

Universidad de Ingenieria y Tecnologia

Facultad de Computacion

Compilador de C a x86-64

Proyecto Final

CS3402 - Compiladores

Integrante 1

Integrante 2

Integrante 3

Noviembre 2025

Índice

1. Introduccion	4
1.1. Objetivos	4
1.2. Alcance	4
2. Arquitectura del Compilador	4
2.1. Estructura de Archivos	5
3. Analisis Lexico (Scanner)	5
3.1. Clase Scanner	5
3.2. Tipos de Tokens	6
3.3. Manejo de Numeros	6
4. Analisis Sintactico (Parser)	7
4.1. Clase Parser	7
4.2. Jerarquia del AST	7
4.3. Precedencia de Operadores	8
4.4. Soporte para Typedef	8
5. Analisis Semantico	8
5.1. Clase SemanticAnalyzer	9
5.2. Sistema de Tipos	9
5.3. Tabla de Simbolos	10
6. Generacion de Codigo	10
6.1. Clase CodeGenerator	10
6.2. Convencion de Llamadas (System V ABI)	11
6.3. Estructura del Stack Frame	11
6.4. Generacion de Prologo y Epilogo	11
6.5. Generacion de Expresiones Binarias	12
6.6. Generacion de Expresiones Ternarias	12
6.7. Generacion de Estructuras de Control	13
7. Optimizaciones	14
7.1. Constant Folding (Plegado de Constantes)	14
7.2. Dead Code Elimination (Eliminacion de Codigo Muerto)	15
8. Extensiones Implementadas	15
8.1. Expresiones Ternarias	15
8.2. Alias de Tipo (typedef)	16
9. Casos de Prueba	16
9.1. Tests de Funciones (3 casos)	16
9.2. Tests de Implementacion Base (5 casos)	17
9.3. Tests de Extensiones (5 casos)	17
9.4. Tests de Optimizacion (5 casos)	17

10.Ejemplo de Codigo Generado	18
10.1. Codigo Fuente	18
10.2. Codigo Ensamblador Generado	18
11.Instrucciones de Uso	19
11.1. Compilacion del Compilador	19
11.2. Ejecucion	20
12.Conclusiones	20
12.1. Trabajo Futuro	20
13.Referencias	21

1. Introducción

El presente documento describe el diseño e implementación de un compilador que traduce un subconjunto del lenguaje C a código ensamblador x86-64. El compilador fue desarrollado en C++ siguiendo una arquitectura modular que comprende las fases clásicas de compilación: análisis léxico, análisis sintáctico, análisis semántico y generación de código.

1.1. Objetivos

- Implementar un compilador funcional para un subconjunto de C
- Generar código ensamblador x86-64 ejecutable
- Aplicar técnicas de optimización en tiempo de compilación
- Soportar tipos de datos numéricos adicionales (float, long, unsigned int)
- Implementar extensiones del lenguaje (expresiones ternarias, typedef)

1.2. Alcance

El compilador soporta las siguientes características del lenguaje C:

- Declaración de variables con tipos int, long, unsigned int y float
- Operaciones aritméticas, relacionales y lógicas
- Estructuras de control: if-else, while, for
- Definición y llamada de funciones, incluyendo recursión
- Expresiones ternarias (operador ?:)
- Alias de tipos mediante typedef

2. Arquitectura del Compilador

El compilador sigue una arquitectura de pipeline con cuatro fases principales, cada una implementada como un módulo independiente.

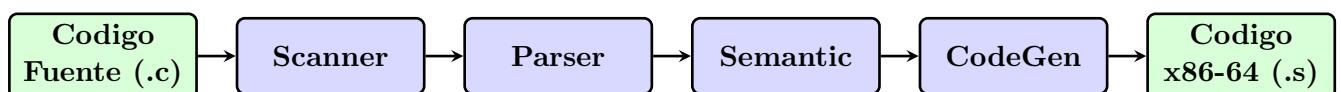


Figura 1: Pipeline del compilador

2.1. Estructura de Archivos

Listing 1: Estructura del proyecto

```

1 Final/
2   |-- src/
3   |   |-- main.cpp          # Punto de entrada
4   |   |-- scanner.cpp       # Analizador lexico
5   |   |-- parser.cpp        # Analizador sintactico
6   |   |-- semantic.cpp      # Analizador semantico
7   |   |-- codegen.cpp       # Generador de codigo
8   |   |-- ast.cpp           # Nodos del AST
9   |   |-- symboltable.cpp   # Tabla de simbolos
10  |-- include/
11  |   |-- scanner.h
12  |   |-- parser.h
13  |   |-- ast.h
14  |   |-- visitor.h
15  |   |-- semantic.h
16  |   |-- codegen.h
17  |   |-- symboltable.h
18  |-- tests/                 # 18 casos de prueba
19  |-- visualizer/           # Bonus: visualizador web

```

3. Análisis Lexico (Scanner)

El analizador lexico convierte el código fuente en una secuencia de tokens. Esta implementado en la clase `Scanner` utilizando un enfoque de análisis carácter por carácter.

3.1. Clase Scanner

Listing 2: Estructura principal del Scanner

```

1 class Scanner {
2 private:
3     std::string input;
4     size_t pos;
5     int line, column;
6     char current;
7
8     void advance();
9     char peek(int offset = 1);
10    void skipWhitespace();
11    void skipComment();
12    void skipPreprocessor();
13    Token scanNumber();
14    Token scanIdentifier();
15    Token scanString();
16
17 public:

```

```

18     Scanner(const std::string& src);
19     Token nextToken();
20     std::vector<Token> tokenize();
21 }

```

3.2. Tipos de Tokens

El scanner reconoce los siguientes tipos de tokens:

Categoría	Tokens	Ejemplos
Palabras clave	int, long, float, unsigned, void if, else, while, for, return typedef	int x; if (x >0) typedef int entero;
Literales	NUM, FLOAT_LIT, STRING_LIT	42, 3.14, "holá"
Identificadores	ID	variable, función
Operadores	+, -, *, /, %, ==, !=, i, i, i=, i=	a + b
Delimitadores	(,), {, }, [,], ;, ,	func(a, b)

Cuadro 1: Tipos de tokens reconocidos

3.3. Manejo de Numeros

El scanner detecta automaticamente el tipo de literal numerico:

Listing 3: Escaneo de literales numericos

```

1 Token Scanner::scanNumber() {
2     std::string num;
3     bool isFloat = false;
4
5     while (std::isdigit(current)) {
6         num += current;
7         advance();
8     }
9
10    // Detectar punto decimal
11    if (current == '.' && std::isdigit(peek())) {
12        isFloat = true;
13        num += current;
14        advance();
15        while (std::isdigit(current)) {
16            num += current;
17            advance();
18        }
19    }
20
21    return Token(isFloat ? TokenType::FLOAT_LIT
22                  : TokenType::NUM, num);
23 }

```

4. Análisis Sintáctico (Parser)

El analizador sintáctico implementa un parser recursivo descendente que construye un Árbol de Sintaxis Abstracta (AST).

4.1. Clase Parser

Listing 4: Estructura del Parser

```

1  class Parser {
2  private:
3      std::vector<Token> tokens;
4      size_t current;
5      std::map<std::string, DataType> typeAliases;
6
7      Token peek(int offset = 0);
8      Token advance();
9      bool match(TokenType type);
10     bool check(TokenType type);
11     void expect(TokenType type, const std::string& msg);
12
13     DataType parseType();
14     void parseTypedef();
15
16     std::unique_ptr<Expr> parseExpression();
17     std::unique_ptr<Expr> parseTernary();
18     std::unique_ptr<Expr> parseLogicalOr();
19
20 public:
21     std::unique_ptr<Program> parse();
22 };

```

4.2. Jerarquía del AST

El AST utiliza el patrón Visitor para permitir múltiples recorridos:

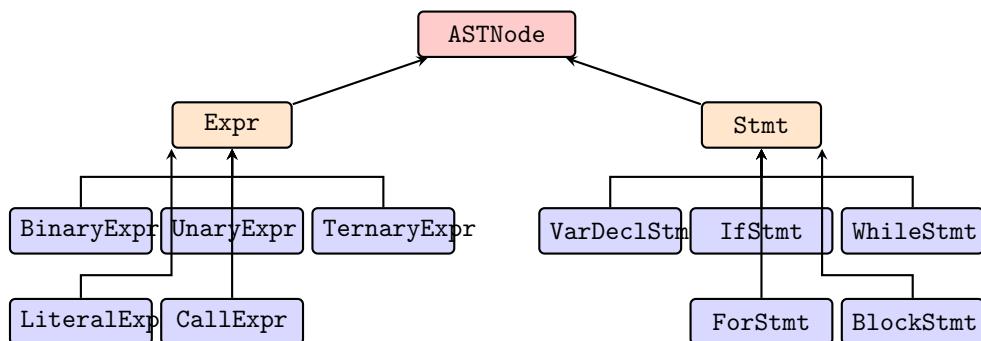


Figura 2: Jerarquía de clases del AST

4.3. Precedencia de Operadores

La precedencia se implementa mediante funciones de parsing separadas:

Nivel	Operadores	Funcion
1 (menor)	?: (ternario)	parseTernary()
2	—	parseLogicalOr()
3	&&	parseLogicalAnd()
4	==, !=	parseEquality()
5	i, i, i=, i=	parseRelational()
6	+, -	parseAdditive()
7	*, /, %	parseMultiplicative()
8 (mayor)	-!, ! (unarios)	parseUnary()

Cuadro 2: Niveles de precedencia

4.4. Soporte para Typedef

El parser mantiene un mapa de alias de tipos:

Listing 5: Implementacion de typedef

```

1 void Parser::parseTypedef() {
2     expect(TokenType::TYPEDEF, "Expected 'typedef'");
3     DataType baseType = parseType();
4
5     std::string aliasName = peek().lexeme;
6     expect(TokenType::ID, "Expected type alias name");
7     expect(TokenType::SEMICOLON, "Expected ';' ");
8
9     // Almacenar el alias
10    typeAliases[aliasName] = baseType;
11 }
12
13 bool Parser::isTypeToken() {
14     if (check(TokenType::INT) || check(TokenType::LONG) ||
15         check(TokenType::FLOAT) || check(TokenType::UNSIGNED)) {
16         return true;
17     }
18     // Verificar si es un alias de typedef
19     if (check(TokenType::ID)) {
20         return typeAliases.find(peek().lexeme)
21             != typeAliases.end();
22     }
23     return false;
24 }
```

5. Analisis Semantico

El analizador semantico verifica la correccion del programa mediante el patron Visitor.

5.1. Clase SemanticAnalyzer

Listing 6: Estructura del analizador semantico

```

1 class SemanticAnalyzer : public Visitor {
2 private:
3     SymbolTable& symbolTable;
4     DataType currentFunctionReturnType;
5     bool hasErrors;
6     std::string errors;
7
8     bool areTypesCompatible(DataType expected,
9                             DataType actual);
10    DataType getCommonType(DataType t1, DataType t2);
11    void error(const std::string& message);
12
13 public:
14     void visit(BinaryExpr* node) override;
15     void visit(TernaryExpr* node) override;
16     void visit(CallExpr* node) override;
17     void visit(VarDeclStmt* node) override;
18     void visit(FunctionDecl* node) override;
19 };

```

5.2. Sistema de Tipos

El compilador soporta cuatro tipos numericos con promocion automatica:

Listing 7: Promocion de tipos

```

1 DataType SemanticAnalyzer::getCommonType(DataType t1,
2                                         DataType t2) {
3     if (t1 == t2) return t1;
4
5     // Float domina sobre todos
6     if (t1 == DataType::FLOAT || t2 == DataType::FLOAT)
7         return DataType::FLOAT;
8
9     // Long domina sobre Int
10    if (t1 == DataType::LONG || t2 == DataType::LONG)
11        return DataType::LONG;
12
13    // UInt con Int -> Long (para evitar overflow)
14    if ((t1 == DataType::UINT && t2 == DataType::INT) ||
15        (t1 == DataType::INT && t2 == DataType::UINT))
16        return DataType::LONG;
17
18    return t1;
19 }

```

5.3. Tabla de Simbolos

La tabla de simbolos maneja ambitos anidados mediante una pila de scopes:

Listing 8: Estructura de la tabla de simbolos

```

1 struct Symbol {
2     std::string name;
3     DataType type;
4     bool isMutable;
5     bool isParameter;
6     int offset; // Offset en el stack frame
7     std::vector<int> arrayDimensions;
8 };
9
10 class SymbolTable {
11 private:
12     std::vector<std::map<std::string, Symbol>> scopes;
13     std::map<std::string, FunctionSymbol> functions;
14     int currentOffset;
15
16 public:
17     void enterScope();
18     void exitScope();
19     bool declareVariable(const std::string& name,
20                         const Symbol& sym);
21     Symbol* lookup(const std::string& name);
22     int allocateStackSpace(int size);
23 };

```

6. Generacion de Codigo

El generador de codigo produce ensamblador x86-64 en sintaxis Intel, siguiendo la convencion de llamadas System V ABI.

6.1. Clase CodeGenerator

Listing 9: Estructura del generador de codigo

```

1 class CodeGenerator : public Visitor {
2 private:
3     SymbolTable& symbolTable;
4     std::stringstream code;
5     std::stringstream dataSection;
6     int labelCounter;
7     int stringCounter;
8     bool enableConstantFolding;
9     bool enableDeadCodeElimination;
10
11     std::string newLabel();
12     void emit(const std::string& instruction);

```

```

13     void emitLabel(const std::string& label);
14     void generatePrologue(const std::string& func,
15                           int stackSize);
16     void generateEpilogue();
17
18 public:
19     void visit(BinaryExpr* node) override;
20     void visit(CallExpr* node) override;
21     void visit(IfStmt* node) override;
22
23     std::string getCode() const;
24 };

```

6.2. Convencion de Llamadas (System V ABI)

Registro	Uso
rax	Valor de retorno, acumulador
rbx	Registro temporal (callee-saved)
rdi, rsi, rdx, rcx, r8, r9	Argumentos 1-6
rbp	Base pointer (marco de pila)
rsp	Stack pointer
xmm0-xmm1	Operaciones de punto flotante

Cuadro 3: Uso de registros x86-64

6.3. Estructura del Stack Frame

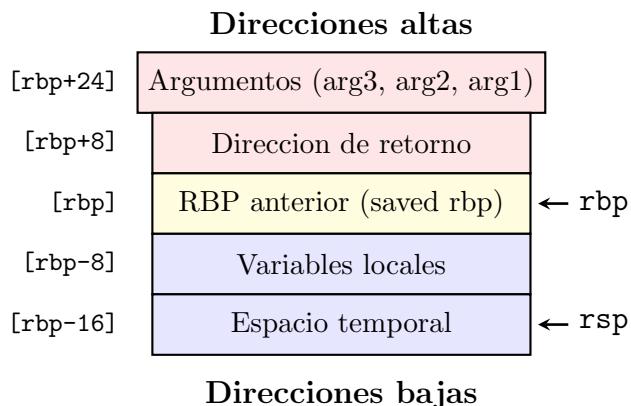


Figura 3: Estructura del stack frame en x86-64

6.4. Generacion de Prologo y Epilogo

Listing 10: Generacion del prologo de funcion

```

1     void CodeGenerator::generatePrologue(const std::string& func,
```

```

2         int stackSize) {
3             emitLabel(func);
4             emit("push rbp");
5             emit("mov rbp, rsp");
6
7             // Alinear stack a 16 bytes (requerido por ABI)
8             int alignedSize = ((stackSize + 15) & ~15);
9             if (alignedSize > 0) {
10                 emit("sub rsp, " + std::to_string(alignedSize));
11             }
12         }
13
14     void CodeGenerator::generateEpilogue() {
15         emit("mov rsp, rbp");
16         emit("pop rbp");
17         emit("ret");
18     }

```

6.5. Generacion de Expresiones Binarias

Listing 11: Generacion de operaciones aritmeticas

```

1 void CodeGenerator::visit(BinaryExpr* node) {
2     // Evaluar operandos
3     node->left->accept(this);
4     emit("push rax"); // Guardar operando izquierdo
5
6     node->right->accept(this);
7     emit("mov rbx, rax"); // Operando derecho en rbx
8     emit("pop rax"); // Operando izquierdo en rax
9
10    // Generar operacion
11    if (node->op == "+") {
12        emit("add rax, rbx");
13    } else if (node->op == "-") {
14        emit("sub rax, rbx");
15    } else if (node->op == "*") {
16        emit("imul rax, rbx");
17    } else if (node->op == "/") {
18        emit("cqo"); // Sign extend rax to rdx:rax
19        emit("idiv rbx"); // Resultado en rax
20    } else if (node->op == "%") {
21        emit("cqo");
22        emit("idiv rbx");
23        emit("mov rax, rdx"); // Resto en rdx
24    }
25}

```

6.6. Generacion de Expresiones Ternarias

Listing 12: Generacion del operador ternario

```

1 void CodeGenerator::visit(TernaryExpr* node) {
2     std::string falseLabel = newLabel();
3     std::string endLabel = newLabel();
4
5     // Evaluar condicion
6     node->condition->accept(this);
7     emit("test rax, rax");
8     emit("jz " + falseLabel);
9
10    // Rama verdadera
11    node->trueExpr->accept(this);
12    emit("jmp " + endLabel);
13
14    // Rama falsa
15    emitLabel(falseLabel);
16    node->falseExpr->accept(this);
17
18    emitLabel(endLabel);
19 }
```

6.7. Generacion de Estructuras de Control

Listing 13: Generacion de if-else

```

1 void CodeGenerator::visit(IfStmt* node) {
2     std::string elseLabel = newLabel();
3     std::string endLabel = newLabel();
4
5     node->condition->accept(this);
6     emit("test rax, rax");
7
8     if (node->elseBranch) {
9         emit("jz " + elseLabel);
10        node->thenBranch->accept(this);
11        emit("jmp " + endLabel);
12        emitLabel(elseLabel);
13        node->elseBranch->accept(this);
14        emitLabel(endLabel);
15    } else {
16        emit("jz " + endLabel);
17        node->thenBranch->accept(this);
18        emitLabel(endLabel);
19    }
20 }
```

Listing 14: Generacion de bucle while

```

1 void CodeGenerator::visit(WhileStmt* node) {
2     std::string startLabel = newLabel();
3     std::string endLabel = newLabel();
```

```

4     emitLabel(startLabel);
5     node->condition->accept(this);
6     emit("test rax, rax");
7     emit("jz " + endLabel);
8
9
10    node->body->accept(this);
11    emit("jmp " + startLabel);
12
13    emitLabel(endLabel);
14 }
```

7. Optimizaciones

El compilador implementa dos optimizaciones principales aplicadas durante la generacion de codigo.

7.1. Constant Folding (Plegado de Constantes)

Esta optimizacion evalua expresiones constantes en tiempo de compilacion:

Listing 15: Implementacion de constant folding

```

1 void CodeGenerator::visit(BinaryExpr* node) {
2     if (enableConstantFolding) {
3         auto leftLit = dynamic_cast<LiteralExpr*>(
4             node->left.get());
5         auto rightLit = dynamic_cast<LiteralExpr*>(
6             node->right.get());
7
8         if (leftLit && rightLit &&
9             node->exprType != DataType::FLOAT) {
10            long long leftVal = std::stoll(leftLit->value);
11            long long rightVal = std::stoll(rightLit->value);
12            long long result = 0;
13
14            if (node->op == "+")
15                result = leftVal + rightVal;
16            else if (node->op == "-")
17                result = leftVal - rightVal;
18            else if (node->op == "*")
19                result = leftVal * rightVal;
20            else if (node->op == "/" && rightVal != 0)
21                result = leftVal / rightVal;
22
23            // Emitir resultado constante directamente
24            emit("mov rax, " + std::to_string(result));
25            return;
26        }
27    }
28 }
```

Ejemplo:

```
1 int x = 5 + 10 * 2; // Se evalua como 25 en compilacion
```

Código generado optimizado:

```
1 mov rax, 25 ; En lugar de multiples operaciones
```

7.2. Dead Code Elimination (Eliminación de Código Muerto)

El compilador detecta y omite código que nunca se ejecutara:

Listing 16: Eliminación de código muerto en if

```
1 void CodeGenerator::visit(IfStmt* node) {
2     // Verificar si la condición es constante
3     if (auto lit = dynamic_cast<LiteralExpr*>(
4         node->condition.get())))
5     {
6         if (lit->value == "0") {
7             // Condición siempre falsa: solo generar else
8             if (node->elseBranch) {
9                 node->elseBranch->accept(this);
10            }
11        }
12    }
13    return; // Omitir rama then
14 }
15 }
```

Ejemplo:

```
1 if (0) {
2     z = 100; // Este código nunca se ejecuta
3 }
```

El compilador no genera código para el bloque interno.

8. Extensiones Implementadas

8.1. Expresiones Ternarias

El operador ternario ?: permite expresiones condicionales en linea:

Listing 17: Ejemplo de expresión ternaria

```
1 int max = (x > y) ? x : y;
2 int result = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
```

La implementación en el parser:

```

1 std::unique_ptr<Expr> Parser::parseTernary() {
2     auto expr = parseLogicalOr();
3
4     if (match(TokenType::QUESTION)) {
5         auto trueExpr = parseExpression();
6         expect(TokenType::COLON, "Expected ':'");
7         auto falseExpr = parseExpression();
8         return std::make_unique<TernaryExpr>(
9             std::move(expr),
10            std::move(trueExpr),
11            std::move(falseExpr)
12        );
13    }
14
15    return expr;
16 }
```

8.2. Alias de Tipo (typedef)

Permite crear nombres alternativos para tipos existentes:

Listing 18: Ejemplo de typedef

```

1 typedef int entero;
2 typedef long numero_grande;
3 typedef float decimal;
4
5 entero x = 10;
6 numero_grande big = 1000000;
7 decimal pi = 3.14;
```

El parser mantiene un mapa `typeAliases` que asocia nombres de alias con sus tipos base.

9. Casos de Prueba

El compilador incluye 18 casos de prueba organizados en cuatro categorías.

9.1. Tests de Funciones (3 casos)

Test	Descripcion	Salida
test_func1.c	Funcion suma simple	15
test_func2.c	Funcion cuadrado anidada	49
test_func3.c	Factorial recursivo	120

Listing 19: test_func3.c - Factorial recursivo

```

1 int factorial(int n) {
2     if (n <= 1) {
3         return 1;
4     }
5     return n * factorial(n - 1);
6 }
7
8 int main() {
9     int result = factorial(5);
10    printf("%d\n", result); // Output: 120
11    return 0;
12 }
```

9.2. Tests de Implementacion Base (5 casos)

Test	Descripcion	Salida
test_base1.c	Aritmetica basica	30, 10, 200, 2
test_base2.c	Sentencias if-else	10, 1
test_base3.c	Bucle while	45
test_base4.c	Bucle for	45
test_base5.c	Expresiones complejas	11, 16, 4

9.3. Tests de Extensiones (5 casos)

Test	Descripcion	Salida
test_ext1.c	Expresiones ternarias	10, 5
test_ext2.c	Unsigned int	150
test_ext3.c	Long int	3000000
test_ext4.c	Float	6.280000
test_ext5.c	Typedef	30, 1000000, 3.140000

9.4. Tests de Optimizacion (5 casos)

Test	Descripcion	Salida
test_opt1.c	Constant folding	5, 50, 25
test_opt2.c	Dead code elimination	30
test_opt3.c	Strength reduction	16, 32
test_opt4.c	Common subexpression	16
test_opt5.c	Loop optimization	100

10. Ejemplo de Código Generado

10.1. Código Fuente

Listing 20: Programa factorial

```

1 int factorial(int n) {
2     if (n <= 1) {
3         return 1;
4     }
5     return n * factorial(n - 1);
6 }
7
8 int main() {
9     int result = factorial(5);
10    printf("%d\n", result);
11    return 0;
12 }
```

10.2. Código Ensamblador Generado

Listing 21: Código x86-64 generado

```

1 .intel_syntax noprefix
2 .text
3 .global main
4
5 factorial:
6     push rbp
7     mov rbp, rsp
8     mov rax, [rbp + 16]          ; Cargar parametro n
9     push rax
10    mov rax, 1
11    mov rbx, rax
12    pop rax
13    cmp rax, rbx
14    setle al
15    movzx rax, al
16    test rax, rax
17    jz .L1                      ; Si n > 1, saltar
18    mov rax, 1                   ; return 1
19    mov rsp, rbp
20    pop rbp
21    ret
22 .L1:
23    mov rax, [rbp + 16]          ; Cargar n
24    push rax
25    sub rsp, 8                  ; Alinear stack
26    mov rax, [rbp + 16]
27    push rax
28    mov rax, 1
```

```

29    mov rbx, rax
30    pop rax
31    sub rax, rbx          ; n - 1
32    push rax
33    call factorial         ; Llamada recursiva
34    add rsp, 16
35    mov rbx, rax           ; resultado en rbx
36    pop rax                ; n en rax
37    imul rax, rbx          ; n * factorial(n-1)
38    mov rsp, rbp
39    pop rbp
40    ret
41
42 main:
43     push rbp
44     mov rbp, rsp
45     sub rsp, 16
46     sub rsp, 8
47     mov rax, 5
48     push rax
49     call factorial
50     add rsp, 16
51     mov [rbp - 8], rax      ; result = factorial(5)
52     mov rax, [rbp - 8]
53     push rax
54     lea rax, [.STR0]
55     push rax
56     pop rdi
57     pop rsi
58     xor eax, eax
59     call printf
60     mov rax, 0
61     mov rsp, rbp
62     pop rbp
63     ret
64
65 .data
66 int_fmt: .asciz "%ld\n"
67 .STR0: .asciz "%d\n"

```

11. Instrucciones de Uso

11.1. Compilacion del Compilador

```

1 # Compilar el compilador
2 make
3
4 # Limpiar archivos generados
5 make clean

```

11.2. Ejecucion

Linux x86-64:

```
1 ./compiler archivo.c
2 gcc -no-pie output.s -o program
3 ./program
```

macOS (Apple Silicon):

```
1 ./run_x86.sh archivo.c
```

Windows (WSL2):

```
1 ./compiler archivo.c
2 gcc -no-pie output.s -o program
3 ./program
```

12. Conclusiones

Se logro implementar exitosamente un compilador funcional que traduce un subconjunto significativo de C a codigo ensamblador x86-64. Los principales logros incluyen:

- Implementacion completa de las cuatro fases del compilador
- Soporte para multiples tipos de datos con promocion automatica
- Manejo correcto de funciones recursivas y estructuras de control
- Optimizaciones efectivas que reducen el codigo generado
- Extensiones utiles como expresiones ternarias y typedef

El proyecto demuestra la aplicacion practica de los conceptos teoricos de compiladores, incluyendo analisis lexico, parsing recursivo descendente, verificacion de tipos, generacion de codigo y optimizaciones.

12.1. Trabajo Futuro

Posibles mejoras para versiones futuras:

- Soporte para estructuras (struct) y uniones
- Implementacion de punteros y memoria dinamica
- Optimizaciones adicionales (register allocation, peephole)
- Soporte para mas operadores de C
- Generacion de codigo para otras arquitecturas

13. Referencias

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
2. Intel Corporation. (2021). *Intel 64 and IA-32 Architectures Software Developer's Manual*.
3. System V Application Binary Interface: AMD64 Architecture Processor Supplement.
4. Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.