)	

loop:sentence → from:assignment* until:expression body:sentence*	until.type == INTEGER	from.forEach(a ->
		  a.owner = loop.owner)
		
		body.forEach(s ->
		  s.owner = loop.owner)

Paula Suárez Prieto, UO269745
Grupo de prácticas PL-02
Curso 2023-2024

ifElse:sentence → condition:expression trueBlock:sentence* falseBlock:sentence*	condition.type == INTEGER	trueBlock.forEach(s ->
		  s.owner = ifElse.owner)
		
		falseBlock.forEach(s ->
		  s.owner = ifElse.owner)

read:sentence → input:expression*	input.forEach(e ->
	  e.lValue == true)
	
	input.forEach(e ->
	  isPrimitive(e.type))	

print:sentence → op:string input:expression*	input.forEach(e -> isPrimitive(e.type))	

return:sentence → value:expression?	value == ∅ then {
	  return.owner.returnType == VOID
	}
	
	return.owner.returnType ≠ VOID then {
	  value ≠ ∅
	}
	
	value ≠ ∅ then {
	  checkSameType
	  (return.owner.returnType, value.type)
	}
		returnValue.owner.hasReturn = TRUE

intConstant:expression → value:string		intConstant.type = INTEGER

		intConstant.lValue = FALSE
realConstant:expression → value:string		realConstant.type = DOUBLE realConstant.lValue = FALSE
charConstant:expression → value:string		charConstant.type = CHARACTER charConstant.lValue = FALSE
variable:expression → name:string		variable.type = variable.definition.tipo variable.lValue = TRUE
castExpr:expression → castType:type value:expression	!checkSameType(castType, value.type) isPrimitive(castType) isPrimitive(value.type) checkCastType(castType, value.type)	castExpr.type = castType castExpr.lValue = FALSE

arithmeticExpr:expression → op1:expression operator:string op2:expression	isPrimitive(op1.type)
	operator == MOD then {
	op1.type == INTEGER
	}
	
	checkSameType(op1.type, op2.type)
		arithmeticExpr.type = op1.type
		
		arithmeticExpr.lValue = FALSE

logicalExpr:expression → op1:expression operator:string op2:expression	op1.type == INTEGER checkSameType(op1.type, op2.type)	logicalExpr.type = op1.type logicalExpr.lValue = FALSE

comparationExpr:expression → op1:expression operator:string op2:expression	(operator == '=' OR operator == '<>') THEN
	{
	isPrimitive(op1.type);
	} else {
	op1.type == INTEGER
	OR op1.type == DOUBLE
	}
	
	sameType(op1.type, op2.type)
		comparationExpr.type = INTEGER
		
		comparationExpr.lValue = FALSE

minusExpr:expression → op:expression	op.type == INTEGER OR op.type == DOUBLE	minusExpr.type = op.type minusExpr.lValue = FALSE
notExpr:expression → op:expression	op.type == INTEGER	notExpr.type = INTEGER notExpr.lValue = FALSE
functionCallExpr:expression → name:string args:expression*	functionCallExpr.definition.tipo!=VoidType functionCallExpr.definition.hasReturn == TRUE args.size == functionCallExpr.definition.params	functionCallExpr.type = functionCallExpr.definition.returnType functionCallExpr.lValue = FALSE

Paula Suárez Prieto, UO269745
Grupo de prácticas PL-02
Curso 2023-2024

	checkArgs(args, functionCallExpr.definition.params)	
fieldAccess:expression → root:expression field:string	root.type == StructType root.type.definition.fields[field] ≠ ∅	fieldAccess.type = root.type.definition.fields[field].tipo fieldAccess.lValue = false
arrayAccess:expression → array:expression index:expression	index.type == INTEGER array.type == ArrayType	arrayAccess.type = array.type.tipo arrayAccess.lValue = true
intType:type → ε		
doubleType:type → ε		
charType:type → ε		
voidType:type → ε		
structType:type → name:string		
arrayType:type → dimension:intConstant tipo:type		

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

Paula Suárez Prieto, UO269745
Grupo de prácticas PL-02
Curso 2023-2024

Auxiliary Functions

Function	Description
isPrimitive(Type t): boolean	Devuelve True si el tipo que se pasa por parámetro es un tipo simple (INTEGER, CHARACTER o DOUBLE)
checkSameType(Type t1, Type t2): boolean	Devuelve True si los tipos pasados por parámetro son del mismo tipo. Está pensado para ser utilizado con tipos simples (INTEGER, CHAR o DOUBLE)
checkCastType(Type castType, Type valueType): boolean	Devuelve True si el tipo al que se quiere castear y el valor que se quiere castear son compatibles. Las combinaciones válidas son las contempladas en la siguiente tabla figura: <table><tr><th></th><th>INTEGER</th><th>DOUBLE</th><th>CHARACTER</th></tr><tr><td>INTEGER</td><td></td><td>SI</td><td>SI</td></tr><tr><td>DOUBLE</td><td>SI</td><td></td><td></td></tr><tr><td>CHARACTER</td><td>SI</td><td></td><td></td></tr></table>
checkArgs(List<Expression> args, List<VarDefinition> params, String functionName): boolean	Recibe la lista de argumentos con la que se invoca a una función y la lista de parámetros de su definición. También recibe el nombre de la función de la cual se está haciendo la comprobación para lanzar errores con más detalle. Realiza dos comprobaciones: 1. Comprueba que la lista de argumentos es del mismo tamaño que la lista de parámetros 2. Si son del mismo tamaño, comprueba que los tipos de los argumentos coinciden con los tipos de los parámetros definidos en la función. Registra un error por cada argumento que no coincida. Devuelve True si se han pasado todas las comprobaciones, False en caso contrario.
getTypeName(Type t):String	Devuelve el nombre formateado del tipo que recibe por parámetro, si el tipo no es un tipo conocido o es null devuelve el String “INDEFINIDO”. Esta función se utiliza para detallar los errores.

Auxiliary Data Structures

Paula Suárez Prieto, UO269745
Grupo de prácticas PL-02
Curso 2023-2024

Symbol	Java Type	Description