Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas, Físico - Químicas y Naturales
Departamento de Computación

TDS25 - Documentación del Proyecto

Autores: Jeremias Avaro, Mateo Cornejo, Maximo Marquez Regis

Materia: Taller de Diseño de Software

Docente: Francisco Bavera

Año: 2025

Índice

1. Introducción

Este documento recopila las distintas etapas de desarrollo del compilador realizado como proyecto académico. Se detallan las tareas llevadas a cabo por cada integrante, las decisiones de diseño adoptadas (y aclaraciones), decisiones clave y y los problemas detectados.

El objetivo es documentar de forma progresiva la evolución del compilador, dejando registro tanto de los aspectos técnicos como de la organización del trabajo en equipo.

2. Primera etapa

División de la primera etapa

- **Jeremias Avaro:** inicialización del proyecto y primeros avances del lexer y la gramática.
- **Mateo Cornejo:** refinamiento de lexer y gramática. Creación de tests.
- **Máximo Marquez Regis:** completó la gramática casi en su totalidad. Integración del parser y lexer.
- Todos: discusión de decisiones de diseño, revisión de código y documentación.

Decisiones de diseño y aclaraciones

- Los comentarios se ignoran en el lexer para simplificar la gramática del parser.
- Se optó por separar el manejo de errores en un módulo propio (`error_handling.c/h`) para centralizar la gestión de mensajes.
- Se decidió que el lexer cuente tokens y líneas para facilitar el reporte de errores y estadísticas.

Diseño y decisiones clave

- La gramática permite declaraciones de variables, funciones y bloques de código.
- Se definió un **union** para manejar tokens con valores enteros o cadenas.
- Las expresiones incluyen operadores aritméticos, relacionales y lógicos con precedencias adecuadas.
- Se implementó un sistema de reporte de errores léxicos y sintácticos.
- El parser está preparado para ser extendido con análisis semántico y generación de código.
- Se utilizó un Makefile para automatizar la compilación y facilitar la integración de nuevos módulos.
- Se crearon tests básicos para validar el funcionamiento del lexer y parser.

Problemas conocidos

- Algunos casos límite no han sido exhaustivamente probados.
- El manejo de comentarios multiline5a podría fallar en comentarios no cerrados.

3. Segunda etapa

División de la segunda etapa

- **Mateo Cornejo:** implementación del AST incluyendo la definición de estructuras y funciones asociadas. Desarrollo de funciones para la visualización del AST. Refactorización del manejo de errores hacia el módulo `error_handling`. Participación en la implementación y correcciones del analizador semántico.
- **Jeremias Avaro:** inicialización de la tabla de símbolos, planteando la estructura base y el mecanismo de scopes. Desarrollo conjunto del analizador semántico.
- **Máximo Marquez Regis:** implementación general de la tabla de símbolos, refinamiento del manejo de scopes y funciones. Correcciones posteriores en el analizador semántico, documentación de código, creación de tests extras.
- Todos: participación en la discusión de decisiones de diseño, testeo y documentación.

Decisiones de diseño y aclaraciones

- Se definió la función `my_strdup` para duplicar strings, evitando el uso de `strdup`.
- Si una función es de tipo `void`, es válido que exista un `return;` o no haya sentencia `return` en el bloque de la función.
- En la estructura `AST_NODE`, se empleó una `union` para almacenar información específica según el tipo de nodo.
- Se diseñó la estructura `AST_NODE_LIST` para manejar listas de sentencias, declaraciones y argumentos.
- Se creó la carpeta `utils` para funciones de utilidad como `my_strdup`.
- Sólo se puede acceder al scope global de la tabla de símbolos, ya que cada id (variable o función) en el AST se almacena un puntero al bloque de la tabla de símbolos perteneciente a dicho id.
- En el AST, cada función tiene un puntero a su bloque en la tabla de símbolos, que a su vez tiene un puntero al scope de la función.
- Se asume que en un archivo existe una y sólo una declaracion de una función `main`, puede ser que esté definido en el mismo archivo o externamente.

Diseño y decisiones clave

- Se creó el enumerado `AST_TYPE` para representar los distintos tipos de nodos.
- Se utilizó una `union` en `AST_NODE` para reducir redundancia en la representación de nodos.
- Se implementó `free_mem` para liberar memoria del AST de manera recursiva (por ahora no se utiliza).
- Se integraron los constructores de nodos en las acciones semánticas del parser.

- Se cambió la gramática a recursión por la izquierda para construir listas de statements y declaraciones, lo que permite almacenarlos e imprimirlos directamente en el orden fuente.
- Se integró la impresión del AST y del scope global tabla de símbolos al final del análisis sintáctico.
- Se incorporó el manejo de **scopes** para soportar bloques anidados.
- Se implementó una pila de scopes (`TABLE_STACK`), donde cada scope mantiene:
  - `head_block` y `end_block`: punteros al inicio y fin de la lista de símbolos declarados en ese ámbito.
  - `up`: puntero al ámbito exterior inmediato, implementando encadenamiento léxico.
- La búsqueda de símbolos recorre desde el scope actual hacia los superiores, logrando visibilidad léxica. Esto permite que variables internas oculten (shadow) a las externas.
- La redeclaración se prohíbe únicamente dentro del mismo scope. En ámbitos anidados se permite el shadowing de símbolos.
- Para soportar recursión y co-recursión, las funciones se predeclaran antes de analizar sus cuerpos. De esta manera, las llamadas recursivas encuentran el símbolo correspondiente.
- Las funciones residen en el ámbito global (ya que no se permite declaraciones de funciones anidadas). Cada bloque de función mantiene una lista de argumentos formales y un puntero al scope asociado.
- El bloque de cada función en la tabla de símbolos almacena una lista con los parámetros formales de dicha función, al momento de realizar el chequeo semántico se utiliza para comparar con las expresiones dadas como parámetro en la llamada a la función.
- Se desarrolló un analizador semántico con una función general `eval`, la cual deriva en evaluadores específicos según el tipo de nodo (`eval_if`, `eval_block`, etc.).
- El proceso de evaluación recorre el AST hasta alcanzar las hojas, obteniendo tipos que luego se utilizan en el chequeo semántico.
- En cada evaluación se conserva la línea de origen (`line = tree->line`) para reportar errores con información precisa.
- En los bloques se registra si se ejecutó un `return`, lo que permite ignorar sentencias posteriores en ese mismo ámbito.
- Los nodos de tipo `DECL` se eliminaron en la etapa de interpretación, ya que su función es únicamente de representación en el AST. Las declaraciones se controlan directamente en la tabla de símbolos.

Problemas conocidos

- Falta testear exhaustivamente la correcta construcción y recorrido del AST con múltiples tipos de nodos.
- El manejo de **scopes** aún no se probó en profundidad, en particular:
  - Casos de sombreado (shadowing) en distintos niveles anidados.

    - Validación de la prohibición de redeclaración dentro del mismo scope.
    - Comportamiento de la predeclaración de funciones recursivas.
- El analizador semántico requiere pruebas adicionales para:
    - Evaluación de bloques con múltiples sentencias y retornos anidados.
    - Consistencia en el reporte de errores con número de línea.
    - Correcto manejo de tipos en expresiones más complejas.