Code Specification

Functions	Code Templates
run⟦*⟧	**run⟦program → name:string definition* functionCreation* functionDefinition* run⟧ =** #source {file} #line {end.line} “call main” “halt” execute[definition] execute[functionDefinition*] execute[run]
value⟦*⟧	**value⟦variable:expression → name:string⟧ =** address[variable] loadx **value⟦intLiteral:expression → intValue:int⟧ =** “pushi “ + intLiteral.getIntValue() **value⟦realLiteral:expression → floatValue:float⟧ =** “pushf “ + realLiteral.getFloatValue() **value⟦charLiteral:expression → name:string⟧ =** # por ser String el tipo if (charLiteral.getName().eq(“\n”)) “pushb “ + (int) ‘\n’ else “pushb “ + (int) charLiteral.getName()[1] **value⟦functionCallExpression:expression → name:string expression*⟧ =** value[functionCallExpression.expressions()] “call “ + functionCallExpression.getName() **value⟦functionCallStatement:statement → name:string expression*⟧ =** value[functionCallStatement.expressions()] “call “ + functionCallStatement.getName()

	value⟦structAccess:expression → expr:expression name:string⟧ = address[structAccess] loadx **value⟦arrayAccess:expression → left:expression right:expression⟧ =** address[arrayAccess] loadx **value⟦cast:expression → castType:type expression⟧ =** value[cast.getExpression] suffix 2 suffix **value⟦arithmeticBinary:expression → left:expression operator:string right:expression⟧ =** value[arithmeticBinary.getLeft()] value[arithmeticBinary.getRight()] transformOperator addSuffix(arithmeticBinary.type) **value⟦arithmeticUnary:expression → operator:string expr:expression⟧ =** value[arithmeticUnary.getExpr()] “-” **value⟦logicBinary:expression → left:expression operator:string right:expression⟧ =** value[logicBinary.getLeft()] value[logicBinary.getRight()] transformOperator **value⟦logicUnary:expression → operator:string expr:expression⟧ =** value[logicUnary.getExpr()] “not”

	value⟦relationalBinary:expression → left:expression operator:string right:expression⟧ = value[relationalBinary.getLeft()] value[relationalBinary.getRight()] transformOperator

execute⟦*⟧	**execute⟦functionDefinition→ ...⟧** =
	
	functionDefinition.getName() + “:”
	cte1 ← type size
	cte2 ← locals size
	cte3 ← parameters size
	cts = [cte1, cte2, cte3]
	execute[functionDefinition.statements(), cts]
	if (cte1 == 0)
	  “ret %s, %s, %s”.format(cts)
	
	execute⟦structDefinition→ ...⟧ =
	
	execute⟦varDefinition→ ...⟧ =
	
	execute⟦Run→ ...⟧ =
	
	“main:”
	“call: “ + run.getName()
	“ret 0,0,0”
	
	execute⟦print:statement → expression*⟧ =
	
	if (empty)
	  “pushb 0”
	  “outb”
	
	else
	  line(print)
	  print.getExpressions().forEach(
	    value[expression]
	    outx
	  )

execute⟦println:statement → expression*⟧ =

```
if (empty)
  “out \n”
else
  line(println)
  println.getExpressions().forEach(
    value[expression]
    outx
    “out \n”
  )
```

execute⟦read:statement → expression*⟧ =

```
read.getExpressions().forEach(
  address[expression]
  inx
  storex
)
```

execute⟦functionCallStatement:statement → name:string expression*⟧ =

```
line(functionCallStatement)
value[functionCallStatement]
if (function.type.isPresent)
  popx
```

execute⟦assignment:statement → left:expression right:expression⟧ =

```
line(assignment)
address[assignment.getLeft()]
value[assignment.getRight()]
storex
```

execute⟦conditional:statement → expression ifStatements:statement* elseStatements:statement*⟧ =

```
jzLabel = “label “ + labelCount++; jmpLabel = “label “ + labelCount++
line(conditional)
value[conditional.getExpression]
“jz “ + jzLabel
execute[conditional.ifStatements()]
“jmp “ + jmpLabel
jzLabel + “:”
execute[conditional.elseStatements()]
jmpLabel + “:”
```

	execute⟦loop:statement → fromStatements:statement* expression loopStatements:statement*⟧ = exitLabel = “label “ + labelCount++ loopLabel = “label “ + labelCount++ line(loop) execute[loop.fromStatements] loopLabel + “:” value[loop.getExpression()] “jnz “ + exitLabel execute[loop.loopStatements] “jmp “ + loopLabel exitLabel + “:” **execute⟦return:statement → expression?⟧ =** int[] ctes = (int[]) param; line(returnValue) if (ctes[0] != 0) value[return.getExpression()] “ret %s, %s, %s”.format(ctes)
address⟦*⟧	**value⟦variable:expression → name:string⟧ =** if (variable.definition.scope == 1) “push “ + variable.definition.address else “push bp” “push “ + variable.definition.address “addi” **value⟦structAccess:expression → expr:expression name:string⟧ =** address[structAccess.getExpr()] “pushi “ + structAccess.getFieldDefinition().getAddress() “addi “ **value⟦arrayAccess:expression → left:expression right:expression⟧ =** address[arrayAccess.getLeft()] value[arrayAccess.getRight()] “pushi “ + arrayAccess.getType().getSize() “muli” “addi”

Loadx → load dependiendo del tipo, (i,f,b)

‘’’ Idem en instructionx ‘’’

suffix 2 suffix → castea consiguiendo el sujido de cada tipo

transformOperator → pasa el operador de minieiffel a mapl

addSuffix(type) → devuelve sufijo para el tipo