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.

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 multilínea 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.

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.

4. Tercer etapa

División de la tercer etapa

- **Mateo Cornejo:** Implementación de la base del generador de código intermedio y corrección de errores.
- **Jeremias Avaro:** Implementación general del generador de código intermedio.
- **Máximo Marquez Regis:** Refactorización de las estructuras del compilador.

Decisiones de diseño y aclaraciones

- Se optó por cambiar las estructuras del compilador y unificarlas en una para facilitar la generación de código intermedio.
- En la generación de código intermedio se utiliza la estructura `Info` para mantener información sobre variables y operadores.
- Se volvió a agregar la operación `DECL` al AST para facilitar la generación de código intermedio y chequear que el tipo de los ID se corresponda con su declaración.
- Para la generación de código intermedio se recorre el AST de igual manera que el Semantic Analyzer. Cada nodo del árbol sintáctico abstracto es procesado por funciones especializadas según su tipo (expresiones, asignaciones, declaraciones, bloques, etc.).
- Se hace uso de un temporal en todas las instrucciones del código de tres direcciones. En cada operación, se genera un nombre de temporal único (por ejemplo, T1, T2, ...) mediante la función new_temp(). El resultado relevante de la operación se almacena en ese temporal, permitiendo encadenar instrucciones y facilitar la traducción a código máquina o ensamblador posteriormente.
- El buffer de instrucciones (code[ ]) almacena cada instrucción generada, incluyendo los operandos y el temporal asociado.
- Se tomó como decisión que las operaciones (instrucciones) también sean almacenadas en la estructura `Info`, para facilitar la generación de código intermedio.

Diseño y decisiones clave

- Se definió una estructura unificada de datos para el compilador, permitiendo compartir información entre las distintas etapas.
- El diseño del AST se mantuvo modular: cada tipo de nodo (expresiones, asignaciones, declaraciones, etc.) tiene una función de procesamiento específica, lo que facilita la extensión futura del lenguaje.

- Se implementó un generador de temporales (`new_temp()`) y un contador global para asegurar que cada temporal sea único, garantizando trazabilidad y evitando colisiones durante la generación de código.
- El buffer de código (`code[]`) se diseñó como una lista secuencial de instrucciones que pueden recorrerse o traducirse sin necesidad de reconstruir el árbol sintáctico.
- Se decidió volver a incorporar la operación de declaración para los nodos de tipo `common`, ya que no se chequeaba si el tipo de la inicialización era el correcto.

Problemas conocidos

- Al tener una sola estructura para el `AST` y para la tabla de símbolos, se hace más engorroso acceder a los distintos campos de la estructura.

5. Cuarta etapa

División de la cuarta etapa

- **Mateo Cornejo:** Implementación general del generador de código objeto e implementación de tests.
- **Jeremias Avaro:** Implementación general del generador de código objeto.
- **Máximo Marquez Regis:** Implementación general del generador de código objeto.

Decisiones de diseño y aclaraciones

- Se construyó un generador de código objeto para x86, siguiendo convenciones de Linux ABI.
- En la llamada a funciones, se copian todos los parámetros a la memoria para evitar "pisamiento" si hay llamadas a otras funciones.

Diseño y decisiones clave

- Se utiliza la estructura del código intermedio como base para generar el código objeto.
- Se creó una estructura `VarLocation` para almacenar nombres de variables o temporales y su respectivo offset en memoria.
- Al seguir la convención de Linux ABI, para las llamadas a funciones los primeros 6 parámetros se guardan en los registros `%rdi, %rsi, %rdx, %rcx", %r8, %r9`. Si se necesitan más de 6 parámetros, se apilan en el stack.

Problemas conocidos

- Poca eficiencia en el código generado, problema a ser solucionado en la próxima etapa.
- Cuando se invoca a una función con llamadas a otras funciones como parámetros, usualmente se pierden o pisan los valores de algunos parámetros.

6. Quinta etapa

División de la quinta etapa

- **Mateo Cornejo:** Implementación de las optimizaciones propagación de constantes y eliminación de código muerto, refinamiento y culminación de la implementación de las flags y script de test.
- **Jeremias Avaro:** Implementación de optimización de memoria y refinamiento de demás optimizaciones.
- **Máximo Marquez Regis:** Implementación de optimización de división por potencias de 2, optimización de memoria, refinamiento y adaptación de la CLI del compilador.

Decisiones de diseño y aclaraciones

- Se comenzó la implementación de una optimización para parámetros repetidos de una función pero se optó por dejar de implementarla debido a inconsistencias en su funcionamiento.
- Se implementó la optimización de eficiencia de memoria sobre el código intermedio ya que era más simple que realizarla sobre el código objeto.

Diseño y decisiones clave

- Las optimizaciones de propagación de constantes y eliminación de código muerto fueron realizadas sobre el AST.
- Las optimizaciones de división por potencia de 2 y eficiencia de memoria fueron realizadas sobre el código intermedio.
- La segunda parte de la optimización de eficiencia de memoria se había comenzado a implementar sobre el generador de código objeto, pero se decidió realizarla como una "segunda pasada" sobre el código intermedio debido a mayor simplicidad en el manejo de la estructura `INFO` que utiliza el compilador (más simple que realizarlo sobre texto).
- En la optimización de divisiones por potencia de 2, se hace uso de un "bias" para poder truncar a 0 los resultados de las divisiones cuando el dividendo es negativo (si el resultado de la división es -0.5, sin realizar este paso el resultado quedaría en -1). Esta optimización se realiza solamente cuando el divisor es positivo.

Problemas conocidos

- Así como se realizó la optimización de división por potencias de 2, se podría haber realizado de igual manera una optimizacion para multiplicaciones por potencias de 2.
- No fue testeado en su totalidad el compilador con las optimizaciones, puede existir algún caso raro en que las optimizaciones no funcionen como deberían.
- Si se compila código con el flag de optimizaciones, se realizan todas las optimizaciones, si no, no se realiza ninguna. Se podría mejorar añadiendo argumentos a el flag para poder decidir cuáles optimizaciones realizar y cuáles no.

- Si se compila código con optimizaciones y además se imprime información de debug, puede pasar que en algunos casos en la impresión del AST (cuando se optimizó código muerto) se impriman bloques (`BLOCK`) fantasmas (que fueron optimizados).