## 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 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 | hasReturn | Boolean | Synthesized | Indica si la sentencia tiene algún 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 créate (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\* | checkArgs(args, runCall.definition.params) |  |
| structDefinition → name:structType fields:fieldDefinition\* |  |  |
| functionDefinition → name:string params:varDefinition\* returnType:type? vars:varDefinition\* sentences:sentence\* | If returnType ≠ ∅ { isPrimitive(returnType)  } | sentences.forEach(s -> s.owner = functionDefinition)  funcionDefinition.hasReturn = sentences.any(s -> s.hasReturn) |
| 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) | functionCallSent.hasReturn = FALSE |
| assignment:sentence → left:expression right:expression | left.lvalue == TRUE  isPrimitive(left.type)  sameType(left.type, right.type) | assignment.hasReturn = FALSE |
| 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)  body.forEach( s-> if (s.hasReturn) loop.hasReturn = TRUE) |
| ifElse:sentence → condition:expression trueBlock:sentence\* falseBlock:sentence\* | condition.t-ype == INTEGER | trueBlock.forEach(s -> s.owner = ifElse.owner)  falseBlock.forEach(s -> s.owner = ifElse.owner)  ifElse.hasReturn = trueBlock.any(s -> s.hasReturn)  ifElse.hasReturn = falseBlock.any(s -> s.hasReturn) |
| read:sentence → input:expression\* | input.all(e -> e.lValue == true)  input.all(e -> isPrimitive(e.type)) | read.hasReturn = FALSE |
| print:sentence → op:string input:expression\* | isPrimitive(input.type) == TRUE | print.hasReturn = FALSE |
| return:sentence → value:expression? | return.definition.returnType == void ? expression == ∅ : sameType (return.definition.returnType, value.type) | Return.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 = CHAR  charConstant.lValue = FALSE |
| variable:expression → name:string |  | variable.type = variable.definition.tipo  variable.lValue = TRUE |
| castExpr:expression → castType:type value:expression | isPrimitive(castType)  compatibleCast(castType, value.type) | castExpr.type = castType  castExpr.lValue = FALSE |
| arithmeticExpr:expression → op1:expression operator:string op2:expression | isPrimitive(op1.type)  sameType(op1.type, op2.type)  If(operator == MOD) {  op1.type == INTEGER  } | arithmeticExpr.type = op1.type  arithmeticExpr.lValue = FALSE |
| logicalExpr:expression → op1:expression operator:string op2:expression | op1.type == INTEGER  sameType(op1.type, op2.type) | LogicalExpr.type = INTEGER  LogicalExpr.lValue = FALSE |
| comparationExpr:expression → op1:expression operator:string op2:expression | 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\* | checkArgs(args,  functionCallExpr.definition.params)  args.size == functionCallExpr.definition.params | functionCallExpr.type =  functionCallExpr.definition.returnType  functionCallExpr.lValue = FALSE |
| fieldAccess:expression → root:expression field:string | root.type == StructType  root.type.definition.fields[field] ≠ ∅ | fieldAccess.type = field.type  fieldAccess.lValue = false |
| arrayAccess:expression → array:expression index:expression | index.type == INTEGER  array.type == ArrayType | arrayAccess.type = arrayType.type  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):  
⇒ ⇔ ≠ ∅ ∈ ∉ ∪ ∩ ⊂ ⊄ ∑ ∃ ∀

### Auxiliary Functions

|  |  |
| --- | --- |
| Function | Description |
| isPrimitive(Type t): boolean | Devuelve True si el tipo que se pasa por parámetro es un tipo simple (INTEGER, CHAR o DOUBLE) |
| sameType(Type t1, Type t2): boolean | Devuelve True si los tipos pasados por parámetro son del mismo tipo. |
| compatibleCast(Type t1, Type t2) | Devuelve True si el tipo al que se quiere castear y el valor que se quiere castear son compatibles. |
| checkArgs(List<Expression> args, List<VarDefinition> params) | Comprueba si la lista de args tiene la misma longitud que la lista de params y, además, si tienen la misma longitud comprueba que estos sean del mismo tipo. |
|  |  |

### Auxiliary Data Structures

|  |  |  |
| --- | --- | --- |
| Symbol | Java Type | Description |
|  |  |  |
|  |  |  |