Code Specification -UO294177

Functions	Code Templates
run⟦**program**⟧	run ⟦**program** → classDef global? create feature* runInvocation⟧ = metadata⟦program⟧ execute⟦runInvocation⟧ HALT features*.foreach(f=> execute⟦f⟧)
metadata⟦**program**⟧	metadata ⟦**program** → classDef global? create feature* runInvocation⟧ = '#source "{program.sourceFile}"' metadata⟦global⟧
f_1⟦**classDef**⟧	f_1⟦**classDef** → name:string⟧ =
execute⟦**runInvocation**⟧	execute⟦**runInvocation** → procedure⟧ = '#line {runInvocation.line}' procedure.expression*.foreach(arg => value⟦arg⟧) CALL {procedure.name}
execute⟦**assignment**⟧	execute⟦**assignment** → left:expression right:expression⟧ = '#line {assignment.line}' address⟦left⟧ value⟦right⟧ STORE<{left.type.suffix()}>
f_4⟦**procedure**⟧	f_4⟦**procedure** → name:string expression*⟧ =
f_5⟦**create**⟧	f_5⟦**create** → idents:string*⟧ =
execute⟦**feature**⟧	execute ⟦**feature** → name:string params:varDefinition* dataType? localBlock? doBlock⟧ = *// Etiqueta y número de línea* {name}: '#line {feature.line}' // Generamos la directiva #func {feature.name} *// Metadatos para Parametros y Variables Locales* params*.foreach(p => '#param {p.name} : {getMAPLTypeName(p.type)} (offset {p.address})') if(localBlock != null) localBlock.varDefinition*.foreach(l => '#local {l.name} : {getMAPLTypeName(l.type)} (offset {l.address})') *// Calculamos bytes de Locales y Parámetros* int localBytes = 0 if(localBlock != null && !localBlock.varDefinition*.isEmpty()) localBytes = -localBlock.varDefinition*.get(localBlock.varDefinition*.size()-1).address int paramBytes = params*.sum(p => p.type.numberOfBytes()) //4. Reservamos memoria para locales if(localBytes > 0) ENTER {localBytes}

	// Ejecutamos el cuerpo
	execute⟦doBlock⟧({localBytes}, {paramBytes})
	
	// Return implícito para void
	RET(returnBytes, localBytes, paramBytes)

execute⟦**returnInvoc**⟧	execute⟦**returnInvoc** → expression?⟧ (localBytes, paramBytes) =
	
	'#line {returnInvoc.line}'
	int returnBytes = 0
	if(expresión != null)
	  value ⟦expression⟧
	  returnBytes = expression.type.numberOfBytes()
	RET {returnBytes}, {localBytes}, {paramBytes}

f_8⟦**localBlock**⟧	f_8⟦**localBlock** → varDefinition*⟧ =
execute⟦**doBlock**⟧	execute⟦**doBlock** → stmt*⟧(localBytes, paramBytes) = stmt*.foreach(s => execute ⟦s⟧(localBytes, paramBytes))
metadata⟦**global**⟧	metadata⟦**global** → globalTypes? varsTypes?⟧ = metadata⟦globalTypes⟧ metadata⟦varsTypes⟧
metadata⟦**globalTypes**⟧	metadata⟦**globalTypes** → deftuple*⟧ = deftuple*.foreach(dt => metadata⟦dt⟧)
metadata⟦**varsTypes**⟧	metadata⟦**varsTypes** → varDefinition*⟧ = varDefinition*.foreach(vd => metadata⟦vd⟧)
metadata⟦**deftuple**⟧	metadata⟦**deftuple** → name:string field*⟧ = '#type {name} : {' + field*.map(f => metadata⟦f⟧).join(", ") + '}'
metadata⟦**field**⟧	metadata⟦**field** → name:string type:dataType⟧ =

metadata⟦**varDefinition**⟧	metadata⟦**varDefinition** → name:string type:dataType⟧ =
	
	if(varDefinition.scope == GLOBAL)
	   '#global {name} : {getMAPLTypeName(type)} offset {address}'

value⟦**expression**⟧	value⟦**intLiteral**:expression → value:string⟧ = PUSHI {value} value⟦**realLiteral**:expression → value:string⟧ = PUSHF {value} value⟦**charLiteral**:expression → value:string⟧ = PUSHB {(int)value.charAt(1)} value⟦**variable**:expression → name:string⟧ = address⟦variable⟧ LOAD<{variable.type.suffix()}> value⟦**procedureExpression**:expression → procedure⟧ = procedure.expression*.foreach(arg => value⟦arg⟧)

	CALL {procedure.name} value⟦**arrayExpression**:expression → array:expression index:expression⟧ = address⟦arrayExpression⟧ LOAD<{arrayExpression.type.suffix()}> value⟦**structExpression**:expression → struct:expression field:string⟧ = address⟦structExpression⟧ LOAD<{structExpression.type.suffix()}> value⟦**minusExpression**:expression → expression⟧ = if(expression.type instanceof IntegerType) PUSHI 0 else PUSHF 0.0 value ⟦expression⟧ SUB<{expression.type.suffix()}> value⟦**notExpression**:expression → expression⟧ = value⟦expression⟧ NOT value⟦**cast**:expression → dataType expression⟧ = value⟦expression⟧ CONVERT<{expression.type.suffix()}2{dataType.suffix()}> value⟦**arithmeticExpression**:expression → left:expression operator:string right:expression⟧ = value⟦left⟧ value⟦right⟧ {getMAPLOperator(operator, arithmeticExpression.type)}<{arithmeticExpression.type.suffix()}> value⟦**comparisonExpression**:expression → left:expression operator:string right:expression⟧ = value⟦left⟧ value⟦right⟧ {getMAPLOperator(operator, left.type)} value⟦**logicExpression**:expression → left:expression operator:string right:expression⟧ = value⟦left⟧ value⟦right⟧ {getMAPLOperator(operator, expression.type)}
address⟦**expression**⟧	address⟦**intLiteral**:expression → value:string⟧ = address⟦**realLiteral**:expression → value:string⟧ = address⟦**charLiteral**:expression → value:string⟧ = address⟦**variable**:expression → name:string⟧ =

	if(variable.definition.scope == GLOBAL) PUSHA else PUSH BP PUSHI {variable.definition.address} ADD address⟦**procedureExpression**:expression → procedure⟧ = address⟦**arrayExpression**:expression → array:expression index:expression⟧ = address⟦array⟧ value⟦index⟧ PUSHI {array.type.dataType.numberOfBytes()} MUL ADD address⟦**structExpression**:expression → struct:expression field:string⟧ = address ⟦struct⟧ PUSHI {struct.type.deftuple.getField(field).offset} ADD address⟦**minusExpression**:expression → expression⟧ = address⟦**notExpression**:expression → expression⟧ = value⟦**cast**:expression → dataType expression⟧ = value⟦**arithmeticExpression**:expression → left:expression operator:string right:expression⟧ = value⟦**comparisonExpression**:expression → left:expression operator:string right:expression⟧ = value⟦**logicExpression**:expression → left:expression operator:string right:expression⟧ =
execute⟦**stmt**⟧	execute⟦**readStmt**:stmt → expression*⟧ = execute⟦**printStmt**:stmt → expression* format:string⟧ = '#line {printStmt.line}' expression*.foreach(exp => value⟦exp⟧ OUT<{exp.type.suffix()}>) if(format == “ln”) PUSHB 10 // ASCII nueva línea OUTB execute⟦**assignStmt**:stmt → assignment⟧ = execute⟦assignment⟧ execute⟦**ifStmt**:stmt → condition:expression ifStmts:stmt* elseStmts:stmt*⟧ (localBytes, paramBytes) = string elseLabel = util.nextLabel() string endIfLabel = util.nextLabel()

	'#line {condition.line}'
	value⟦condition⟧
	JZ {elseLabel} // Salta a ELSE si la condición es falsa (0)
	ifStmts*.foreach(s => execute⟦s⟧(localBytes, paramBytes))
	JMP {endIfLabel}
	{elseLabel}:
	elseStmts*.foreach(s => execute⟦s⟧(localBytes, paramBytes))
	{endIfLabel}:
	
	execute⟦**fromStmt**:stmt → declarations:assignment* condition:expression stmts:stmt*⟧
	(localBytes, paramBytes)=
	
	string loopLabel = util.nextLabel()
	string endLoopLabel = util.nextLabel()
	declarations*.foreach(decl => execute⟦decl⟧)
	{loopLabel}:
	'#line {condition.line}'
	value⟦condition⟧ //Salimos si la condición es falsa (0)
	stmt*.foreach(s => execute⟦s⟧(localBytes, paramBytes))
	JMP {loopLabel} // Saltamos hacia atras a comprobar la condición
	{endLoopLabel}:
	
	execute⟦**procedureStmt**:stmt → procedure⟧ =
	
	'#line {procedureStmt.line}'
	procedure.expression*.foreach(arg => value⟦arg⟧) // Evaluamos los argumentos
	CALL {procedure.name}
	if(procedure.invocation.returnType != null) // Si devuelve
	    POP<{procedure.invocation.returnType.suffix()}>
	{procedure.invocation.returnType.numberOfBytes()}
	
	execute⟦**returnStmt**:stmt → returnInvoc⟧ (localBytes, paramBytes) =
	
	'#line {returnStmt.line}'
	execute⟦returnInvoc⟧ ({localBytes}, {paramBytes})

f_{18}⟦**dataType**⟧	f_{18}⟦**integerType**:dataType → ε⟧ = f_{18}⟦**doubleType**:dataType → ε⟧ = f_{18}⟦**characterType**:dataType → ε⟧ = f_{18}⟦**structType**:dataType → name:string⟧ = f_{18}⟦**arrayType**:dataType → size:string dataType⟧ = f_{18}⟦**voidType**:dataType → ε⟧ = f_{18}⟦**errorType**:dataType → ε⟧ =

Auxiliary Functions

Function	Description
getMAPLTypeSuffix(DataType type)	Devuelve el sufijo para MAPL dado un tipo
getMAPLTypeString(DataType type)	Dado un tipo devuelve el nombre que conoce para comentario MAPL

getConversionInstruction(DataType from, DataType to)	Realiza la operación de conversión dados dos tipos
arithmetic(String operator, DataType type)	Realiza la operación aritmética en MAPL dado un operador y un tipo
comparison(String operator, DataType operandType)	Realiza la operación de comparación en MAPL dado un operador y un tipo
logical(String operator)	Realiza la operación de lógica en MAPL dado un operador and o or