Code Specification

Functions	Code Templates
generateProgram⟦**program**⟧	generateProgram⟦**program** → classDeclaration runStatement⟧ = #SOURCE "source_file.mapl" ' Programa principal execute[runStatement] HALT generateClass[classDeclaration]

generateClass⟦**classDeclaration**⟧	generateClass⟦**classDeclaration** → name:string globalSection? createSection featureSection*⟧ =
	
	if globalSection present:
	  generateGlobals[globalSection]
	for each featureSection:
	  generateFunction[featureSection]

generateGlobals⟦**globalSection**⟧	generateGlobals⟦**globalSection** → typesSection? varSection?⟧ =
	
	if typesSection present:
	  generateTypes[typesSection]
	if varSection present:
	  generateVars[varSection]

generateTypes⟦**typesSection**⟧	generateTypes⟦**typesSection** → structDeclaration*⟧ = for each structDeclaration: generateStruct[structDeclaration]
generateVars ⟦**varSection**⟧	generateVars⟦**varSection** → variableDeclaration*⟧ = for each variableDeclaration: for each identifier in

	variableDeclaration.identifiers: #GLOBAL identifier : variableDeclaration.type
generateStruct⟦**structDeclaration**⟧	generateStruct⟦**structDeclaration** → name:string structField*⟧ = #TYPE name : (for each structField: structField.name : structField.type)
generateConstructors⟦**createSection**⟧	generateConstructos⟦**createSection** → string*⟧ =

generateFunction⟦**featureSection**⟧	generateFunction⟦**featureSection** → name:string args? type? localSection? statement*⟧ =
	
	#FUNC name
	
	if args presente:
	
	generateArgs[args]
	
	if type presente:
	
	#RET type
	
	else:
	
	#RET VOID
	
	if localSection presente:
	
	generateLocals[localSection]
	
	name:
	
	if size_de_variables_locales > 0:
	
	ENTER size_de_variables_locales

	for each statement: execute[statement] if (type es void) y no hay return explícito: RET 0, size_de_variables_locales, size_de_params
generateLocals[[**localSection**]]	generateLocals[[**localSection** → variableDeclaration*]] = for each variableDeclaration: for each identifier in variableDeclaration.identifiers: #LOCAL identifier : variableDeclaration.type
generateArgs [[**args**]]	generateArgs[[**args** → arg*]] = for each arg: #PARAM arg.name : arg.type
value[[**expression**]]	value[intLiteral:expression → name:string] = pushi name value[realConstant:expression → name:string] = pushf name value[charConstant:expression → name:string] = pushb ascii(name) value[functionCallExp:expression → name:string expresiones:expression*] = for each expr in expresiones: value[expr] call name

	value[arrayAcces:expression → exp2:expression exp3:expression] = address[arrayAcces] load<type> value[variableAcces:expression → name:string] = address[variableAcces] load<type> value[restaUnaria:expression → exp2:expression] = value[exp2] neg<type> value[parentesis:expression → exp2:expression] = value[exp2] value[relacional:expression → exp2:expression name:string exp3:expression] = value[exp2] value[exp3] <relational_op><type_suffix> value[negacion:expression → exp2:expression] = value[exp2] not

	value[cast:expression → tipoCast:type exp2:expression] = value[exp2] <conversion_instr> // (e.g., i2f, f2i, b2i...) value[arithmetic:expression → exp2:expression name:string exp3:expression] = value[exp2] value[exp3] <arithmetic_op><type_suffix> value[booleanExp:expression → exp2:expression name:string exp3:expression] = value[exp2] value[exp3] <logical_op><type_suffix> value[structFieldAcces:expression → exp2:expression name:string] = address[structFieldAcces] load<type>
address[expression]	address[arrayAcces:expression → exp2:expression exp3:expression] = address[exp2] value[exp3] pushi size_of_element muli addi

	address[variableAcces:expression → name:string] =
	if variable is global:
	pusha address
	else if variable is local or param:
	pusha bp
	pushi offset
	addi / subi
	
	address[parentesis:expression → exp2:expression] =
	address[exp2] // solo si exp2 es lvalue
	
	address[cast:expression → tipoCast:type exp2:expression] =
	address[exp2] // solo si exp2 es lvalue
	
	address[structFieldAcces:expression → exp2:expression name:string] =
	address[exp2]
	pushi offset_of_field
	addi

execute[[**statement**]]	execute[assignment:statement → left:expression right:expression] = #line n address[left] value[right] store<type> execute[print:statement → expression*] =

	#line n
	for each expr:
	value[expr]
	out<type>
	
	execute[println:statement → expression*] =
	#line n
	for each expr:
	value[expr]
	out<type>
	pushb 10
	outb
	
	execute[read:statement → expression*] =
	#line n
	for each expr:
	address[expr]
	in<type>
	store<type>
	
	execute[bloqueif:statement → expression st2:statement* st3:statement*] =
	#line n
	value[expression]
	if st3 is not empty:
	jz elseLabel
	else:

	jz endLabel
	
	for each stmt in st2:
	execute[stmt]
	
	if st3 is not empty:
	jmp endLabel
	elseLabel:
	for each stmt in st3:
	execute[stmt]
	
	endLabel:
	
	execute[loopFrom:statement → st1:statement* expression body:statement*] =
	#line n
	for each stmt in st1:
	execute[stmt]
	
	loopLabel:
	value[expression]
	jnz endLabel
	for each stmt in body:
	execute[stmt]
	jmp loopLabel
	endLabel:

	execute[return:statement → expression?] =
	#line n
	if expression present:
	value[expression]
	ret returnSize, localsSize, paramsSize
	
	execute[functionCallStatement:statement → name:string expression*] =
	#line n
	for each expr:
	value[expr]
	call name
	if returns value:
	pop
	
	execute[runStatement:statement → name:string expression*] =
	#line n
	for each expr:
	value[expr]
	call name
	if returns value:
	pop

declareType[[**type**]]	**declareType[intType:type → name:string]** =
	// No code generated (primitive type)
	declareType[doubleType:type → name:string] =
	// No code generated (primitive type)
	declareType[characterType:type → name:string] =
	// No code generated (primitive type)
	declareType[identType:type → name:string] =
	// References #TYPE definition

	declareType[arraytype:type → intValue:int type2:type] =
	// Array type defined implicitly in variable declarations
	declareType[errorType:type → name:string] =
	// No code generated
	declareType[voidType:type → name:string] =
	// No code generated (used in functions)

declare⟦**declaration**⟧	declare[variableDeclaration:declaration → identifiers:string* type] =
	
	for each identifier:
	
	if scope == 0:
	
	#GLOBAL identifier : type
	
	else:
	
	#LOCAL identifier : type
	
	declare[structField:declaration → name:string type] =
	
	// No se genera código. Los campos se emiten dentro de #TYPE en generateStruct
	
	declare[arg:declaration → name:string type] =
	
	#PARAM name : type