Proyecto

Taller de Diseño de Software

Conti, Bruno
Gonzalez, Juan Cruz
Vollenweider, Erich

Universidad Nacional de Río Cuarto

Índice

1. Entrega 1: Análizador Sintáctico y Léxico **3**
- 1.1. Repositorio y archivos entregados . . . 3
- 1.2. Proceso de trabajo . . . 3
- 1.3. Análisis léxico (Lexer) . . . 3
- 1.4. Análisis sintáctico (Parser) . . . 4
- 1.5. Pruebas y verificación . . . 4
- 1.6. Diagrama del flujo de compilación (parcial) . . . 5

2. Entrega 2.a: Tabla de Símbolos y Árbol AST **6**
- 2.1. Repositorio y archivos de la Entrega 2 . . . 6
- 2.2. División del trabajo . . . 6
- 2.3. Decisiones de diseño . . . 6
- 2.4. Diseño e implementación del Árbol AST . . . 7
- 2.5. Diseño e implementación de la Tabla de Símbolos . . . 7
- 2.6. Problemas conocidos y limitaciones . . . 7
- 2.7. Diagramas ilustrativos . . . 7
  - 2.7.1. Interacción entre el parser, AST y Tabla de Símbolos . . . 7
  - 2.7.2. Ejemplo de expresión en el Árbol AST . . . 8
  - 2.7.3. Ejemplo de uso de la Tabla de Símbolos . . . 8
- 2.8. Archivo principal `main.c` y sistema de compilación . . . 8
- 2.9. Conclusiones . . . 9

3. Entrega 2.b: Análisis Semántico **10**
- 3.1. Cambios principales respecto a la Entrega 2 . . . 10
- 3.2. Chequeos semánticos implementados . . . 10
- 3.3. Pruebas . . . 11
- 3.4. Decisiones de diseño . . . 11
- 3.5. Ejecución de pruebas y organización del proyecto . . . 11
- 3.6. Conclusiones . . . 12

4. Entrega 3: Generación de Código Intermedio **13**
- 4.1. Repositorio y archivos de la Entrega 3 . . . 13
- 4.2. Objetivo y concepto . . . 13
- 4.3. Diseño general del generador de código . . . 13
- 4.4. Estructura de datos para las instrucciones . . . 14
- 4.5. Ejemplo ilustrativo . . . 14
- 4.6. Problemas conocidos y limitaciones . . . 15
- 4.7. Pruebas realizadas . . . 15
- 4.8. Conclusiones . . . 15

5. Entrega 4: Generación de Código Objeto (Assembler) **16**
- 5.1. Repositorio y archivos de la Entrega 4 . . . 16
- 5.2. Mejoras al código intermedio . . . 16
- 5.3. Gestión de offsets y variables locales . . . 16
- 5.4. Convención de llamadas (Calling Convention) . . . 17
- 5.5. Soporte para funciones externas . . . 17
- 5.6. Restricciones en variables globales . . . 18

5.7. Características del código assembler generado 18
5.8. Refactorización y modularización 19
5.9. Sistema de pruebas extendido 19
5.10. Ejemplo completo de generación 20
5.11. Problemas conocidos y limitaciones 22
5.12. Conclusiones 22

1. Entrega 1: Análizador Sintáctico y Léxico

En esta primera entrega se entregan el analizador léxico (*lexico.l*) y el analizador sintáctico (*parser.y*) para el lenguaje **TDS25**. El equipo trabajó de forma colaborativa: primero se implementó el lexer, luego el parser y finalmente los tests. Algunos tests pasan correctamente, mientras que otros no lo hacen de forma intencional, ya que están diseñados para probar casos fuera del alcance de esta primera entrega (por ejemplo, errores semánticos o características que se implementarán más adelante). En consecuencia, los resultados observados son los esperados para el estado actual del proyecto.

1.1. Repositorio y archivos entregados

- Repositorio del proyecto: `https://github.com/brunocontii/proyecto-compiladores`
- Rama con la primera entrega: `lexico-sintactico`. En esa rama se encuentra todo lo relativo a esta etapa.
- En la rama `lexico-sintactico` hay un `README.md` que explica cómo compilar y ejecutar el **lexer**, el **parser** y los respectivos **tests**.
- Los dos archivos más importantes para esta entrega están dentro de la carpeta `lexico_sintactico/`:
  - `lexer.l` (scanner — Flex)
  - `parsar.y` (parser — Bison)

1.2. Proceso de trabajo

Durante el desarrollo de esta primera entrega seguimos un flujo de trabajo colaborativo y secuencial. Primero nos concentramos en construir el analizador léxico, luego avanzamos con el analizador sintáctico y finalmente preparamos y ejecutamos un conjunto de pruebas para verificar el funcionamiento de ambos componentes. A continuación se describen brevemente cada una de estas fases.

1.3. Análisis léxico (Lexer)

El analizador léxico fue implementado en el archivo `lexer.l` utilizando **Flex**. Entre los aspectos más destacados de su diseño se encuentran:

- Reconocimiento de **palabras reservadas**: `program`, `integer`, `bool`, `void`, `extern`, `return`, `if`, `else`, `then`, `while`, `true`, `false`.
- Soporte para **operadores lógicos y aritméticos**: `&&`, `||`, `!`, `+`, `-`, `*`, `/`, `%`, así como los operadores de comparación `=`, `==`, `<`, `>`.
- Reconocimiento de **delimitadores y símbolos de agrupación**: `( ) { } ; ,`.
- Manejo de **identificadores** (letras, números y guión bajo, iniciando con letra) y **constantes enteras positivas**.
- Ignora espacios en blanco, tabulaciones, saltos de línea y comentarios de línea (`//`) o de bloque (`/* ... */`).

- Estrategia de **depuración**: cada token reconocido se imprime junto a su lexema, lo cual facilita verificar la correcta ejecución.
- **Manejo de errores léxicos**: si se detecta un caracter no válido, se informa en pantalla y el programa finaliza.

El lexer constituye el primer paso del compilador, transformando el código fuente en un flujo de tokens que alimenta al parser en la etapa siguiente.

1.4. Análisis sintáctico (Parser)

El analizador sintáctico fue implementado en el archivo `parser.y` utilizando **Bison**. Sus características principales son:

- Definición de todos los **tokens** producidos por el lexer, garantizando la integración entre ambas fases.
- Especificación de **precedencia y asociatividad** de los operadores mediante directivas de Bison, respetando el orden definido en la especificación del lenguaje (por ejemplo: unario `-` y `!` tienen mayor precedencia que multiplicación/división, etc.).
- La regla inicial `prog` establece la estructura global de un programa: debe comenzar con la palabra reservada `program` y contener entre llaves las declaraciones de variables y/o métodos.
- Soporte para **declaraciones de variables** con asignación y **declaraciones de métodos**, incluyendo parámetros, tipo de retorno y la opción de declarar funciones externas con la palabra clave `extern`.
- Definición de **sentencias** básicas: asignaciones, llamadas a métodos, estructuras de control (`if/then/else`, `while`), sentencias `return`, bloques anidados y sentencias vacías.
- Manejo completo de **expresiones**: literales enteros y booleanos, identificadores, llamadas a funciones, operaciones aritméticas, lógicas, de comparación, así como operadores unarios (menos y negación lógica).
- Inclusión de **bloques** (`{ ... }`) con variables locales y/o secuencias de sentencias.
- **Manejo de errores sintácticos**: se implementó la función `yyerror()`, que informa la línea donde se produjo el error, facilitando la depuración:

  ```
  -> ERROR Sintactico en la linea: N
  ```

En conjunto, el parser valida que la secuencia de tokens producida por el lexer cumpla con la gramática del lenguaje TDS25 y prepara la base para las etapas posteriores.

1.5. Pruebas y verificación

Para esta entrega se preparó un conjunto de casos de prueba en la carpeta `tests/`. Estos casos se dividen en **tests positivos** (que deben pasar correctamente) y **tests negativos** (diseñados para fallar de forma intencional, validando la detección de errores sintácticos).

Tests positivos

- `test1.ctds`
- `test2.ctds`
- `test3.ctds`

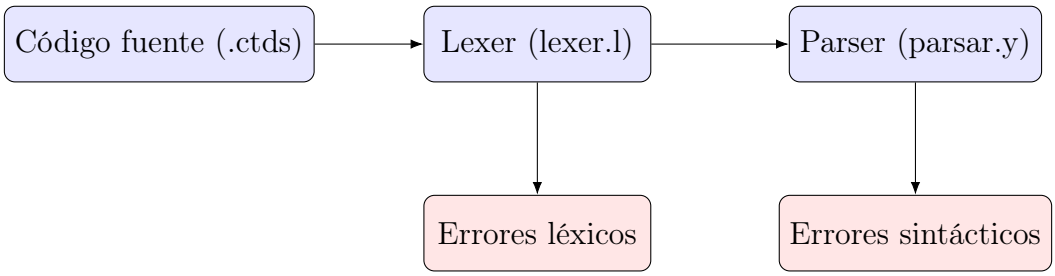
Estos archivos contienen programas válidos en TDS25. El lexer y el parser procesan estos casos sin errores y generan la salida esperada.

Tests negativos (fallan intencionalmente)

- `testneg1.ctds`: contiene una declaración de variable después de un método. Según la gramática, primero deben declararse todas las variables y luego los métodos.
- `testneg2.ctds`: no incluye la definición obligatoria de `program`.
- `testneg3.ctds`: define un método que no posee ni un bloque ni la palabra clave `extern`.
- `testneg4.ctds`: dentro de un bloque aparecen sentencias antes de las declaraciones, lo cual viola la regla de que deben declararse primero las variables.

1.6. Diagrama del flujo de compilación (parcial)

El siguiente diagrama ilustra el flujo de trabajo de las primeras etapas del compilador implementadas en esta entrega:



— Fin de la documentación de la Entrega 1 —

2. Entrega 2.a: Tabla de Símbolos y Árbol AST

Fecha de entrega: 24 de septiembre de 2025

En esta segunda entrega se desarrollaron e integraron dos estructuras fundamentales para el compilador: la **tabla de símbolos** y el **árbol de sintaxis abstracta (AST)**, junto con un controlador principal que unifica todo el proceso de compilación y un sistema de pruebas automatizado. Esta entrega representa un avance significativo hacia un compilador funcional, estableciendo las bases para futuras etapas como el análisis semántico y la generación de código.

2.1. Repositorio y archivos de la Entrega 2

- El código completo de esta entrega se encuentra en la rama: `ts_y_ast`
- En dicha rama hay un `README.md` con instrucciones detalladas para compilar, ejecutar y probar todas las funcionalidades implementadas
- Los archivos principales desarrollados incluyen las implementaciones del AST, tabla de símbolos, controlador principal y sistema de pruebas

2.2. División del trabajo

El equipo se organizó de la siguiente manera:

- **Implementación del Árbol AST**: construcción de las estructuras de nodos, funciones auxiliares para creación, impresión y liberación de memoria.
- **Implementación de la Tabla de Símbolos**: diseño de la estructura de *scopes* anidados, inserción, búsqueda y manejo de símbolos.
- **Integración y pruebas**: validación de la interacción entre AST y tabla de símbolos en casos de prueba positivos y negativos.

2.3. Decisiones de diseño

- Se definió un tipo `info` compartido entre el AST y la tabla de símbolos, que centraliza la información relevante de cada nodo o símbolo (tipo, token, nombre, valores asociados).
- El AST se modeló como una estructura de nodos con hasta tres hijos (`izq`, `med`, `der`), lo que permite representar tanto árboles binarios (expresiones aritméticas) como ternarios (condiciones con else).
- La tabla de símbolos se implementó mediante **listas enlazadas por scope**, organizadas jerárquicamente en forma de pila (stack de scopes), facilitando la apertura y cierre de alcances.
- Se decidió que al cerrar un scope solo se libera la estructura del scope, pero no los símbolos en sí, ya que estos pueden ser referenciados desde el AST.

2.4. Diseño e implementación del Árbol AST

- Cada nodo contiene un puntero a una estructura `info`, con campos para números, identificadores, operadores y tipo de dato.
- Se implementaron funciones de utilidad:
  - `crearNodo`, `crearArbol`, `crearArbolTer` para la construcción de nodos.
  - `mostrarArbol` para imprimir el árbol en formato indentado, útil en depuración.
  - `liberarArbol` para liberar memoria dinámica, contemplando casos específicos según el tipo de nodo.
- Esta representación permite recorrer y analizar el código fuente de manera estructurada, además de servir como base para futuras etapas (generación de código intermedio, optimización).

2.5. Diseño e implementación de la Tabla de Símbolos

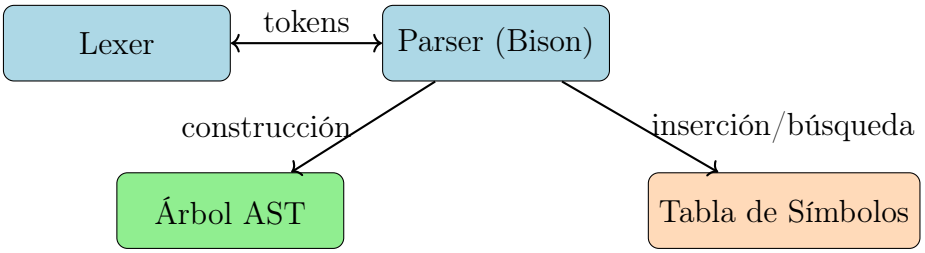
- Se utilizó un enfoque basado en **scopes anidados**, donde cada scope mantiene su propia lista de símbolos y un puntero a su scope padre.
- Funciones principales:
  - `inicializar`: crea el scope global.
  - `abrir_scope` y `cerrar_scope`: permiten el manejo de bloques y funciones.
  - `insertar`: agrega símbolos en el scope actual, validando duplicados.
  - `buscar`: realiza búsquedas desde el scope actual hacia scopes superiores.
  - `imprimir_scope_actual`: imprime los símbolos declarados en el scope actual (útil para depuración).
- La tabla permite manejar correctamente el **alcance de las variables**, evitando colisiones y detectando declaraciones múltiples en un mismo bloque.

2.6. Problemas conocidos y limitaciones

- El AST aún no contempla optimizaciones (por ejemplo, division estructural entre nodo binario y ternario).
- No se implementó verificación de tipos entre operaciones en esta etapa (ejemplo: suma entre enteros y booleanos).

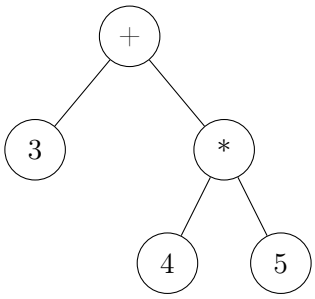
2.7. Diagramas ilustrativos

2.7.1. Interacción entre el parser, AST y Tabla de Símbolos



2.7.2. Ejemplo de expresión en el Árbol AST

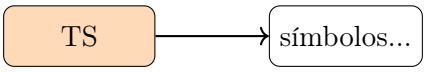
A continuación se muestra cómo se representa en el AST la expresión simple: `3 + 4 * 5`.



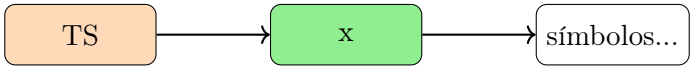
2.7.3. Ejemplo de uso de la Tabla de Símbolos

Supongamos la declaración: `integer x = 10;` A continuación se muestra el estado de la tabla de símbolos antes y después de procesar esta declaración:

Antes:



Después de leer `integer x = 10`:



2.8. Archivo principal `main.c` y sistema de compilación

En esta entrega también se incorporó un archivo `main.c`, que centraliza la ejecución de las distintas etapas del compilador. Desde este archivo se maneja la interacción con el `lexer`, el `parser`, el AST y la tabla de símbolos.

Características principales

- **Gestión de opciones por línea de comandos**: se definieron banderas para controlar el comportamiento del compilador:
  - `-o <salida>` : nombre del archivo de salida.
  - `-target <etapa>` : permite seleccionar la etapa a ejecutar (`lex`, `parse`, `sem`, `codinter`, `assembly`).
  - `-opt <opciones>` : reservado para futuras optimizaciones.
  - `-debug` : activa información adicional de depuración (todavia no funciona correctamente).
- **Manejo de targets**: según la opción pasada por parámetro, el compilador ejecuta únicamente el lexer, el parser o etapas posteriores (por ahora simuladas).
- **Integración con AST y tabla de símbolos**:
  - Se inicializa la tabla de símbolos global antes de comenzar el parseo.

    - Una vez construido, el AST puede mostrarse en consola o exportarse en formato DOT para generar diagramas.

- **Coloreado de mensajes**: se incorporaron códigos ANSI para imprimir mensajes en colores (rojo para errores, verde para éxito, amarillo para advertencias).

Ejecución de pruebas

Además, se añadió al `Makefile` la regla `make test-all`, que automatiza la validación del compilador. Esta regla recorre todos los archivos dentro de la carpeta `tests/`, ejecuta el compilador sobre cada uno de ellos y muestra en pantalla un reporte indicando:

- cuáles pasaron correctamente (en verde),
- cuáles fallaron como estaba previsto (en rojo o amarillo).

De esta manera, el proceso de pruebas se simplifica y se garantiza una retroalimentación rápida sobre el estado de la implementación.

2.9. Conclusiones

Con esta entrega, el compilador cuenta con una representación estructurada del programa (AST) y un mecanismo de manejo de identificadores (tabla de símbolos). Ambas estructuras serán fundamentales para las próximas etapas, como el análisis semántico y la generación de código intermedio.

— Fin de la documentación de la Entrega 2.a —

3. Entrega 2.b: Análisis Semántico

Fecha de entrega: 1 de octubre de 2025

En esta entrega se extendió el compilador incorporando un **módulo de análisis semántico**, encargado de verificar la corrección del programa más allá de la sintaxis. Esta etapa garantiza que las construcciones del lenguaje tengan sentido, previniendo errores de tipos, parámetros y uso indebido de métodos o variables. Todo el desarrollo de esta entrega se encuentra en la rama `analisis-semantico` del repositorio.

3.1. Cambios principales respecto a la Entrega 2

- Se implementó un nuevo archivo `semantico.c`, donde se recorre el AST y se realizan todos los chequeos semánticos.
- En `parser.y` se eliminó toda lógica relacionada con la tabla de símbolos. El parser ahora se limita únicamente a construir el AST, dejando los chequeos a `semantico.c`.
- Se creó la carpeta `utils/`, destinada a concentrar funciones auxiliares de uso general. Actualmente incluye rutinas para:
  - Reporte uniforme de errores semánticos.
  - Chequeo de tipos de expresiones.
  - Verificación de asignaciones a métodos.
  - Validación de parámetros actuales frente a los formales.
- Se agregó la función `buscarNodo` en el manejo del AST, que permite localizar nodos específicos. Esta función es utilizada para el chequeo semántico y, en un futuro, se podría optimizar.

3.2. Chequeos semánticos implementados

Los principales chequeos realizados durante esta etapa son los siguientes:

- **Compatibilidad de tipos en declaraciones de variables**.
- **Consistencia en el tipo de retorno de los métodos**: el valor retornado debe coincidir con el tipo declarado.
- **Verificación de parámetros de métodos**: se comprueba la cantidad y los tipos de cada parámetro actual en comparación con los parámetros formales.
- **Restricciones en invocación de métodos**:
  - Métodos con tipo `integer` o `bool` deben usarse siempre como expresiones (no pueden invocarse "sin más").
  - Métodos `void` pueden invocarse como sentencias independientes.
- **Método principal (`main`)**: debe existir un único método llamado `main`, con las siguientes condiciones:
  - Tipo de retorno `void`.

    - Sin parámetros.
    - No retorna valores.
- **Detección de redeclaración de variables y métodos**.
- **Compatibilidad de tipos en expresiones aritméticas, booleanas y de comparación**.

3.3. Pruebas

Al igual que en las entregas anteriores, se incorporaron nuevos casos de prueba en la carpeta `tests/`. Estos incluyen:

- **Tests positivos**: programas bien formados que cumplen todas las restricciones semánticas.
- **Tests negativos**: programas diseñados para fallar al detectar:
    - Incompatibilidad de tipos en expresiones.
    - Uso incorrecto de métodos según su tipo de retorno.
    - Parámetros incorrectos en la invocación de métodos.
    - Variables o métodos redeclarados.
    - Definición incorrecta o ausencia del `main`.

3.4. Decisiones de diseño

- Separar completamente las responsabilidades entre el parser y el análisis semántico: el primero construye el AST y el segundo lo recorre para aplicar las reglas semánticas.
- Centralizar funciones auxiliares en una carpeta `utils/`, con el fin de mejorar la organización y la reutilización de código.

3.5. Ejecución de pruebas y organización del proyecto

Nuevas opciones del `Makefile`

Con el objetivo de simplificar la validación del compilador, se incorporaron nuevas reglas al `Makefile`:

- `make test-all`: ejecuta todos los tests presentes en la carpeta `tests/`, abarcando tanto los sintácticos como los semánticos.
- `make test-sintactico`: ejecuta únicamente los archivos de prueba ubicados en `tests/tests-sintactico/`. En este caso, el compilador se invoca con la opción `-target parse`, de manera que solo se alcanza la etapa de construcción del AST.
- `make test-semantico`: ejecuta únicamente los archivos ubicados en `tests/tests-semantico/`, permitiendo validar los chequeos implementados en el módulo semántico.

Estas reglas automatizan el proceso de validación, mostrando en consola qué casos pasaron correctamente y cuáles fallaron como se esperaba.

Reorganización de la estructura de carpetas

Para mejorar la mantenibilidad y la claridad del proyecto, se reorganizó la carpeta `tests/` en subdirectorios específicos:

- `tests/tests-sintactico/`: contiene los casos diseñados para probar la construcción del AST y las validaciones puramente sintácticas.
- `tests/tests-semantico/`: contiene los casos diseñados para verificar los distintos chequeos semánticos implementados en esta entrega.

Esta división facilita la ejecución selectiva de pruebas y mantiene el repositorio mejor organizado.

3.6. Conclusiones

Con esta entrega, el compilador alcanza un nivel en el cual no solo reconoce la estructura sintáctica del programa, sino también su validez semántica. Esto permite detectar errores más cercanos a los que un programador real cometería al escribir código. Además, la separación de responsabilidades y la inclusión de la carpeta `utils/` sientan las bases para una implementación más modular y mantenible en las próximas etapas.

— Fin de la documentación de la Entrega 2.b —

4. Entrega 3: Generación de Código Intermedio

Fecha de entrega: 8 de octubre de 2025

En esta entrega se implementó la etapa de **generación de código intermedio**, cuyo objetivo es traducir el Árbol de Sintaxis Abstracta (AST) a una representación más cercana al lenguaje máquina, denominada **código de tres direcciones** Esta etapa constituye el puente entre el análisis semántico y la futura generación de código assembler real.

4.1. Repositorio y archivos de la Entrega 3

- Rama correspondiente: `codigo-intermedio`
- Archivos principales:
  - `generador.h`: define las estructuras de datos, constantes y prototipos para la generación del código intermedio.
  - `generador.c`: implementa la lógica principal que recorre el AST, detecta los patrones de cada tipo de nodo y produce las instrucciones de tres direcciones.
  - `codigo-intermedio.txt`: archivo de salida donde se almacena el código generado.

4.2. Objetivo y concepto

El propósito del código intermedio es disponer de una representación independiente de la arquitectura, que capture la lógica del programa en una secuencia de operaciones elementales. Cada instrucción tiene la forma general:

```
instruccion resultado argumento1 argumento2
```

Por ejemplo, la expresión `a = b + c * d` podría traducirse como:

```
LOAD t0 d
LOAD t1 c
MUL t2 t1 t0
LOAD t3 b
ADD t4 t3 t2
ASSIGN a t4
```

Este enfoque permite simplificar la generación de código assembler en la siguiente etapa.

4.3. Diseño general del generador de código

El generador recorre el AST de manera recursiva y, según el tipo de nodo, produce una o más instrucciones intermedias. Cada nodo se analiza buscando patrones reconocibles (por ejemplo: operaciones binarias, asignaciones, sentencias de control, etc.).

- Los nodos `T_OP_MAS`, `T_OP_MENOS`, `T_OP_MULT`, `T_OP_DIV`, etc., generan instrucciones aritméticas.

- Las asignaciones producen una instrucción de tipo `ASSIGN` o `MOV`.
- Las estructuras de control (`if`, `while`) generan etiquetas (`L1`, `L2`, ...) y saltos condicionales o incondicionales.
- Los métodos y funciones definen bloques delimitados por etiqueta de entrada.

4.4. Estructura de datos para las instrucciones

Para representar las instrucciones de tres direcciones se definió una estructura en `generador.h`:

```
typedef struct codigo3dir {
    char instruccion[25];
    char resultado[10];
    char argumento1[10];
    char argumento2[20];
}codigo3dir;
```

Todas las instrucciones generadas se almacenan en un arreglo global de tipo `codigo3dir`, con un índice que se incrementa a medida que se agregan nuevas líneas de código.

Este arreglo cumple una doble función:

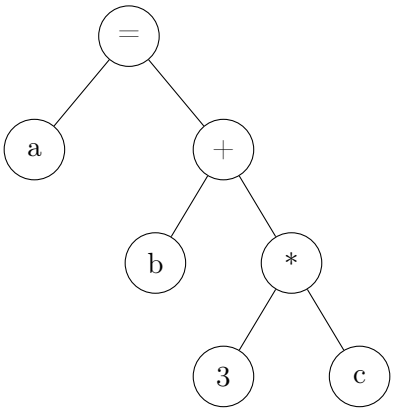
1. Permitir escribir el código intermedio en un archivo de texto (`codigo-intermedio.txt`).
2. Ser reutilizado en la próxima entrega para generar código assembler real.

4.5. Ejemplo ilustrativo

Para la expresión simple:

```
a = b + 3 * c
```

Representación en el Árbol AST



Traducción a código intermedio

Instrucción	Resultado	Argumento1	Argumento2
LOAD	t0	c	-
LOAD	t1	3	-
MUL	t2	t0	t1
LOAD	t3	b	-
ADD	t4	t3	t2
ASSIGN	a	t4	-

Al finalizar el proceso, se crea el archivo `codigo-intermedio.txt`, que contiene las instrucciones en formato legible. Cada línea sigue el patrón:

```
<instr> <res> <arg1> <arg2>
```

4.6. Problemas conocidos y limitaciones

- **Estructura de almacenamiento de instrucciones:** actualmente se utiliza un arreglo para almacenar las instrucciones del código de tres direcciones. Esta solución es simple y rápida de indexar.En futuras iteraciones conviene evaluar estructuras más flexibles (por ejemplo, listas enlazadas para reducir costes de inserción y mejorar la escalabilidad).
- **Uso ineficiente de temporales:** el generador crea más temporales de los estrictamente necesarios.
- **Generación independiente de arquitectura:** el código producido es un pseudo-assembler (representación de tres direcciones) y no está pensado para ejecutarse tal cual en una arquitectura real. Es una representación intermedia cuya transformación a assembler real requiere una etapa adicional.

4.7. Pruebas realizadas

No se crearon nuevos tests específicos ni carpetas adicionales para la generación de código intermedio. El flujo de pruebas se apoya en los tests ya existentes: cuando un test pasa correctamente la etapa de análisis semántico, el generador produce automáticamente el archivo de salida `codigo-intermedio.txt` correspondiente.

La verificación actual se realiza de forma manual/visual comparando el contenido de `codigo-intermedio.txt` con el resultado esperado para cada programa.

4.8. Conclusiones

Con la generación del código intermedio, el compilador ya puede traducir programas escritos en TDS25 a una representación más baja y manipulable. Esta etapa marca un punto clave, ya que el código intermedio será la base sobre la cual se implementará la próxima fase: la generación de **código assembler real**.

— Fin de la documentación de la Entrega 3 —

5. Entrega 4: Generación de Código Objeto (Assembler)

Fecha de entrega: 27 de septiembre de 2025

En esta entrega se implementó la etapa final de **generación de código objeto**, traduciendo el código intermedio de tres direcciones a **código assembler x86-64** funcional que sigue la convención de llamadas estándar de Linux. Esta etapa representa la culminación del proceso de compilación, produciendo ejecutables válidos a partir de programas escritos en TDS25.

5.1. Repositorio y archivos de la Entrega 4

- Rama correspondiente: `assembler`
- Rama adicional: `refactorizacion-modularizacion` (reorganización del código una vez validado el funcionamiento)
- Archivos principales:
  - `assembler.h` y `assembler.c`: implementan la lógica de generación de código assembler a partir del código intermedio.
  - `assembler.s`: archivo de salida con el código assembler generado.
  - `func-extern.c`: ejemplo de funciones externas escritas en C que pueden ser invocadas desde programas TDS25.

5.2. Mejoras al código intermedio

Durante el desarrollo de esta etapa, se identificaron limitaciones en la representación del código de tres direcciones que dificultaban la generación correcta de assembler. Para resolverlas, se realizaron los siguientes cambios:

- **Cambio en la estructura de instrucciones**: los campos de la estructura `codigo3dir` que antes eran simples strings ahora son punteros a estructuras `info` del árbol AST. Esto permite acceder directamente a información de tipos, alcance y otros atributos necesarios para la generación de código.
- **Sustitución de arreglos por listas enlazadas**: estructuras que antes utilizaban arreglos con tamaño fijo (como la lista de instrucciones) fueron reemplazadas por listas enlazadas dinámicas. Esto mejora la flexibilidad y permite liberar memoria adecuadamente tras su uso.

Estas modificaciones mejoraron significativamente la capacidad del generador para producir código assembler correcto y completo.

5.3. Gestión de offsets y variables locales

Uno de los desafíos principales fue la generación de offsets válidos para las distintas entidades del programa (variables locales, temporales y parámetros). La solución implementada fue:

- Dentro de cada método, se tratan de manera uniforme las variables locales, las variables temporales y los parámetros.
- Todas estas entidades se almacenan en una lista enlazada que luego se recorre para asignar offsets relativos al frame pointer (`%rbp`).
- Este enfoque garantiza que cada variable tenga una posición única y correcta en el stack frame del método.

5.4. Convención de llamadas (Calling Convention)

Se implementó la **convención de llamadas estándar de System V AMD64 ABI** utilizada en Linux, que establece:

- Los primeros 6 argumentos de una función se pasan en registros: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`.
- Los argumentos adicionales (a partir del séptimo) se pasan a través del stack.
- El método llamador (*caller*) es responsable de colocar los argumentos en los registros o en el stack antes de la llamada.
- El método llamado (*callee*) puede acceder directamente a estos valores siguiendo la convención.

Esta implementación permite:

- Interoperabilidad con funciones externas escritas en C u otros lenguajes que sigan la misma convención.
- Llamadas correctas entre métodos del mismo programa TDS25.

5.5. Soporte para funciones externas

Gracias a la convención de llamadas implementada, el compilador puede invocar funciones externas definidas en otros lenguajes. Como demostración, se creó el archivo `func-extern.c` que contiene funciones escritas en C:

```
// func-extern.c
#include <stdio.h>

void print_int(long long i) {
    printf("%lld\n", i);
    fflush(stdout);
}
```

Desde un programa TDS25, es posible declarar e invocar esta función usando la palabra clave `extern`:

```
program {
    extern void print_int(integer x);

    void main() {
        integer x = 10;
        imprimir(10);
    }
}
```

El assembler generado incluye únicamente la instrucción `call print_int`, sin generar el cuerpo de la función. El enlazador se encarga de resolver la referencia al compilar junto con `func-extern.c`.

5.6. Restricciones en variables globales

Se modificó el análisis semántico para restringir la inicialización de variables globales:

- **Antes**: las variables globales podían inicializarse con literales, expresiones o llamadas a funciones.
- **Ahora**: las variables globales solo pueden inicializarse con **valores literales** (números enteros o los valores booleanos `true`/`false`).
- Si se intenta inicializar una variable global con una expresión o llamada a función, se reporta un error semántico.

Esta decisión simplifica la generación de la sección `.data` del assembler y evita problemas de evaluación en tiempo de compilación.

5.7. Características del código assembler generado

El assembler producido tiene las siguientes propiedades:

- **Funcional**: el código generado es ejecutable y produce resultados correctos.
- **No optimizado**: se priorizó la corrección sobre la eficiencia. El código incluye instrucciones redundantes y no aprovecha todas las optimizaciones posibles.
- **Estructura clara**: cada método genera un prólogo y epílogo estándar para el manejo del stack frame.
- **Compatibilidad**: el código sigue las convenciones de Linux x86-64 y puede enlazarse con código C.

La decisión de no optimizar en esta etapa es intencional: la siguiente entrega se enfocará en aplicar técnicas de optimización sobre el código ya funcional.

5.8. Refactorización y modularización

Una vez validado el funcionamiento del generador de assembler y confirmado que todos los tests pasaban correctamente, se realizó una refactorización integral del código:

- Se reorganizó el código en módulos más pequeños y especializados.
- Se mejoraron los nombres de funciones y variables para aumentar la legibilidad.
- Se documentaron las funciones principales con comentarios descriptivos.
- Se separaron responsabilidades entre distintos archivos fuente.

Esta refactorización se encuentra en la rama `refactorizacion-modularizacion` y tiene como objetivos:

- Facilitar el mantenimiento del código.
- Simplificar la implementación de optimizaciones en la siguiente etapa.
- Mejorar la comprensión del proyecto para futuras extensiones.

5.9. Sistema de pruebas extendido

Consideraciones sobre la ejecución de tests

Es importante destacar que, dado el carácter incremental del compilador, algunos tests son válidos únicamente para ciertas etapas. Un programa puede ser sintácticamente correcto pero fallar en el análisis semántico, o ser semánticamente válido pero diseñado para probar características específicas de la generación de código.

Por esta razón, la forma más adecuada de validar el funcionamiento del compilador es ejecutar manualmente cada test especificando la etapa objetivo:

```
make
./c-tds -target <etapa> <archivo_test>
```

Esto permite verificar que cada fase funciona correctamente de manera independiente.

Es importante tener presente esta recomendación, ya que las reglas de conveniencia del `Makefile` utilizan la etapa `assembly` por defecto:

- `make run`: ejecuta un test predeterminado con la etapa `assembly`.
- `make run TEST=<test>`: ejecuta el test especificado, pero también con la etapa `assembly` por defecto.

Por lo tanto, si se desea validar una etapa específica (como `parse` o `sem`), es necesario invocar el compilador directamente con el flag `-target` correspondiente.

Automatización de pruebas

Se incorporó una nueva regla al `Makefile` para facilitar la validación del código assembler:

- `make test-assembler`: compila y ejecuta todos los tests relacionados con la generación de código objeto, mostrando un reporte detallado de qué casos pasaron y cuáles fallaron.

Los tests cubren una amplia variedad de casos:

- Estructuras de control: ciclos `while`, condicionales `if-then-else`.
- Funciones recursivas: validación de la correcta gestión del stack.
- Funciones con distintas cantidades de parámetros: 0, 1, 6 (todos en registros), y más de 6 (combinando registros y stack).
- Funciones externas: invocación de código C desde TDS25.
- Expresiones aritméticas y booleanas complejas.

Todos los tests pasan correctamente, validando la robustez del generador de código.

5.10. Ejemplo completo de generación

Código fuente TDS25

```
program {

    integer factorial(integer x) {

        if (x < 0) then {
            return -1;
        }

        if (x < 2) then {
            return 1;
        }

        return x * factorial(x-1);
    }

    void print_int(integer x) extern;
    void print_bool(bool a) extern;

    void main() {
        integer x = factorial(5);
        if (x == 120) then{
            print_bool(true);
        } else {
            print_bool(false);
```

```
        }
    }
}
```

Fragmento del assembler generado

```
factorial:
    pushq %rbp
    movq %rsp, %rbp
    subq $64, %rsp
    movq %rdi, -56(%rbp)
    movq -56(%rbp), %rax
    cmpq $0, %rax
    setl %al
    movzbl %al, %eax
    movq %rax, -48(%rbp)
    movq -48(%rbp), %rax
    cmpq $0, %rax
    je L0
    movq $1, %rax
    negq %rax
    movq %rax, -40(%rbp)
    movq -40(%rbp), %rax
    movq %rbp, %rsp
    popq %rbp
    ret
L0:
    movq -56(%rbp), %rax
    cmpq $2, %rax
    setl %al
    movzbl %al, %eax
    movq %rax, -32(%rbp)
    movq -32(%rbp), %rax
    cmpq $0, %rax
    je L1
    movq $1, %rax
    movq %rbp, %rsp
    popq %rbp
    ret
L1:
    movq -56(%rbp), %rax
    subq $1, %rax
    movq %rax, -24(%rbp)
    movq -24(%rbp), %rdi
    call factorial
    movq %rax, -16(%rbp)
    movq -56(%rbp), %rax
    imulq -16(%rbp), %rax
    movq %rax, -8(%rbp)
```

```
    movq -8(%rbp), %rax
    movq %rbp, %rsp
    popq %rbp
    ret
main:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $5, %rdi
    call factorial
    movq %rax, -24(%rbp)
    movq -24(%rbp), %rax
    movq %rax, -16(%rbp)
    movq -16(%rbp), %rax
    cmpq $120, %rax
    sete %al
    movzbl %al, %eax
    movq %rax, -8(%rbp)
    movq -8(%rbp), %rax
    cmpq $0, %rax
    je L2
    movq $1, %rdi
    call print_bool
    jmp L3
L2:
    movq $0, %rdi
    call print_bool
L3:
    movq %rbp, %rsp
    popq %rbp
    ret
```

5.11. Problemas conocidos y limitaciones

- **Código no optimizado**: se generan más instrucciones de las necesarias, con movimientos redundantes entre memoria y registros.
- **Uso limitado de registros**: no se implementó asignación eficiente de registros, por lo que muchos valores se almacenan en memoria incluso cuando podrían mantenerse en registros.
- **Manejo básico de expresiones**: cada subexpresión se evalúa de forma independiente sin aprovechar optimizaciones algebraicas.

Estas limitaciones serán abordadas en la siguiente entrega, enfocada en optimizaciones.

5.12. Conclusiones

Con esta entrega, el compilador TDS25 alcanza su funcionalidad completa como generador de código ejecutable. El código assembler producido es correcto, sigue las con-

venciones estándar de Linux, y puede interoperar con código escrito en otros lenguajes. La refactorización realizada deja el proyecto en un estado sólido para incorporar optimizaciones en la próxima etapa.

— Fin de la documentación de la Entrega 4 —