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

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`

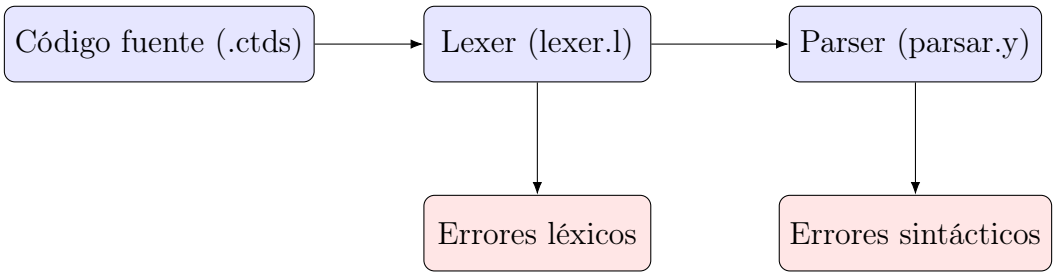
Estos archivos contienen programas válidos en TDS25. El lexer y el parser procesan estos casos sin errores y generan la salida esperada.

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: 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

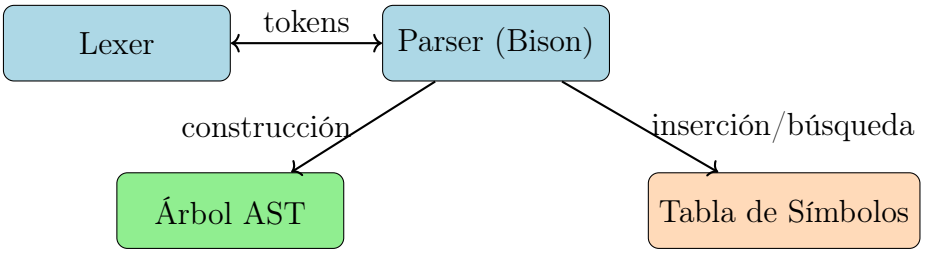
- Se utilizó un enfoque basado en **scopes anidados**, donde cada scope mantiene su propia lista de símbolos y un puntero a su scope padre.
- Funciones principales:
  - `inicializar`: crea el scope global.
  - `abrir_scope` y `cerrar_scope`: permiten el manejo de bloques y funciones.
  - `insertar`: 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).

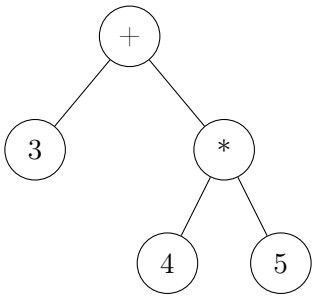
2.7. Diagramas ilustrativos

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



2.7.2. Ejemplo de expresión en el Árbol AST

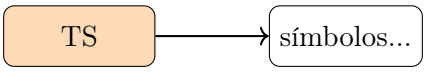
A continuación se muestra cómo se representa en el AST la expresión simple: `3 + 4 * 5`.



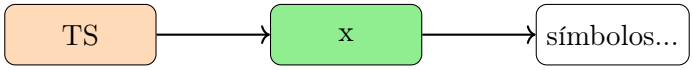
2.7.3. Ejemplo de uso de la Tabla de Símbolos

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 —