



Informe Técnico: Compilador de subconjunto C

Portada

- **Título:** Compilador de subconjunto C
- **Integrantes:** Lucas Manuel Moyano Gómez, Ignacio Jesús Olariaga Oliveto
- **Materia:** Técnicas de Compilación
- **Profesores:** Maximiliano Andrés Eschoyez, Francisco Ameri Lopez Lozano
- **Fecha:** 11/12/2025

Introducción

Compilador académico construido con ANTLR4 (v4.13.1) y Java 17 para un subconjunto de C procedimental. Implementa análisis léxico, sintáctico, semántico, generación de código intermedio, optimización y backend NASM x86. Objetivos: ejercitarse el pipeline completo de compilación, soportar tipos básicos (int, char, double), control de flujo y funciones con llamadas anidadas.

Análisis del Problema (subconjunto de C)

- Tipos: `int`, `char`, `double`, `void`.

- Estructuras: declaraciones, asignaciones, expresiones aritméticas/lógicas, `if/else`, `while`, `for`, `break`, `continue`, `return`.
- Funciones: prototipos y definiciones; llamadas con argumentos por valor; retornos `int/char/double/void`.
- Sin clases, plantillas, herencia, ni manejo de memoria dinámica. IO no contemplada; el backend solo emite asm.

Diseño de la Solución

La solución se estructuró siguiendo el pipeline clásico de compilación, pero adaptado a las restricciones del trabajo práctico.

- **Arquitectura general:** gramática ANTLR4 → lexer/parser generados → listener semántico (`Escucha`) → visitor de CI (`GeneradorCodigoIntermedio`) → optimizador (`Optimizador`) → backend NASM (`GeneradorAssembler`) → archivos en `salida/`.
- **Fases de compilación:** léxico, sintáctico, semántico, código intermedio, optimización y generación de código (resumidas también en el anexo).

Decisiones de diseño principales

- **Uso de ANTLR4 con listener + visitor:** se eligió ANTLR4 para concentrar el esfuerzo en las fases altas (semántica, CI, optimización, backend) en lugar de escribir a mano el parser. Se usa el patrón listener (`Escucha`) para el análisis semántico (natural para construir/actualizar la tabla de símbolos) y el patrón visitor (`GeneradorCodigoIntermedio`) para generar código de tres direcciones.
- **Representación intermedia en tres direcciones:** se introduce un nivel de CI con temporales (`tN`) y etiquetas (`IN`) para simplificar optimizaciones (propagación de constantes, folding, CSE, liveness) y separar preocupaciones respecto del backend x86.
- **Backend x86 de 32 bits y x87 para double:** NASM 32-bit con convención cdecl (prólogo/epílogo con `ebp`), retorno en `eax` para `int/char` y `st0` para `double`. La FPU x87 simplifica operaciones en punto flotante (`fld`, `fstp`, `fadd`, etc.). Constantes en `.data` y variables en `.bss`.
- **Tipado de llamadas y separación CI/backend:** el CI mantiene llamadas con notación compacta de argumentos; el backend consulta tabla de símbolos

y árbol para tipos reales y tamaños (4/8 bytes). Esto evita sobrecargar el CI con detalles de bajo nivel.

- **Optimización basada en CFG y liveness completo:** se construye un CFG para aplicar propagación de constantes y folding según orden de ejecución, CSE por bloque y análisis de vida para eliminar código muerto en temporales y no temporales.
- **Subconjunto de C acotado:** se restringe a tipos y estructuras esenciales para focalizar en las fases del compilador y concretar una implementación completa.
- **Sistema de reporte unificado:** `Reportador` centraliza mensajes con colores y niveles (info/warning/error) para depuración y experiencia de uso.

Implementación

- **Detalles técnicos:** Java 17, Maven; ANTLR4 plugin genera lexer/parser en `target/generated-sources/antlr4/` (no editar). `Reportador` centraliza mensajes coloreados.
- **Gramática ANTLR4:** `src/main/antlr4/compiladores/compiladores.g4` define tokens y reglas (programa, instrucciones, expresiones, control, funciones, llamadas). Genera `compiladoresLexer`, `compiladoresParser`, visitors/listeners base.
- **Tabla de símbolos:** `TablaSimbolos` (singleton) con pila de contextos; guarda variables y funciones con tipo (`TipoDatos`), inicialización, ámbito y firmas. `Escucha` gestiona alta/baja de contextos y chequeos (no declarado, doble declaración, no inicializado, firmas compatibles, conteo de argumentos).
- **Algoritmos por fase:**
 - Léxico/sintáctico: LL(*) de ANTLR sobre la gramática.
 - Semántico: recorridos en `Escucha` con reglas de ámbito y tipo; validaciones básicas de argumentos y retorno.
 - CI: `GeneradorCodigoIntermedio` emite tres direcciones (temporales `tN`, etiquetas `IN`).
 - Optimización: `Optimizador` aplica propagación de constantes, constant folding, CSE intra-bloque, y eliminación de código muerto vía liveness (CFG con etiquetas/if/goto).

- Backend: `GeneradorAssembler` visita el árbol con tipado `int/char/double`, maneja llamadas, convierte `int→double` en retornos y emite NASM.
- **Técnicas de optimización:** propagación de constantes, constant folding, eliminación de subexpresiones comunes, eliminación de código inalcanzable tras `goto` y `return`, eliminación de `x = x;` y liveness completo.

Ejemplos y Pruebas

- **Casos:** `entrada/programa.txt` (flujo completo sin errores), `entrada/programa_errores.txt` (reportes semánticos). Otros ejemplos en `entrada/` (if/else, while, for, aritmética).
- **Salidas generadas:**
 - CI: `salida/codigo_intermedio.txt`.
 - CI optimizado: `salida/codigo_optimizado.txt`.
 - ASM: `salida/programa.asm`.
- **Código de prueba principal (`programa.txt`):**

```
c int f(int x);int sumar(int a, int b); int max_int(int a, int b); double promedio3
(int a, int b, int c); void contar_hasta(int n); int factorial(int n); int prueba_bre
ak_continue(int limite); char siguiente_char(char c);

int main() {
    int x = 5;
    int y = 10;
    int z;
    double d1 = 3.5;
    double d2 = 2.0;
    char c1 = 'A';
    char c2;

    z = x + y;
    z = z - 3 * 2;
    z = z / 3;

    d1 = d1 + d2 * 2.0;
    d1 = d1 / 2.5;
```

```

int condicion1 = (x < y);
int condicion2 = (z == 3);
int condicion3 = (x > y);
int condicion4 = (condicion1 && condicion2);
int condicion5 = (condicion3 || condicion2);
int condicion6 = !condicion3;

if (condicion4) {
    z = z + 1;
} else {
    z = z - 1;
}

if (condicion3) {
    z = -1;
} else {
    z = z + 2;
}

int suma = sumar(x, y);
int maximo = max_int(x, y);
double prom = promedio3(3, 4, 5);
int fact_5 = factorial(5);
int resultado_bc = prueba_break_continue(10);

int contador = 0;
while (contador < 5) {
    contador = contador + 1;
}

int suma_for = 0;
int i;
for (i = 0; i < 5; i = i + 1) {
    suma_for = suma_for + i;
}

int suma_pares = 0;

```

```

for (i = 0; i < 10; i = i + 1) {
    if (i % 2 != 0) {
        continue;
    }
    if (i > 6) {
        break;
    }
    suma_pares = suma_pares + i;
}

c2 = siguiente_char(c1);
char c3 = 'z';
char c4 = siguiente_char(c3);

return 0;
}

int sumar(int a, int b) {
    int r = a + b;
    return r;
}

int max_int(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

double promedio3(int a, int b, int c) {
    int suma = a + b + c;
    double prom = suma / 3;
    return prom;
}

void contar_hasta(int n) {
    int i = 0;

```

```
while (i < n) {
    i = i + 1;
}

int factorial(int n) {
    int res = 1;
    int i;
    for (i = 1; i <= n; i = i + 1) {
        res = res * i;
    }
    return res;
}

int prueba_break_continue(int limite) {
    int i = 0;
    int suma = 0;

    while (i < limite) {
        i = i + 1;

        if (i == 3) {
            continue;
        }

        if (i == 7) {
            break;
        }

        suma = suma + i;
    }

    return suma;
}

char siguiente_char(char c) {
    char r = c + 1;
```

```
    return r;  
}
```

- **Código con errores** ([programa_errores.txt](#)):

```
c int sumar(int a, int b); int sinReturn(int x); void g(); int h(int a); void testBreak(); void testContinue();
```

```
int main() {  
    int a = 1;  
    int b;  
    int a;  
    x = 5;  
    int c = d + 1;  
    double d1 = 2.5;  
    double d2 = 1;  
    int e = d1;  
    char ch = 'A';  
    int r1 = sumar(a, d1);  
    int r2 = sumar(a);  
    int r3 = sinReturn(10);  
    if (z > 0) {  
        int y;  
        y = y + 1;  
    }  
    for (i = 0; i < 10; i = i + 1) {  
        break;  
    }  
    return 0.5;  
}
```

```
int sumar(int a, int b) {  
    return a + b;  
}
```

```
int sinReturn(int x) {  
    int y = x + 1;  
}
```

```

void g() {
    return 1;
}

int h(int a) {
    return a;
}

double h(int a) {
    return 1.0;
}

void testBreak() {
    break;
}

void testContinue() {
    continue;
}

```

- **Análisis:** `programa.txt` compila sin errores y genera ASM coherente con operaciones int,double y control de flujo; `programa_errores.txt` reporta dobles declaraciones, no declarados, no inicializados y firmas incompatibles.

Dificultades Encontradas y Soluciones Aplicadas

- **Integración incremental de fases:** inconsistencias entre gramática, CI y backend al agregar fases. Se resolvió estabilizando un flujo claro y usando archivos intermedios en `salida/` para depurar cada fase.
- **Manejo de `double` y `x87`:** definición de layout, convención de retorno (`st0`) y conversiones int↔double. Solución con instrucciones x87 y reglas explícitas de conversión.
- **Tipado de llamadas y paso de parámetros:** el backend recupera tipos desde árbol/tabla de símbolos para decidir bytes a empujar y tratamiento de `double`.

- **Liveness y eliminación de código muerto:** rediseño del análisis de vida con CFG y propagación de conjuntos IN/OUT a punto fijo.
- **Sincronización gramática-semántica:** ajustes coordinados en gramática, `Escucha`, CI y backend al extender el subconjunto.

Conclusiones

Se completó un pipeline funcional de compilación para un subconjunto de C: lexer/parser ANTLR, semántica con tabla de símbolos, CI, optimización y backend NASM con soporte de `double`. Posibles mejoras: tipado más estricto en CI, generación de temporales por argumento y optimizaciones de bucles.

Referencias Bibliográficas

- Aho, Lam, Sethi, Ullman. *Compilers: Principles, Techniques, and Tools*.
- Dick Grune et al. *Modern Compiler Design*.
- Alfred V. Aho, Jeffrey D. Ullman. *Construcción de Compiladores*.

Anexos

Manual de Usuario

Requisitos e instalación

- Java 17 instalado y disponible en `PATH`.
- Maven instalado (el proyecto incluye `pom.xml`).
- Clonar o descargar el repositorio y ubicarse en la carpeta raíz del proyecto.

Para compilar el proyecto y generar el lexer/parser de ANTLR4:

```
mvn clean compile
```

Ejecución básica del compilador

Desde la raíz del proyecto, para compilar un programa de ejemplo:

```
mvn -q exec:java "-Dexec.mainClass=compiladores.App" "-Dexec.args=en  
trada/programa.txt"
```

- El argumento en `-Dexec.args` es la ruta al archivo fuente en el subconjunto de C.
- Puede reemplazarse `entrada/programa.txt` por cualquier otro archivo de `entrada/`.

Archivos de salida generados

- `salida/codigo_intermedio.txt` : código de tres direcciones.
- `salida/codigo_optimizado.txt` : versión optimizada (const-prop, folding, CSE, liveness).
- `salida/programa.asm` : código ensamblador NASM x86.

Para ensamblar y enlazar en Linux de 32 bits:

```
nasm -f elf32 salida/programa.asm -o salida/programa.o
ld -m elf_i386 salida/programa.o -o salida/programa
./salida/programa
```

Interpretación de mensajes, errores y warnings

`Reportador` usa colores: Verde (éxito/info), Amarillo (warnings), Rojo (errores). Incluye línea y columna cuando aplica.

Buenas prácticas

- Editar solo `src/main/antlr4` y `src/main/java`; no modificar `target/`.
- Tras cambiar `compiladores.g4`, ejecutar `mvn clean compile`.
- Mantener los programas de prueba dentro de `entrada/`.

Notas técnicas ampliadas

- **Arquitectura y flujo (detalle):** gramática → lexer/parser generados → `Escucha` → `GeneradorCodigoIntermedio` → `Optimizador` → `GeneradorAssembler`.
- **Convenciones rápidas:** tokens en MAYÚSCULAS, reglas en minúsculas; cdecl simple con `ebp`; retorno en `eax` (int/char) o `st0` (double).
- **Ejemplo de traducción ASM (fragmento):**

```
section .bss
x: resd 1
```

```

y: resd 1
z: resd 1
section .text
_start:
    mov dword [x], 5
    mov dword [y], 10
    ; z = x + y
    mov eax, [x]
    push eax
    mov eax, [y]
    pop ebx
    add eax, ebx
    mov [z], eax

```

- **Resumen de las 6 fases:**

- 1) Léxico: `compiladoresLexer` tokeniza y detecta caracteres inválidos.
- 2) Sintáctico: `compiladoresParser` valida estructura y construye el parse tree.
- 3) Semántico: `Escucha` verifica tipos, ámbitos, inicialización y firmas; llena tabla de símbolos.
- 4) Código intermedio: `GeneradorCodigoIntermedio` produce tres direcciones (temporales/etiquetas).
- 5) Optimización: `Optimizador` ejecuta const-prop, folding, CSE y liveness.
- 6) Generación de código: `GeneradorAssembler` emite NASM x86 (int/char/double con x87).

<https://github.com/SecondPort/TecCompilacion2024>