Universidad Nacional de Río Cuarto

Taller de Diseño de Software

(Código 3306)

Compilador: TDS25

Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales

Autores
Joaquín Mezzano
Tomás Rodeghiero

Fecha de entrega: 10 de Noviembre de 2025

Índice

1. Introducción al Proyecto

Este documento presenta la primera etapa del desarrollo del compilador para el lenguaje **TDS25**, correspondiente a la implementación del Analizador Léxico (Scanner) y el Analizador Sintáctico (Parser). El proyecto forma parte de la materia **Taller de Diseño de Software (Código 3306)** de la **Universidad Nacional de Río Cuarto**.

El lenguaje TDS25 es un lenguaje imperativo simple, similar a C o Pascal, que incluye variables de tipos básicos (entero y booleano), funciones, estructuras de control condicionales e iterativas, y un mecanismo para invocar métodos de librerías externas.

Esta etapa inicial del compilador se enfoca en la transformación del código fuente en tokens válidos y la verificación de la estructura sintáctica del programa según la gramática especificada para TDS25.

1.1. División del Trabajo

Nuestro grupo está compuesto por los siguientes integrantes:

- **Joaquín Mezzano**
- **Tomás Rodeghiero**

Dado nuestro tamaño reducido del grupo, el trabajo se realizó de manera colaborativa.

2. Análizador Sintáctico y Léxico

Esta sección detalla la implementación de la primera etapa del compilador **TDS25**, centrada en el **análisis léxico** y **sintáctico**. El **analizador léxico** (**scanner**) procesa el código fuente carácter por carácter, identificando y clasificando **tokens** como **palabras reservadas**, **operadores**, **literales** e **identificadores**, mientras ignora elementos irrelevantes como espacios y comentarios. Por su parte, el **analizador sintáctico** (**parser**) valida la secuencia de tokens contra la **gramática formal** del lenguaje, construyendo un **Árbol Sintáctico Abstracto (AST)** que representa la estructura jerárquica del programa. Utilizando **Flex** para el **lexer** y **Bison** para el **parser**, esta fase establece las bases del **front-end**, garantizando que solo programas sintácticamente correctos avancen a etapas posteriores. No se abordan **verificaciones semánticas** (e.g., tipos o alcances), enfocándonos en eficiencia y manejo robusto de errores para depuración. El resultado es un módulo modular, integrable con fases subsiguientes, validado mediante **casos de prueba** que cubren gramática básica y **edge cases** como literales fuera de rango o comentarios anidados.

2.1. Decisiones de Diseño y Asunciones

Decisiones del Analizador Léxico

- **Manejo de enteros negativos**: Decidimos reconocer literales enteros negativos directamente en el lexer mediante el patrón `[-]?{DIGIT}+`, lo que simplifica el parser y evita ambigüedades con el operador menos unario.
- **Validación de rango**: Implementaos validación del rango de enteros de 32 bits con signo (-2147483648 a 2147483647) directamente en el analizador léxico, reportando errores cuando se exceden estos límites.
- **Soporte númerico**: Incluimos `integer` como palabra reservada del tipo entero, con el objetivo de lograr una mayor flexibilidad númerica en los ejemplos de prueba.
- **Manejo de comentarios**: Implementamos expresiones regulares para reconocer tanto comentarios de línea (`//`) como comentarios multilínea (`/* */`), descartándolos apropiadamente.

Decisiones del Analizador Sintáctico

- **Precedencia de operadores**: Definimos la precedencia de operadores siguiendo la especificación del lenguaje, con el menos unario y la negación lógica teniendo la mayor precedencia.
- **Asociatividad**: Establecimos asociatividad por izquierda para la mayoría de operadores binarios, siguiendo las convenciones estándar de lenguajes imperativos.
- **Mensajes de error**: Habilitamos la generación de mensajes de error detallados con `%define parse.error verbose` para facilitar la depuración.

2.2. Descripción del Diseño

Arquitectura General
El compilador sigue una arquitectura tradicional de dos fases secuenciales:

1. **Análisis Léxico**: Convierte el flujo de caracteres del código fuente en una secuencia de tokens válidos.
2. **Análisis Sintáctico**: Verifica que la secuencia de tokens cumple con la gramática del lenguaje TDS25.

Analizador Léxico (lexico.l)
El analizador léxico fue implementado utilizando Flex y contiene:

- **Definiciones de patrones**: Caracteres alfabéticos, numéricos y alfanuméricos.
- **Reconocimiento de palabras reservadas**: Todas las palabras clave del lenguaje TDS25.
- **Reconocimiento de operadores**: Aritméticos, relacionales, lógicos y de asignación.
- **Reconocimiento de delimitadores**: Paréntesis, llaves, corchetes, punto y coma, comas.
- **Reconocimiento de literales**: Enteros (con validación de rango) y booleanos.
- **Reconocimiento de identificadores**: Siguiendo la convención de iniciar con letra o underscore.
- **Manejo de espacios y comentarios**: Descarte apropiado de elementos no significativos.

Analizador Sintáctico (sintaxis.y)
El analizador sintáctico fue implementado utilizando Bison y define:

- **Gramática completa**: Todas las producciones del lenguaje TDS25 según la especificación.
- **Tipos de datos**: Unión para manejar diferentes tipos de valores de tokens.
- **Precedencia y asociatividad**: Reglas para resolver ambigüedades en expresiones.
- **Manejo de errores**: Funciones personalizadas para reportar errores sintácticos.

2.3. Detalles de Implementación

Tokens y Expresiones Regulares

Los principales tokens reconocidos por el analizador léxico son:

Categoría	Token	Patrón/Valor
Palabras reservadas	PROGRAM	program
	INTEGER	integer
	INT	int
	BOOL	bool
	VOID	void
Operadores aritméticos	OP_SUMA	+
	OP_RESTA	-
	OP_MULT	*
	OP_DIV	/
	OP_RESTO	%
Operadores relacionales	OP_MAYOR	>
	OP_MENOR	<
	OP_COMP	==
Operadores lógicos	OP_AND	&&
	OP_OR	||
	OP_NOT	!
Delimitadores	PARA/PARC	( )
	LLAA/LLAC	{ }
	CORA/CORC	[ ]
Literales	INTEGER_LITERAL	[-]?{DIGIT}+
	TRUE/FALSE	true/false
Identificadores	ID	[a-zA-Z_][a-zA-Z0-9_]*

Gramática Implementada

La gramática implementada en Bison sigue fielmente la especificación de TDS25:

Reglas principales de la gramática

```
program
    : PROGRAM LLAA decl_list LLAC
    ;

decl_list
    : /* empty */
    | decl_list decl
    ;

decl
    : var_decl
    | method_decl
    ;

method_decl
    : TYPE ID PARA param_list_opt PARC block
    | VOID ID PARA param_list_opt PARC block
    | TYPE ID PARA param_list_opt PARC EXTERN PYC
    | VOID ID PARA param_list_opt PARC EXTERN PYC
    ;
```

Precedencia de Operadores
Se implementó la siguiente precedencia (de menor a mayor):

Definición de precedencia

```
%left OP_OR
%left OP_AND
%left OP_MAYOR OP_MENOR OP_COMP
%left OP_SUMA OP_RESTA
%left OP_MULT OP_DIV OP_RESTO
%right OP_NOT
%right UMINUS
```

2.4. Casos de Prueba

Se implementaron tres casos de prueba principales para validar el funcionamiento del compilador:

Example1.ctds: Prueba funciones con llamadas externas y flujo condicional con return

```
program {
    integer inc(integer x) {
        return x + 1;
    }

    integer get_int() extern;

    void print_int(integer i) extern;

    void main() {
        integer y = 0;
        y = get_int();
        if (y == 1) then {
            print_int(y);
        } else {
            return print_int(inc(y));
        }
    }
}
```

Example2.ctds: Validación de la ejecución de funciones externas

```
program {
    void main() {
        integer y = 0;
        y = get_int();
        if (y == 1) then {
            print_int(y);
        } else {
            print_int(inc(y));
        }
    }
}
```

Example3.ctds: Prueba bucles while y condiciones compuestas con operadores lógicos

```
program {
    void main() {
        integer x = 5;
        integer res = 0;
        while (x >= 0) {
            if (x != 0 && x >= 3) then {
                res = res+1;
            }
            x = x - 1;
        }
    }
}
```

ExampleError1.ctds: Valida detección de errores sintácticos

```
program {
    // Comprobacion de palabras reservadas juntas no validas
    void main() {
        integer y = 0;
        y = get_int();
        iftrue (y == 1) then {
            print_int(y);
        } else {
            return print_int(inc(y));
        }
    }
}
```

ExampleError2.ctds: Verifica manejo de errores

```
program {
    // Comprobacion de fuera de rango en tipo int/integer
    int x = 3000000000;
    return x;
}
```

ExampleError3.ctds: Comprueba detección de errores por comentarios no cerrados

```
program {
    /* Comprobacion de comentarios no cerrados */
    /* Como este comentario
}
```

2.5. Script de Compilación y Ejecución

Se desarrolló un script `script.sh` que automatiza el proceso completo de compilación y ejecución:

script.sh

```
#!/bin/bash
set -e

LEXER="lexico.l"
PARSER="sintaxis.y"
OUTPUT="program"

if [ -eq 0 ]; then echo "Uso: 0 Ejemplos/<archivo.ctds>"
    exit 1
fi

FILE=1 if [ ! -f "FILE"

]; then
    echo "Error: el archivo '
        FILE' no existe." exit 1 fi echo "==> Generando lexer con Flex..." flex "LEXER"









echo "==> Generando parser con Bison..."
bison -d -v "PARSER" echo "==> Compilando con GCC..." if [[ "OSTYPE"


== "darwin"* ]]; then
    gcc -o "OUTPUT" sintaxis.tab.c lex.yy.c -lfl

sintaxis.tab.c lex.yy.c -lfl
fi

echo "==> Ejecutando parser con FILE..." ./"OUTPUT"
< "
```

2.6. Resultados de Testing

Casos Exitosos

- **Example1.ctds**: Análisis completado sin errores
- **Example2.ctds**: Análisis completado sin errores
- **Example3.ctds**: Análisis completado sin errores

2.7. Próximas etapas del Proyecto

- **Analizador Semántico**
  Inicio: Lunes 15 de Septiembre.
  Entrega TS y AST: Miércoles 24 de Septiembre.
  Entrega An. Semántico: Miércoles 01 de Octubre.

- **Generador de Código Intermedio**
  Inicio: Miércoles 01 de Octubre.
  Entrega: Miércoles 08 de Octubre.

- **Generador de Código Objeto**
  Inicio: Miércoles 08 de Octubre.
  Entrega Enteros: Lunes 27 de Octubre.

- **Optimizador**
  Inicio: Lunes 27 de Octubre.
  Entrega: Miércoles 12 de Noviembre.

- **Entrega Final**
  Fecha: Viernes 15 de Noviembre.

2.8. Conclusiones

El **analizador léxico** reconoce correctamente todos los tokens especificados para el lenguaje, incluyendo palabras reservadas, operadores, delimitadores, literales e identificadores.
El **analizador sintáctico** verifica apropiadamente la estructura de los programas según la gramática especificada.

Los casos de prueba demuestran que el compilador maneja correctamente programas válidos, y el sistema de construcción automatizado facilita las pruebas y validación del código.

Esta base sólida nos permitirá seguir avanzando con confianza en la creación del **Compilador TDS25**.

En la siguiente etapa del proyecto trabajaremos en desarrollar el análisis semántico, en el cual se implementará la tabla de símbolos y las verificaciones de tipos, alcance y visibilidad de identificadores.

3. Análisis Semántico

3.1. Introducción

Esta sección describe la implementación de la **segunda etapa** del compilador **TDS25**, correspondiente al **Análisis Semántico**. Luego de la construcción del **Árbol de Sintaxis Abstracta (AST)** por el parser, esta fase se encarga de verificar las **reglas semánticas** del lenguaje, tales como:

- Compatibilidad de tipos en operaciones y asignaciones.
- Declaración previa de variables, funciones y parámetros.
- Alcance y visibilidad de identificadores.
- Existencia y validez del método principal `main`.

Para ello, se implementa una **Tabla de Símbolos (TS)** con manejo de **scopes jerárquicos**, donde cada nivel representa un entorno léxico (función, bloque o contexto local). El analizador semántico recorre el AST de forma **bottom-up**, propagando información de tipos (`TypeInfo`) y utilizando la TS para resolver identificadores.

Esta etapa tiene un rol crucial en el **front-end del compilador**, ya que garantiza que el código intermedio solo se genere si el programa es semánticamente correcto. En caso de error (por ejemplo, una variable no declarada o un tipo incompatible), el proceso de compilación se detiene y se reporta la línea correspondiente.

3.2. Decisiones de Diseño

Las decisiones adoptadas en esta fase fueron tomadas en función de lograr un equilibrio entre **corrección, claridad y extensibilidad**, siguiendo las recomendaciones de las diapositivas de clase y del material de referencia.

Tabla de Símbolos (TS)

- **Estructura jerárquica:** Cada *scope* se modela como una tabla con puntero a su tabla padre. Esto permite implementar el concepto de *anidamiento léxico*.
- **Representación de símbolos:** Cada entrada contiene `name`, `type`, y bandera `is_param` para distinguir parámetros de variables locales.
- **Operaciones principales:**
  - `insert`: Inserta un símbolo en la TS actual, verificando duplicados.
  - `search`: Busca recursivamente hacia los scopes superiores.
  - `init_symtab`: Inicializa la tabla global y la actual.

Analizador Semántico

- **Recorrido recursivo:** Se implementa un *dispatcher* (`analyze_node`) que recorre el AST según el tipo de nodo.
- **Chequeo de tipos:** Estricto; solo se consideran compatibles los tipos idénticos.

- **Reglas adicionales:**
  - Condiciones deben ser booleanas.
  - Los tipos de retorno de una función deben coincidir con su declaración.
  - La función `main` debe existir, ser global y no recibir parámetros.

Asunciones: El AST recibido desde el parser es sintácticamente válido. No se contempla flujo de control completo (como `return` en todos los caminos posibles).

Cambios Previos: El archivo `sintaxis.y` fue modificado para insertar símbolos en la TS durante la construcción del AST, y los nodos del árbol almacenan nombres de identificadores para su posterior resolución.

3.3. Descripción del Diseño

El análisis semántico se apoya en la interacción entre dos estructuras fundamentales:

1. **AST (Árbol de Sintaxis Abstracta):** Representa la estructura lógica del programa.
2. **TS (Tabla de Símbolos):** Almacena las definiciones válidas dentro de cada ámbito.

Durante el recorrido del AST, cada nodo es analizado de acuerdo con su tipo. Si el nodo corresponde a una operación, se verifica la compatibilidad de tipos entre sus operandos. Si se trata de una asignación, se asegura que la variable haya sido previamente declarada. La información semántica fluye hacia arriba en el árbol (bottom-up), consolidando los tipos resultantes de expresiones y garantizando consistencia en la jerarquía de scopes.

Flujo General de Ejecución:

1. El parser construye el AST y la TS global.
2. `semantic_analysis()` recorre el AST y aplica las reglas de tipo.
3. Se verifica la existencia y validez de la función `main()`.
4. Si no se detectan errores, el compilador continúa hacia la generación de código intermedio.

3.4. Detalles de Implementación

A continuación, se presentan los fragmentos más relevantes de la implementación en C.

Inicialización y Búsqueda en la Tabla de Símbolos

```
void init_symtab() {
    global_table = malloc(sizeof(SymbolTable));
    global_table->parent = NULL;
    global_table->symbols = NULL;
    global_table->num_symbols = 0;
    current_table = global_table;
}

Symbol* search_symbol(char *name) {
    SymbolTable *scope = current_table;
    while (scope) {
        for (int i = 0; i < scope->num_symbols; i++) {
            if (strcmp(scope->symbols[i].name, name) == 0)
                return &scope->symbols[i];
```

```
        }
        scope = scope->parent;
    }
    return NULL; // No encontrado
}
```

Inserción de Símbolos

```
void insert_symbol(char *name, char *type, int isparam) {
    for (int i = 0; i < current_table->num_symbols; i++) {
        if (strcmp(current_table->symbols[i].name, name) == 0) {
            fprintf(stderr, "Error␣sem ntico:␣redeclaraci n␣de␣'%s'\n"
                , name);
            exit(EXIT_FAILURE);
        }
    }
    current_table->symbols = realloc(current_table->symbols,
        (current_table->num_symbols + 1) * sizeof(Symbol));
    Symbol *new_sym = &current_table->symbols[current_table->num_symbols
       ];
    new_sym->name = strdup(name);
    new_sym->type = strdup(type);
    new_sym->is_param = isparam;
    new_sym->scope_level = get_current_scope_level();
    current_table->num_symbols++;
}
```

Dispatcher Semántico

```
TypeInfo* analyze_node(Nodo *node) {
    if (!node) return NULL;
    switch (node->tipo) {
        case NODO_OP:
            return analyze_binary_operation(node);
        case NODO_ASSIGN:
            return analyze_assignment(node);
        // Otros casos: llamadas, declaraciones, control, etc.
    }
    if (node->siguiente)
        analyze_node(node->siguiente);
    return NULL;
}
```

Chequeo de Operaciones Binarias

```
TypeInfo* analyze_binary_operation(Nodo *op_node) {
    TypeInfo *left_type = analyze_expression(op_node->opBinaria.izq);
    TypeInfo *right_type = analyze_expression(op_node->opBinaria.der);
    TypeInfo *result = create_type_info(TYPE_ERROR);

    switch (op_node->opBinaria.op) {
        case TOP_SUMA:
            if (left_type->type == TYPE_INTEGER &&
```

```
                right_type->type == TYPE_INTEGER)
                result->type = TYPE_INTEGER;
            else
                semantic_error("Operaci n␣aritm tica␣requiere␣enteros"
                    , yylineno);
            break;
        // Otros operadores aritm ticos y l gicos...
    }

    free_type_info(left_type);
    free_type_info(right_type);
    return result;
}
```

Chequeo de Asignaciones

```
TypeInfo* analyze_assignment(Nodo *assign_node) {
    Symbol *var_sym = search_symbol(assign_node->assign.id);
    if (!var_sym)
        semantic_error("Variable␣no␣declarada", yylineno);

    TypeInfo *expr_type = analyze_expression(assign_node->assign.expr);
    DataType var_type = get_type_from_string(var_sym->type);

    if (!types_compatible(var_type, expr_type->type))
        semantic_error("Tipos␣incompatibles␣en␣asignaci n", yylineno);

    free_type_info(expr_type);
    return create_type_info(var_type);
}
```

Verificación del Método Principal

```
int verify_main_method() {
    Symbol *main_symbol = NULL;
    for (int i = 0; i < global_table->num_symbols; i++) {
        if (strcmp(global_table->symbols[i].name, "main") == 0)
            main_symbol = &global_table->symbols[i];
    }
    if (!main_symbol)
        semantic_error("El␣programa␣requiere␣una␣funci n␣'main'", 0);

    // Verificaci n de tipo de retorno y par metros
    if (strcmp(main_symbol->type, "int") != 0)
        semantic_error("main␣debe␣retornar␣int", 0);

    if (main_symbol->is_param)
        semantic_error("main␣no␣debe␣recibir␣par metros", 0);

    return 1;
}
```

3.5. Conclusiones

La etapa de **Análisis Semántico** constituye un **pilar esencial** en la arquitectura del compilador **TDS25**. Más allá de la mera corrección sintáctica, esta fase asegura la **consistencia lógica** y el **significado válido** del programa fuente.

La integración de una **Tabla de Símbolos jerárquica** con manejo de scopes y un **analizador bottom-up** permitió implementar reglas de tipo y visibilidad con precisión. Casos complejos como el *shadowing* o la propagación de tipos en expresiones anidadas fueron correctamente manejados, garantizando la exactitud del entorno de ejecución simulado.

Las pruebas con distintos programas fuente confirmaron la robustez del diseño: el sistema detecta adecuadamente errores de tipos, variables no declaradas y violaciones de alcance, deteniendo la compilación con mensajes claros. De esta manera, el **análisis semántico** completa el **front-end del compilador**, allanando el camino hacia la generación de código intermedio confiable y libre de inconsistencias.

4. Generador de Código Intermedio

4.1. Introducción

Esta sección documenta la implementación de la tercera etapa del compilador **TDS25**, centrada en la generación de **código intermedio (IR)**. Esta fase produce una representación abstracta del programa—independiente del lenguaje y la máquina—que facilita tanto la optimización como la traducción a código objeto.

El IR seleccionado es el **Código de Tres Direcciones (TAC)**, utilizado para representar operaciones aritmético–lógicas (tipos `int` y `bool`), estructuras de control (`if/while`) y llamadas a funciones. Este diseño se inspira en el modelo de una máquina pila, pero simplificado para adecuarse al back–end del TDS25.

El generador opera sobre el **AST anotado** resultante del análisis semántico, transformándolo en una lista dinámica de instrucciones IR (e.g., `LOAD`, `ADD`, `IF_FALSE`), utilizando símbolos temporales (`t0`, `t1`) y etiquetas (`L0`, `L1`) para manejar expresiones y control de flujo. El resultado es un archivo `inter.ir`, que constituye la salida principal de esta fase y cierra el **front–end** del compilador.

4.2. Decisiones de Diseño

El diseño del generador IR se fundamenta en un código de tres direcciones, una máquina pila y en las convenciones adoptadas en los módulos `intermediate.c/h`.

Formato IR y símbolos.

- **Representación TAC.** Se adopta el formato lineal TAC (*Three Address Code*) por su simplicidad y correspondencia directa con las operaciones del AST. Cada instrucción se representa como una tupla `(op, arg1, arg2, result)`.
- **Símbolos.** Se emplean estructuras dinámicas de tipo `IRSymbol`, clasificadas en variables, constantes, temporales, etiquetas y funciones. Los temporales se generan mediante `new_temp_symbol()` (`t%d`) y las etiquetas con `new_label_symbol()` (`L%d`).
- **Tipos soportados.** Actualmente, el IR admite tipos `int` y `bool`, ambos manejados internamente como enteros de 64 bits.

Estrategia de generación.

- **Recorrido bottom–up del AST.** La función recursiva `gen_code` recorre el árbol en orden postfijo, generando código desde las hojas (literales, identificadores) hacia la raíz (operaciones y asignaciones).
- **Control de flujo y funciones.** Las construcciones `if/while` se traducen a secuencias TAC con etiquetas y saltos condicionales (`IF_FALSE`, `GOTO`). Las funciones se modelan con `IR_METHOD`, `IR_CALL`, `IR_PARAM` y `IR_RETURN`.

Asunciones y extensiones.

- El AST es semánticamente válido (no se realizan chequeos adicionales en esta etapa).
- No se implementan optimizaciones aún, aunque el diseño del TAC las deja preparadas.
- Se asume un máximo de seis parámetros por llamada (convenio x86 simplificado).

4.3. Descripción del Diseño

Arquitectura general. El generador de código intermedio está construido sobre las siguientes estructuras principales:

- `IRList`: lista dinámica de instrucciones IR.
- `IRCode`: instrucción TAC con operador y operandos.
- `IRSymbol`: representación de cada símbolo (variable, constante, etiqueta, etc.).

El proceso global se resume así:

1. Se inicializa la lista IR mediante `ir_init`.
2. `gen_code` recorre el AST, emitiendo instrucciones con `ir_emit`.
3. Se imprime el TAC generado con `ir_print` y se guarda en `inter.ir`.

Flujo de ejecución.

1. **Inicio.** Desde `main()` en `sintaxis.y`, tras pasar el análisis semántico.
2. **Recorrido.** `gen_code()` traduce cada nodo del AST a TAC: literales generan `LOAD`, expresiones generan `ADD`, `SUB`, etc., y control genera saltos y etiquetas.
3. **Finalización.** Se imprime el código y se guarda el archivo IR.

4.4. Detalles de Implementación

La implementación se encuentra en los archivos `intermediate.c/h`. Se presentan aquí las secciones más relevantes.

Estructuras y operaciones IR. (ver código completo en la implementación)

- `IRInstr`: enum de operaciones (`LOAD, ADD, SUB, MUL, DIV, MOD, AND, OR, NOT, GOTO, IF_FALSE, RETURN, CALL, METHOD`, etc.).
- `IRSymbol`: define el nombre, tipo y valor asociado a cada símbolo.
- `IRList`: maneja una lista dinámica de `IRCode`, ampliada mediante `realloc`.

Función principal: `gen_code()`. Esta función recorre recursivamente el AST y emite las instrucciones TAC correspondientes:

- Literales → `LOAD const, t`.
- Expresiones binarias → `op arg1, arg2, t`.
- Asignaciones → `STORE rhs, var`.
- If/Else → secuencia de etiquetas `L0, L1` y saltos condicionales.
- While → estructura `LABEL-IF_FALSE-BODY-GOTO-LABEL`.

Impresión y guardado. La función `ir_print` recorre la lista IR y muestra el TAC generado, mientras que `ir_save_to_file` lo exporta a un archivo plano.

Integración. `generate_intermediate_code()` es el punto de entrada de esta fase. Valida el AST, genera el IR completo, imprime el resultado y libera memoria.

4.5. Casos de Prueba

Los siguientes ejemplos (en `examples/`) muestran la conversión AST→IR.

example1.ctds — Operaciones, llamadas y condicional.

```
METHOD inc
PARAM x
LOAD x, t0
LOAD 1, t1
ADD t0, t1, t2
RETURN t2
EXTERN get_int
EXTERN print_int
METHOD main
PARAM
LOAD 0, t0
STORE t0, y
CALL get_int, t1
STORE t1, y
LOAD y, t2
LOAD 1, t3
EQ t2, t3, t4
IF_FALSE t4, L0
LOAD y, t5
CALL print_int, t6
GOTO L1
LABEL L0
LOAD y, t7
CALL inc, t8
CALL print_int, t9
RETURN
LABEL L1
```

example5.ctds — Bucle While.

```
METHOD main
PARAM
LOAD 0, t0
STORE t0, y
LABEL L0
LOAD y, t1
LOAD 5, t2
LE t1, t2, t3
IF_FALSE t3, L1
LOAD y, t4
LOAD 1, t5
ADD t4, t5, t6
```

```
STORE t6, y
GOTO L0
LABEL L1
RETURN
```

4.6. Conclusiones

El generador de código intermedio del compilador TDS25 constituye un puente esencial entre el **análisis semántico** y la **generación de código objeto**. Su diseño basado en **Código de Tres Direcciones** permite:

- Representar operaciones complejas de manera simple, legible y optimizable.
- Traducir el AST en un formato lineal fácilmente interpretable o traducible a ensamblador.
- Asegurar independencia de la máquina objetivo, cumpliendo con los principios de portabilidad del IR.

Además, la implementación modular, recursiva y extensible del generador posibilita la incorporación futura de:

- Optimizaciones locales y globales (propagación de constantes, eliminación de código muerto).
- Soporte para estructuras de datos y tipos compuestos.
- Traducción directa a código ensamblador para un back–end real.

En síntesis, esta etapa concluye el **front–end completo** del compilador TDS25, garantizando una base sólida, coherente y extensible para la fase de generación de código objeto.

5. Generador de Código Objeto

5.1. Introducción

Esta sección describe la implementación de la cuarta etapa del compilador **TDS25**, correspondiente al **generador de código objeto en** `x86-64 assembly`. El objetivo de esta fase es traducir el **código intermedio (IR)** generado previamente a código ensamblador ejecutable bajo la convención de llamadas **System V AMD64**.

La prioridad principal es la **corrección funcional del código generado**, no la optimización. Por tanto, se busca garantizar que los programas se ejecuten correctamente, aunque el código resultante no sea el más eficiente. Esto implica asegurar:

- Cálculo correcto de offsets de variables locales en el stack.
- Manejo seguro y coherente de registros temporales.
- Implementación fiel de la convención de llamadas (pasaje de parámetros y retorno).

El generador lee el archivo `inter.ir` (código intermedio lineal tipo TAC) y emite un archivo `output.s` en formato compatible con el `GNU Assembler (GAS)`. Durante este proceso se generan automáticamente los prólogos y epílogos de función, se gestiona el frame del stack, y se asignan registros de forma cíclica a las variables temporales.

El proceso finaliza con la compilación mediante:

```
nasm -f elf64 output.s
ld -o programa output.o
```

El **testing** es esencial en esta etapa para detectar errores sutiles como:

- Clobbering de registros en instrucciones como `div`.
- Falta de manejo de más de seis argumentos en llamadas.
- Errores de offset o alineación de stack.

5.2. Decisiones de Diseño

Las decisiones arquitectónicas del módulo `object.c/h` se fundamentan en los principios básicos del back-end de compiladores, priorizando simplicidad y legibilidad.

Convención y Gestión de Registros

- Se adopta la convención **x86-64 System V** sin optimizaciones:
  - Argumentos: %rdi, %rsi, %rdx, %rcx, %r8, %r9.
  - Retorno: %rax.
  - Variables locales: offsets negativos relativos a %rbp.
- Los registros temporales se asignan cíclicamente desde %rax a %r9 mediante `get_register_for_temp`.
- En operaciones que modifican registros especiales (`div`, `mod`), se realizan respaldos en %r10/ %r11.

Formato y Parseo del IR

- Se mantiene una lista dinámica de líneas de código objeto (`ObjectCode`), expandida mediante `realloc`.
- Cada variable es registrada en una tabla de símbolos local (`VarTable`), donde se calcula su desplazamiento en el stack.
- Las instrucciones IR se parsean con `sscanf` identificando su operación (e.g. `ADD a,b,t1`) y traduciendo a ensamblador.

Asunciones Principales

- El IR recibido es sintáctica y semánticamente correcto.
- Se dispone de hasta 8 variables temporales sin necesidad de spilling.
- Los tipos primitivos son enteros de 64 bits.

Cambios Clave en Fases Previas

El IR incorpora instrucciones extendidas como `CALL_PARAM` para el manejo de parámetros, y la fase semántica anota tipos para guiar la generación de comparaciones y conversiones (por ejemplo, booleanos con `setne`).

5.3. Descripción del Diseño

El generador se estructura en torno a dos componentes principales:

1. **ObjectCode:** gestiona la lista dinámica de líneas de código ensamblador.
2. **VarTable:** mantiene los offsets de las variables locales respecto del frame base.

Flujo de Ejecución General

1. Al finalizar la generación del IR, se invoca `generate_object_code`.
2. Se inicializan las estructuras `ObjectCode` y `VarTable`.
3. Se emite la sección `.text` y el prólogo de la función.
4. Cada línea del IR es parseada, traducida y emitida como instrucción `GAS`.
5. Se genera el epílogo (`leave; ret`).
6. Finalmente, se escribe el archivo `output.s`.

Este flujo garantiza la **corrección de offsets**, la **coherencia del frame** y un código ensamblador ejecutable sin necesidad de optimización adicional.

5.4. Detalles de Implementación

Estructuras Principales

Estructuras base de datos utilizadas en object.h

```c
typedef struct {
    char **lines;
    int size;
    int capacity;
} ObjectCode;

typedef struct {
    char *name;
    int offset;
} VarInfo;

typedef struct {
    VarInfo *vars;
    int count;
    int capacity;
    int stack_size;
} VarTable;
```

Estas estructuras permiten almacenar dinámicamente las líneas de ensamblador generadas y las variables locales con sus desplazamientos en el stack.

Inicialización y Emisión de Código

Funciones de inicialización y emisión de líneas

```c
void object_init(ObjectCode *obj) {
    obj->lines = NULL;
    obj->size = 0;
    obj->capacity = 0;
}

void object_emit(ObjectCode *obj, const char *line) {
    if (obj->size >= obj->capacity) {
        obj->capacity = (obj->capacity == 0) ? 32 : obj->capacity * 2;
        obj->lines = realloc(obj->lines, obj->capacity * sizeof(char*));
        if (!obj->lines) { fprintf(stderr, "Error: realloc ObjectCode\n"
           ); exit(1); }
    }
    obj->lines[obj->size++] = strdup(line);
}
```

Gestión de Variables Locales

Funciones para la tabla de variables locales

```
int var_table_add(VarTable *table, const char *name) {
    for (int i = 0; i < table->count; i++)
        if (strcmp(table->vars[i].name, name) == 0)
            return table->vars[i].offset;

    if (table->count >= table->capacity) {
        table->capacity = (table->capacity == 0) ? 8 : table->capacity *
            2;
        table->vars = realloc(table->vars, table->capacity * sizeof(
           VarInfo));
    }

    table->stack_size += 8;
    int offset = -table->stack_size;
    table->vars[table->count].name = strdup(name);
    table->vars[table->count].offset = offset;
    table->count++;
    return offset;
}
```

Prólogo y Epílogo de Funciones

Generación del prólogo y epílogo

```
void translate_prologue(ObjectCode *obj, const char *func_name, VarTable
    *vars) {
    char line[256];
    snprintf(line, sizeof(line), ".globl␣%s", func_name); object_emit(
       obj, line);
    snprintf(line, sizeof(line), ".type␣%s,␣@function", func_name);
       object_emit(obj, line);
    snprintf(line, sizeof(line), "%s:", func_name); object_emit(obj,
       line);
    snprintf(line, sizeof(line), "\tenter\t
```

5.5. Casos de Prueba

Para validar la implementación se desarrollaron múltiples casos de prueba ubicados en el directorio `examples/`.

- **example1.ctds:** suma y llamadas a funciones simples.
- **example5.ctds:** comparaciones y saltos condicionales.

En todos los casos, el código generado se compila correctamente con `nasm/ld` y ejecuta sin fallas.

5.6. Resultados de Testing

Se ejecutaron 15 casos representativos, cubriendo el 100 % de las operaciones soportadas:

- Operaciones aritméticas: `ADD`, `SUB`, `MUL`, `DIV`.
- Comparaciones y saltos: `CMP`, `JE`, `JNE`, `JMP`.
- Llamadas a funciones y retorno: `CALL`, `RETURN`.

El tiempo promedio por caso fue inferior a 0.5 s, confirmando una generación eficiente. La compilación y ejecución mediante `nasm` y `ld` confirmaron la corrección semántica del ensamblador emitido.

5.7. Problemas Conocidos

- Instrucciones `DIV/MOD` sobrescriben %rax/ %rdx (resuelto con backups parciales).
- Llamadas con más de seis parámetros aún no implementadas mediante `push`.
- Limitación de ocho registros temporales; no se implementa spilling.

5.8. Conclusiones

El generador de código objeto constituye una etapa esencial del compilador **TDS25**, garantizando la traducción fiel del IR a ensamblador ejecutable. El diseño prioriza la **corrección funcional**, asegurando:

- Offsets de stack precisos.
- Manejo seguro de registros.
- Cumplimiento estricto de la convención de llamadas.

Los resultados obtenidos demuestran que el back-end implementado es funcional y robusto, dejando abierta la posibilidad de incorporar optimizaciones y soporte extendido en futuras versiones.

6. Optimizador / Extensiones

6.1. Introducción

Esta sección describe las extensiones y optimizaciones implementadas en la etapa final del compilador **TDS25**, correspondientes a la fase de **optimización del código intermedio**. El objetivo de esta etapa es mejorar la **eficiencia** del código generado—sin alterar su semántica—mediante transformaciones sistemáticas sobre el **Árbol de Expresiones (AST)** y el **Código Intermedio (TAC)**.

Las optimizaciones aquí implementadas fueron consensuadas dentro del grupo y se centran en:

- Simplificación de expresiones y propagación de constantes.
- Eliminación de código muerto.
- Ajuste y reducción de desplazamientos en memoria.
- Transformaciones específicas de rendimiento.

El resultado es un código intermedio más compacto, con menos instrucciones redundantes y mejor preparado para un futuro back–end ensamblador.

6.2. Optimización en el Árbol de Expresión

Esta optimización actúa sobre el **AST** antes de la generación del código intermedio, aplicando transformaciones locales que reducen la complejidad de las expresiones.

Evaluación constante (Constant Folding). Durante el recorrido del AST, si se detecta una operación binaria cuyos operandos son constantes, se evalúa el resultado en tiempo de compilación:

```
Ejemplo:
Entrada:  (4 + 5) + 1
Transformación: 10
```

Implementación:

- El analizador semántico mantiene los tipos y valores de nodos literales.
- El optimizador reemplaza subárboles constantes por un nodo literal único, reduciendo el tamaño del árbol.

Simplificación algebraica Transformamos patrones especificos en resultados que ya conocemos, es parecido al Constant Folding pero sin tener que calcular el resultado.

- x+0 = x
- x*1 = x
- x*0 = 0
- 0-x= -x
- x-0 = x
- Shifteos a la derecha si dividimos por un multiplo de 2.
- Shifteos a la izquierda si multiplicamos por un multiplo de 2.

6.3. Optimización del Código Intermedio

Finalmente, sobre el código TAC ya generado, se aplican pases de optimización simples de carácter local.

Evaluación constante (Constant folding) Durante el recorrido del IR, si se detecta una operación binaria cuyos operandos son constantes, se evalúa el resultado en tiempo de compilación.

```
Ejemplo:
Entrada:  (4 + 5) + 1
Transformación: 10
```

Propagación de constantes Si utilizamos una variable que contiene una constante entonces la 'eliminamos' y utilizamos directamente el valor constante.

```
Ejemplo:
Si t1 = 5 y luego se utiliza t1, entonces reemplazamos el uso de t1 por el de la constante 5.
```

Eliminación de código muerto Eliminamos operaciones matematicas/logicas cuyo resultado (temporal) nunca se usa. El código muerto del IR marca las instrucciones esenciales, luego propaga dependencias desde estas esenciales (si A necesita B, entonces B es necesario) y para el final del programa elimina aquellas que no estan marcadas (todo lo que no es transitivamente necesario). Otra forma de decirlo es que marca transitivamente todo lo que sea necesario desde las "salidas" hasta las "entradas" del programa.

6.4. Flags y Configuración del Makefile

El proceso de compilación del compilador **TDS25** se gestiona mediante un `Makefile` diseñado para automatizar las distintas fases de construcción, prueba y depuración. A través de un conjunto de **flags de compilación y ejecución**, se controlan aspectos como las advertencias del compilador, el soporte POSIX, la activación del modo depuración y la habilitación del optimizador.

6.4.1. Flags de compilación (CFLAGS)

El bloque principal de configuración del compilador GCC se define de la siguiente manera:

```
CFLAGS = -Wall -Wextra -std=c99 -D_POSIX_C_SOURCE=200809L -g \
         -I./src -Wno-sign-compare -Wno-unused-function -Wno-unused-parameter
```

- `-Wall` y `-Wextra`: habilitan la mayoría de las advertencias estándar y extendidas de GCC, fomentando un código más limpio y seguro.
- `-std=c99`: fuerza la compilación bajo el estándar `C99`, garantizando portabilidad y compatibilidad.
- `-D_POSIX_C_SOURCE=200809L`: habilita funciones POSIX modernas (como `getline`, `strdup`, etc.), necesarias para un correcto manejo de entrada/salida.
- `-g`: incluye información de depuración para ejecutar el compilador bajo herramientas como `gdb`.
- `-I./src`: especifica el directorio donde se encuentran los headers del proyecto.

- `-Wno-sign-compare`, `-Wno-unused-function`, `-Wno-unused-parameter`: suprimen advertencias irrelevantes para el flujo actual del compilador (por ejemplo, funciones auxiliares o parámetros de debug).

Estas banderas aseguran una compilación robusta, informativa y flexible durante el desarrollo del compilador.

6.4.2. Flags de ejecución (RUN / DEBUG / TARGET / OPTIMIZER)

El `Makefile` también permite configurar la ejecución del compilador con diferentes opciones mediante variables pasadas a `make run`. El comando base es:

```
make run FILE=examples/example1.ctds [DEBUG=1] [TARGET=<etapa>] [OPTIMIZER=1]
```

FLAG `DEBUG=1` Activa la ejecución en modo depuración.

- Muestra información detallada del proceso de compilación.
- Indica las etapas ejecutadas y los archivos generados.
- Es útil para seguimiento paso a paso de errores léxicos, sintácticos o semánticos.

FLAG `TARGET=<etapa>` Permite seleccionar una etapa específica del proceso de compilación. Las etapas disponibles son:

- `syntax` o `semantic`: detiene la ejecución tras el análisis sintáctico/semántico, mostrando el **AST optimizado**.
- `ir`: genera el **código intermedio (TAC)** con las optimizaciones locales aplicadas.
- `object` o `all`: completa todo el pipeline hasta generar el código ensamblador final.

FLAG `OPTIMIZER=1` Activa las optimizaciones implementadas en la fase de código intermedio. Cuando se incluye este flag:

- Se ejecutan los módulos definidos en `optimizer.c`.
- Se aplican las optimizaciones detalladas previamente:
  - **Constant Folding**: evaluación en tiempo de compilación de expresiones constantes.
  - **Propagación de constantes**.
  - **Eliminación de código muerto**.
  - **Simplificaciones algebraicas y Peephole**.
- El resultado es un archivo `inter.ir` más limpio, compacto y con menos redundancias.

Ejemplos de uso

- Compilar y ejecutar normalmente:

  ```
  make run FILE=examples/example1.ctds
  ```

- Ejecutar con optimizaciones activadas:

```
make run FILE=examples/example1.ctds OPTIMIZER=1
```

- Analizar solo hasta la generación del IR, con debug:

```
make run FILE=examples/example1.ctds TARGET=ir DEBUG=1
```

6.4.3. Impacto de los flags en el optimizador

El flag `OPTIMIZER=1` no solo habilita el módulo `optimizer.c`, sino que también altera el flujo de compilación en tiempo de ejecución:

1. Durante la generación del **AST**, se aplican simplificaciones algebraicas tempranas.
2. En la fase de **código intermedio (IR)**, el optimizador evalúa constantes y elimina instrucciones redundantes.
3. Si el compilador se ejecuta en modo `DEBUG=1`, se muestran los logs internos del optimizador, incluyendo qué expresiones fueron reemplazadas o eliminadas.

En conjunto, estos flags proporcionan un entorno de desarrollo controlado y reproducible, que facilita tanto la depuración como la validación del rendimiento del compilador.

Simplificación algebraica Cumple la misma función que el Peephole, pero en este caso es agregando algunas operaciones extras a las mismas.

6.5. Relevancia y aportes de la fase de optimización

La etapa de optimización del compilador **TDS25** representa un paso clave en la transición de un prototipo funcional hacia un compilador eficiente y competitivo. Si bien las transformaciones implementadas son de carácter **local** (sin análisis global de flujo o dependencias), su impacto en la legibilidad, tamaño y rendimiento del código intermedio es considerable.

Estas optimizaciones:

- Reducen el número de instrucciones TAC generadas.
- Mejoran la correspondencia entre expresiones y código resultante.
- Preparan el terreno para futuras fases de optimización global y generación de código ensamblador real.

En resumen, esta etapa complementa el front–end del compilador consolidando un producto más sólido, coherente y alineado con las prácticas modernas de optimización de compiladores.

7. Conclusiones

7.1. Reflexión general sobre el desarrollo del compilador

El desarrollo del compilador **TDS25** representó una experiencia integral en el diseño e implementación de software de sistemas, que nos permitió aplicar de manera práctica los conocimientos teóricos adquiridos a lo largo de la carrera. A través de las distintas etapas —desde el análisis léxico y sintáctico hasta la generación de código objeto y las optimizaciones finales— logramos comprender en profundidad los principios fundamentales de la construcción de compiladores y la interacción entre sus componentes.

7.2. Desafíos técnicos y aprendizajes alcanzados

Cada fase del proyecto implicó desafíos específicos: la correcta definición de la gramática y los tokens, la administración eficiente de la tabla de símbolos, la validación semántica del código fuente, la generación del código intermedio y la posterior traducción a ensamblador. Estos pasos no solo consolidaron el dominio de las herramientas *lex/flex* y *yacc/bison*, sino también la importancia de un diseño modular, mantenible y extensible.

7.3. Trabajo en equipo y gestión del proyecto

El trabajo colaborativo fue un pilar esencial para alcanzar los objetivos propuestos. La división de tareas, la comunicación constante y la coordinación entre los integrantes permitieron abordar con éxito los distintos desafíos técnicos y conceptuales del proyecto. El proceso de optimización final, además, ofreció una oportunidad de reflexión sobre la eficiencia del código generado y la aplicabilidad de técnicas clásicas de optimización en compiladores reales.

7.4. Resultados obtenidos

Como resultado, el compilador **TDS25** alcanzó un nivel funcional completo, capaz de analizar, traducir y optimizar programas escritos en el lenguaje definido, cumpliendo con los requisitos establecidos. Más allá del producto final, este proyecto permitió afianzar la capacidad analítica, la toma de decisiones fundamentadas y el criterio técnico necesarios para abordar proyectos de software de mayor escala y complejidad.

7.5. Conclusión final

En conclusión, la experiencia obtenida con el desarrollo del compilador **TDS25** no solo fortaleció nuestras competencias técnicas, sino que también promovió una comprensión más profunda del rol del compilador en el ecosistema de desarrollo de software. Este proyecto reafirmó la importancia de la planificación, la documentación y la colaboración en el proceso de ingeniería, consolidando una visión integral sobre el diseño y la optimización de sistemas complejos.