CodeGenerator.java

Archivo copiado desde `src/main/java/codegen/CodeGenerator.java`.

```java
// src/main/java/codegen/CodeGenerator.java
package codegen;

import parser.ast.*;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.LinkedHashMap;

public class CodeGenerator implements ASTVisitor<Void> {
    private StringBuilder sb = new StringBuilder();
    private Map<String, Integer> varOffsets = new HashMap<>();
    private Map<String, String> stringLiterals = new LinkedHashMap<>(); //
Mantiene orden de inserción
    private int nextOffset = 0;
    private int labelCounter = 0;

    private String getUniqueLabel(String prefix) {
        return prefix + (labelCounter++);
    }


    public String generate(ASTNode root) {
        // Primera pasada: recolectar todos los strings
        collectStrings(root);

        // Generar sección .data con todos los strings
        generateDataSection();

        // Generar sección .text
        sb.append("section .text\n");
        sb.append("global _start\n\n");
        sb.append("_start:\n");
        sb.append("    mov rbp, rsp\n\n");

        // Segunda pasada: generar código
        root.accept(this);

        sb.append("_start_end:\n");
        sb.append("    mov rax, 60\n");
        sb.append("    mov rdi, 0\n");
        sb.append("    syscall\n");
        return sb.toString();
    }
```

```
    private void collectStrings(ASTNode node) {
        if (node instanceof parser.ast.StringNode) {
            parser.ast.StringNode stringNode = (parser.ast.StringNode) node;
            String value = stringNode.getValue();
            String label = "str" + Math.abs(value.hashCode());
            stringLiterals.put(label, value);
        } else if (node instanceof ProgNode) {
            ProgNode progNode = (ProgNode) node;
            for (ASTNode stmt : progNode.getStatements()) {
                collectStrings(stmt);
            }
        } else if (node instanceof AssignNode) {
            AssignNode assignNode = (AssignNode) node;
            collectStrings(assignNode.getExpr());
        } else if (node instanceof ExprStmtNode) {
            ExprStmtNode exprStmt = (ExprStmtNode) node;
            collectStrings(exprStmt.getExpr());
        } else if (node instanceof BinaryOpNode) {
            BinaryOpNode binOp = (BinaryOpNode) node;
            collectStrings(binOp.getLeft());
            collectStrings(binOp.getRight());
        } else if (node instanceof UnaryOpNode) {
            UnaryOpNode unaryOp = (UnaryOpNode) node;
            collectStrings(unaryOp.getOperand());
        } else if (node instanceof FuncCallNode) {
            FuncCallNode funcCall = (FuncCallNode) node;
            for (ASTNode arg : funcCall.getArgs()) {
                collectStrings(arg);
            }
        } else if (node instanceof ForNode) {
            ForNode forNode = (ForNode) node;
            collectStrings(forNode.getIterable());
            for (ASTNode stmt : forNode.getBody()) {
                collectStrings(stmt);
            }
        } else if (node instanceof WhileNode) {
            WhileNode whileNode = (WhileNode) node;
            collectStrings(whileNode.getCondition());
            for (ASTNode stmt : whileNode.getBody()) {
                collectStrings(stmt);
            }
        } else if (node instanceof IfNode) {
            IfNode ifNode = (IfNode) node;
            collectStrings(ifNode.getCondition());
            for (ASTNode stmt : ifNode.getThenBody()) {
                collectStrings(stmt);
            }
            for (IfNode.ElifClause elifClause : ifNode.getElifClauses()) {
                collectStrings(elifClause.getCondition());
                for (ASTNode stmt : elifClause.getBody()) {
                    collectStrings(stmt);
                }
            }
            if (ifNode.getElseBody() != null) {
```

```
                for (ASTNode stmt : ifNode.getElseBody()) {
                    collectStrings(stmt);
                }
            }
        } else if (node instanceof RangeNode) {
            RangeNode rangeNode = (RangeNode) node;
            for (ASTNode arg : rangeNode.getArgs()) {
                collectStrings(arg);
            }
        }
        // VarRefNode, IntNode y BoolNode no contienen strings, no necesitan
procesamiento
    }

    private void generateDataSection() {
        if (!stringLiterals.isEmpty()) {
            sb.append("section .data\n");
            for (Map.Entry<String, String> entry : stringLiterals.entrySet()) {
                sb.append(entry.getKey()).append(": db
\"").append(entry.getValue()).append("\", 0x0A, 0\n");
            }
            sb.append("\n");
        }
    }

    public Void visit(ProgNode node) {
        for (ASTNode stmt : node.getStatements()) {
            stmt.accept(this);
        }
        return null;
    }

    public Void visit(AssignNode node) {
        node.getExpr().accept(this);
        varOffsets.computeIfAbsent(node.getName(), k -> nextOffset -= 8);
        int offset = varOffsets.get(node.getName());
        sb.append("    pop rax\n");
        sb.append("    mov [rbp").append(offset).append("], rax\n\n");
        return null;
    }

    public Void visit(ExprStmtNode node) {
        node.getExpr().accept(this);
        sb.append("    pop rax\n\n");
        return null;
    }

    public Void visit(BinaryOpNode node) {
        node.getLeft().accept(this);
        node.getRight().accept(this);
        sb.append("    pop rbx\n");
        sb.append("    pop rax\n");
        switch (node.getOp()) {
            case "+": sb.append("    add rax, rbx\n"); break;
```

```
            case "-": sb.append("    sub rax, rbx\n"); break;
            case "*": sb.append("    imul rax, rbx\n"); break;
            case "/": sb.append("    cqo\n    idiv rbx\n"); break;
            case "%": sb.append("    cqo\n    idiv rbx\n    mov rax, rdx\n");
break;
            case "==": sb.append("    cmp rax, rbx\n    sete al\n    movzx rax,
al\n"); break;
            case "!=": sb.append("    cmp rax, rbx\n    setne al\n    movzx rax,
al\n"); break;
            case ">": sb.append("    cmp rax, rbx\n    setg al\n    movzx rax,
al\n"); break;
            case "<": sb.append("    cmp rax, rbx\n    setl al\n    movzx rax,
al\n"); break;
            case ">=": sb.append("    cmp rax, rbx\n    setge al\n    movzx rax,
al\n"); break;
            case "<=": sb.append("    cmp rax, rbx\n    setle al\n    movzx rax,
al\n"); break;
            case "**": sb.append("    ; potencia placeholder\n"); break;
            case "and":
                sb.append("    ; and: si rax es 0, resultado es 0; si no,
resultado es rbx\n");
                sb.append("    test rax, rax\n");
                sb.append("    cmovz rbx, rax\n");
                sb.append("    mov rax, rbx\n");
                break;
            case "or":
                sb.append("    ; or: si rax no es 0, resultado es rax; si no,
resultado es rbx\n");
                sb.append("    test rax, rax\n");
                sb.append("    cmovnz rax, rax\n");
                sb.append("    cmovz rax, rbx\n");
                break;
        }
        sb.append("    push rax\n\n");
        return null;
    }

    public Void visit(UnaryOpNode node) {
        node.getOperand().accept(this);
        sb.append("    pop rax\n");
        switch (node.getOp()) {
            case "+":
                // +x no hace nada, el valor ya está en rax
                break;
            case "-":
                sb.append("    neg rax\n");
                break;
            case "not":
                sb.append("    ; not: si rax es 0, resultado es 1; si no,
resultado es 0\n");
                sb.append("    test rax, rax\n");
                sb.append("    setz al\n");
                sb.append("    movzx rax, al\n");
                break;
```

```
        }
        sb.append("    push rax\n\n");
        return null;
    }

    @Override
    public Void visit(IntNode node) {
        sb.append("    push ").append(node.getValue()).append("\n\n");
        return null;
    }

    @Override
    public Void visit(BoolNode node) {
        // True = 1, False = 0
        int value = node.getValue() ? 1 : 0;
        sb.append("    push ").append(value).append("\n\n");
        return null;
    }

    @Override
    public Void visit(VarRefNode node) {
        String name = node.getName();
        if (!varOffsets.containsKey(name)) {
            throw new RuntimeException("Variable no definida: " + name);
        }
        int offset = varOffsets.get(name);
        sb.append("    mov rax, [rbp").append(offset).append("]\n");
        sb.append("    push rax\n\n");
        return null;
    }

    public Void visit(FuncCallNode node) {
        // placeholder si llegas a usar llamadas
        List<ASTNode> args = node.getArgs();
        for (ASTNode arg : args) {
            arg.accept(this);
        }
        sb.append("    call ").append(node.getName()).append("\n");
        sb.append("    push rax\n\n");
        return null;
    }

    @Override
    public Void visit(parser.ast.StringNode node) {
        System.out.println("[CodeGen] visit(StringNode): \"" + node.getValue() +
"\"");

        // El string ya fue recolectado en la primera pasada
        String label = "str" + Math.abs(node.getValue().hashCode());

        // Solo generar el código para cargar la dirección del string
        sb.append("    lea rdi, [rel ").append(label).append("]\n");
        sb.append("    push rdi\n\n");
        return null;
```

```
    }

    @Override
    public Void visit(ForNode node) {
        // Generar código para el ciclo for
        String loopStart = getUniqueLabel("loop_start_");
        String loopEnd = getUniqueLabel("loop_end_");

        // Asegurar que la variable de iteración tenga espacio en la pila
        varOffsets.computeIfAbsent(node.getVariable(), k -> nextOffset -= 8);
        int varOffset = varOffsets.get(node.getVariable());

        if (node.getIterable() instanceof RangeNode) {
            RangeNode range = (RangeNode) node.getIterable();
            int start = range.getStart();
            int stop = range.getStop();
            int step = range.getStep();

            // Inicializar variable de iteración
            sb.append("    ; Inicializar ciclo for
").append(node.getVariable()).append("\n");
            sb.append("    mov rax, ").append(start).append("\n");
            sb.append("    mov [rbp").append(varOffset).append("], rax\n\n");

            // Etiqueta de inicio del ciclo
            sb.append(loopStart).append(":\n");

            // Condición del ciclo
            sb.append("    ; Verificar condición del ciclo\n");
            sb.append("    mov rax, [rbp").append(varOffset).append("]\n");
            sb.append("    cmp rax, ").append(stop).append("\n");
            if (step > 0) {
                sb.append("    jge ").append(loopEnd).append("\n\n");
            } else {
                sb.append("    jle ").append(loopEnd).append("\n\n");
            }

            // Ejecutar cuerpo del ciclo
            sb.append("    ; Cuerpo del ciclo\n");
            for (ASTNode stmt : node.getBody()) {
                stmt.accept(this);
            }

            // Incrementar variable de iteración
            sb.append("    ; Incrementar variable de iteración\n");
            sb.append("    mov rax, [rbp").append(varOffset).append("]\n");
            if (step == 1) {
                sb.append("    inc rax\n");
            } else if (step == -1) {
                sb.append("    dec rax\n");
            } else {
                sb.append("    add rax, ").append(step).append("\n");
            }
            sb.append("    mov [rbp").append(varOffset).append("], rax\n");
```

```
            sb.append("    jmp ").append(loopStart).append("\n\n");

            // Etiqueta de fin del ciclo
            sb.append(loopEnd).append(":\n");
            sb.append("    ; Fin del ciclo for\n\n");
        } else {
            // Por ahora solo soportamos range()
            throw new RuntimeException("Solo se soporta range() en ciclos for");
        }

        return null;
    }

    @Override
    public Void visit(WhileNode node) {
        // Generar código para el ciclo while
        String loopStart = getUniqueLabel("while_start_");
        String loopEnd = getUniqueLabel("while_end_");

        // Etiqueta de inicio del ciclo
        sb.append(loopStart).append(":\n");

        // Evaluar condición del while
        sb.append("    ; Evaluar condición del while\n");
        node.getCondition().accept(this);
        sb.append("    pop rax\n");
        sb.append("    test rax, rax\n");
        sb.append("    jz ").append(loopEnd).append("\n\n");

        // Ejecutar cuerpo del ciclo
        sb.append("    ; Cuerpo del ciclo while\n");
        for (ASTNode stmt : node.getBody()) {
            stmt.accept(this);
        }

        // Volver al inicio del ciclo
        sb.append("    jmp ").append(loopStart).append("\n\n");

        // Etiqueta de fin del ciclo
        sb.append(loopEnd).append(":\n");
        sb.append("    ; Fin del ciclo while\n\n");

        return null;
    }

    @Override
    public Void visit(IfNode node) {
        // Generar código para la estructura if-elif-else
        String endLabel = getUniqueLabel("if_end_");

        // Evaluar condición del if
        sb.append("    ; Evaluar condición del if\n");
        node.getCondition().accept(this);
        sb.append("    pop rax\n");
```

```
        sb.append("    test rax, rax\n");

        // Si hay elif o else, saltar al siguiente bloque
        String nextLabel = null;
        if (!node.getElifClauses().isEmpty() || node.getElseBody() != null) {
            nextLabel = getUniqueLabel("elif_else_");
            sb.append("    jz ").append(nextLabel).append("\n\n");
        } else {
            // Si no hay elif ni else, saltar al final
            sb.append("    jz ").append(endLabel).append("\n\n");
        }

        // Cuerpo del then
        sb.append("    ; Cuerpo del if (then)\n");
        for (ASTNode stmt : node.getThenBody()) {
            stmt.accept(this);
        }
        sb.append("    jmp ").append(endLabel).append("\n\n");

        // Procesar cláusulas elif
        for (int i = 0; i < node.getElifClauses().size(); i++) {
            if (nextLabel != null) {
                sb.append(nextLabel).append(":\n");
            }

            IfNode.ElifClause elifClause = node.getElifClauses().get(i);

            // Evaluar condición del elif
            sb.append("    ; Evaluar condición del elif ").append(i +
1).append("\n");
            elifClause.getCondition().accept(this);
            sb.append("    pop rax\n");
            sb.append("    test rax, rax\n");

            // Si hay más elif o else, saltar al siguiente bloque
            if (i < node.getElifClauses().size() - 1 || node.getElseBody() !=
null) {
                nextLabel = getUniqueLabel("elif_else_");
                sb.append("    jz ").append(nextLabel).append("\n\n");
            } else {
                sb.append("    jz ").append(endLabel).append("\n\n");
            }

            // Cuerpo del elif
            sb.append("    ; Cuerpo del elif ").append(i + 1).append("\n");
            for (ASTNode stmt : elifClause.getBody()) {
                stmt.accept(this);
            }
            sb.append("    jmp ").append(endLabel).append("\n\n");
        }

        // Procesar cláusula else
        if (node.getElseBody() != null) {
            if (nextLabel != null) {
```

```
                sb.append(nextLabel).append(":\n");
            }

            sb.append("    ; Cuerpo del else\n");
            for (ASTNode stmt : node.getElseBody()) {
                stmt.accept(this);
            }
        }

        // Etiqueta de fin del if
        sb.append(endLabel).append(":\n");
        sb.append("    ; Fin del if\n\n");

        return null;
    }

    @Override
    public Void visit(RangeNode node) {
        // RangeNode normalmente se procesa dentro de ForNode
        // Si se visita directamente, podríamos generar un array o similar
        // Por simplicidad, no hacer nada aquí por ahora
        return null;
    }
}
```