Code Generation Specification

Paula Suárez Prieto – UO269745

Diseño de Lenguajes de Programación, curso 2023-2024

Grupo PL-02

Functions	Code Templates
run⟦program⟧	run⟦**program** → name:string types:structDefinition* vars:varDefinition* builders:functionBuilder* features:functionDefinition* runCall:runCall⟧ = metadata[[program]] execute⟦runCall⟧ **HALT** generate[[features]]
metadata⟦program⟧	metadata⟦**program** → name:string types:structDefinition* vars:varDefinition* builders:functionBuilder* features:functionDefinition* runCall:runCall⟧ = #SOURCE {source_file} ‘Clase: {name} ‘Declaraciones globales metadata⟦types$_i$⟧ metadata⟦vars$_i$⟧ metadata⟦builders$_i$⟧
execute⟦runCall⟧	execute⟦**runCall** → name:string args:expression*⟧ = value[[args$_i$]] **CALL** name if(runCall.owner.returnType != VOID) **POP** maplSuffix(runCall.owner.returnType)

metadata⟦structDefinition⟧	metadata⟦**structDefinition** → name:structType fields:fieldDefinition*⟧ =
	
	#type {name}: {
	metadata⟦fields$_i$⟧
	}

generate⟦functionDefinition⟧	generate⟦**functionDefinition** → name:string params:varDefinition* returnType:type? vars:varDefinition* sentences:sentence*⟧ = #FUNCTION {name} #ret {maplType(returnType)} {name}: metadata[[params$_i$]] metadata[[vars$_i$]] int bytesLocals = getVarsSize(vars) if(bytesLocals > 0) **ENTER** bytesLocals

	int bytesParams = getVarsSize(params) int bytesReturn = maplTypeSize(returnType) execute[[$sentences_i$]] if (bytesReturn == 0) **RET** bytesReturn, bytesLocals, bytesParams
metadata⟦fieldDefinition⟧	metadata⟦**fieldDefinition** → name:string tipo:type⟧ = #FIELD {name}: {maplType(tipo)}
metadata⟦functionBuilder⟧	metadata⟦**functionBuilder** → name:string⟧ = '**builder {name}

metadata⟦varDefinition⟧	metadata⟦**varDefinition** → name:string tipo:type⟧ =
	
	 if varDefinition.scope == GLOBAL
	  #global {name}: {maplType(tipo)}
	 else if varDefinition.scope == LOCAL
	  #local {name}: {maplType(tipo)}
	 else if varDefinition.scope == PARAMETER
	  #param {name}: {maplType(tipo)}

execute⟦sentence⟧	execute ⟦**functionCallSent**:sentence → name:string args:expression*⟧ =
	
	 #LINE {functionCallSent}
	 value[[$args_i$]]
	 CALL name
	 if(functionCallSent.owner.returnType !=VoidType)
	  POP maplSuffix(
	    functionCallSent.owner.returnType)

	execute ⟦**assignment**:sentence → left:expression right:expression⟧ = #LINE {assignment} address[[left]] value[[right]] **STORE** maplSuffix(left.type)
	execute ⟦**loop**:sentence → from:assignment* until:expression body:sentence*⟧ = labelCount++; String condLabel = formatLabel("untilcond_", labelCount) String endLabel = formatLabel("untilend_", labelCount) #LINE {loop} 'from execute[[$from_i$]] condLabel:

	value[[until]] **JNZ** endLabel ‘loop body execute[[$body_i$]] **JMP** condLabel endLabel:
	execute ⟦**ifElse**:sentence → condition:expression trueBlock:sentence* falseBlock:sentence*⟧ = labelCount++; String elseLabel = formatLabel(“else_”, labelCount) String endLabel = formatLabel(“endif_”, labelCount) #LINE {ifElse} ‘condition value[[condition]] **JZ** elseLabel ‘if block execute[[$trueBlock_i$]] **JMP** endLabel ‘else block elseLabel: execute[[$falseBlock_i$]] endLabel:
	execute ⟦**read**:sentence → input:expression*⟧ = #LINE {input} address[[$expression_i$]] **IN** maplSuffix(input[0].type) **STORE** maplSuffix(input[0].type)

	execute ⟦**print**:sentence → op:string input:expression*⟧ =
	
	#LINE (input.start.line)
	input*.forEach(Expression e ->
	    value[[e]]
	    OUT maplSuffix(e.type)
	)
	if(op==”println”){
	    PUSHB 10
	    OUTB
	}

	execute ⟦**return**:sentence → value:expression?⟧ = #LINE (end.line) value[[value]] int bytesLocals = getVarsSize(return.owner.vars) int bytesParams = getVarsSize(return.owner.params)

	int bytesReturn = maplTypeSize(return.owner.returnType) **RET** bytesReturn, bytesLocals, bytesParams
address⟦expression⟧	address ⟦**intConstant**:expression → value:string⟧ = #Error
	address ⟦**realConstant**:expression → value:string⟧ = #Error
	address ⟦**charConstant**:expression → value:string⟧ = #Error

	address ⟦**variable**:expression → name:string⟧ =
	
	if variable.definition.scope == GLOBAL
	    PUSHA {variable.definition.address}
	else
	    PUSH BP
	    PUSH {variable.definition.address}
	    ADDI

	address ⟦**castExpr**:expression → castType:type value:expression⟧ = #Error
	address ⟦**arithmeticExpr**:expression → op1:expression operator:string op2:expression⟧ = #Error
	address ⟦**logicalExpr**:expression → op1:expression operator:string op2:expression⟧ = #Error
	address ⟦**comparationExpr**:expression → op1:expression operator:string op2:expression⟧ = #Error
	address ⟦**minusExpr**:expression → op:expression⟧ = #Error
	address ⟦**notExpr**:expression → op:expression⟧ = #Error

	address ⟦**functionCallExpr**:expression → name:string args:expression*⟧ = #Error
	address ⟦**fieldAccess**:expression → root:expression field:string⟧ = address[[root]] **PUSHI** getFieldOffset(root.type, field) **ADDI**
	address ⟦**arrayAccess**:expression → array:expression index:expression⟧ = address[[array]] value[[index]] **PUSHI** maplTypeSize(arrayAccess.type) **MULI** **ADDI**
value⟦expression⟧	value ⟦**intConstant**:expression → value:string⟧ = **PUSHI** value
	value ⟦**realConstant**:expression → value:string⟧ = **PUSHF** value
	value ⟦**charConstant**:expression → value:string⟧ if(value == "\n") **PUSHB** 10 else **PUSHB** value.charAt(1)
	value ⟦**variable**:expression → name:string⟧ = address[[variable]] **LOAD** maplTypeSuffix(variable.definition.type)
	value ⟦**castExpr**:expression → castType:type value:expression⟧ = value[[value]] String castInstr = maplSuffix(castType.type) + "2" + maplSuffix(value.type) If (castInstructions.contains(castInstr)) **castInstr**
	value ⟦**arithmeticExpr**:expression → op1:expression operator:string op2:expression⟧ = value[[op1]] value[[op2]] **maplOperator(operator, op2.type)**

	value ⟦**logicalExpr**:expression → op1:expression operator:string op2:expression⟧ = value[[op1]] value[[op2]] **maplOperator(operator)**
	value ⟦**comparationExpr**:expression → op1:expression operator:string op2:expression⟧ = value[[op1]] value[[op2]] **maplOperator(operator, op1.type)**
	value ⟦**minusExpr**:expression → op:expression⟧ = value[[op]] **PUSHI -1** **MULI**
	value ⟦**notExpr**:expression → op:expression⟧ = value[[op]] **NOT**
	value ⟦**functionCallExpr**:expression → name:string args:expression*⟧ = value[[args_i]] **CALL** name
	value ⟦**fieldAccess**:expression → root:expression field:string⟧ = address[[fieldAccess]] **LOAD** maplSuffix(fieldAccess.type)
	value ⟦**arrayAccess**:expression → array:expression index:expression⟧ = address[[arrayAccess]] **LOAD** maplSuffix(arrayAccess.type)
f_9⟦type⟧	f_9⟦**intType**:type → ε⟧ =
	f_9⟦**doubleType**:type → ε⟧ =
	f_9⟦**charType**:type → ε⟧ =
	f_9⟦**voidType**:type → ε⟧ =

	f_9⟦**structType**:type → name:string⟧ =
	f_9⟦**arrayType**:type → dimension:intConstant tipo:type⟧ =

Auxiliar functions

Estas funciones están definidas en un fichero de utilidad llamado **`MaplUtils.java`**.

Método	Descripción

maplType(Type t)**: String**	Retorna el nombre del tipo que se le pasa por parámetro.
	`switch (t) {`
	`        case IntType i -> "int";`
	`        case DoubleType f -> "float";`
	`        case CharType c -> "char";`
	`        case StructType s -> s.getName();`
	`        case ArrayType a -> a.getDimension().getValue() + " * " + maplType(a.getTipo());`
	`        case VoidType v -> "void";`
	`        default -> throw new IllegalArgumentException("Unrecognized type");`
	`    };`

maplTypeSize(Type t)**: int**	Retorna el tamaño que ocupa en MAPL el tipo que se pasa por parámetro
	`switch (t) {`
	`        case IntType i -> 2;`
	`        case DoubleType f -> 4;`
	`        case CharType c -> 1;`
	`        case StructType s -> getStructSize(s);`
	`        case ArrayType a -> Integer.valueOf(a.getDimension().getValue()) * maplTypeSize(a.getTipo());`
	`        case VoidType v -> 0;`
	`        default -> throw new IllegalArgumentException("Unrecognized type");`
	`    };`

maplSuffix(Type t)**: String**	Retorna el sufijo de MAPL correspondiente al tipo que se pasa por parámetro `switch (t) { case IntType i -> "I"; case DoubleType f -> "F"; case CharType c -> "B"; default -> throw new IllegalArgumentException("Unrecognized type"); };`
maplOperator(String op)**: String**	Recorre un Map con todos los operadores reconocidos en el lenguaje (MAP_TRANSLATION) y devuelve el operador de MAPL correspondiente con el String que se pasa por parámetro.
maplOperator(String op, Type type)**: String**	Retorna la instrucción MAPL correspondiente al operador (recorre MAP_TRANSLATION) que se pasa por parámetro junto con el sufijo correspondiente al Type.
getVarsSize(List<VarDefinition> vars)**: int**	Devuelve el tamaño total de la lista de variables que se pasa por parámetro
getStructSize(StructType t)**: int**	Devuelve el tamaño total del Struct que se pasa por parámetro
getFieldOffset(StructType struct, String field)**: int**	Devuelve el desplazamiento (*offset*) de un campo del struct. Lanza una excepción si el struct no contiene el campo.
formatLabel(String labelName, int count)**: String**	Devuelve una etiqueta formateada, recibe el nombre de la etiqueta y un contador. `return labelName + String.format("%04d", count);`

Estructuras de datos utilizadas

Estructura de datos	Descripción
Set<String> castInstructions	Instrucciones de MAPL correspondientes a las operaciones de cast permitidas en el lenguaje. En este caso:

	`HashSet<String>(Set.of("I2F", "F2I", "I2B", "B2I"))`

Map<String, String> MAP_TRANSLATION	Recoge las instrucciones MAPL correspondientes a todos los operadores permitidos en el lenguaje.
	`MAP_TRANSLATION = Map.ofEntries(`
	`Map.entry("+", "ADD"),`
	`Map.entry("-", "SUB"),`
	`Map.entry("*", "MUL"),`
	`Map.entry("/", "DIV"),`
	`Map.entry("mod", "MOD"),`
	`Map.entry("=", "EQ"),`
	`Map.entry("<>", "NE"),`
	`Map.entry("<", "LT"),`
	`Map.entry("<=", "LE"),`
	`Map.entry(">", "GT"),`
	`Map.entry(">=", "GE"),`
	`Map.entry("and", "AND"),`
	`Map.entry("or", "OR"),`
	`Map.entry("not", "NOT")`
	`);`

Explicación de las funciones de código utilizadas

- **Run** → Ejecuta el programa
- **Execute** → Ejecuta las sentencias y las instrucciones RunCall
- **Metadata** → Registra los metadatos del programa y de las definiciones (definición de variables, definición de structs y sus campos, definición de constructores) que contiene el programa
- **Generate** → Genera el código de definición de una función
- **Value** → Apila el valor de una expresión
- **Address** → Apila la dirección de una expresión