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

Taller de Diseño de Software: Proyecto TDS25

- Asignatura: Taller de Diseño de Software (Cod. 3306)
- Alumnos: Irigoyen Juan Cruz, Gutierrez Marysol
- Año: 2025

1. Introducción

El presente documento describe el desarrollo de un compilador para el lenguaje de programación TDS25, realizado en la asignatura Taller de Diseño de Software (Cod. 3306) de la Universidad de Río Cuarto.

El desarrollo del proyecto fue de manera incremental, teniendo una división por etapas para construir de forma progresiva el compilador: Análizador sintáctico y léxico (Scanner y Parser), análizador semántico, generador de código intermedio, generador de código objeto y optimizador.

El objetivo del proyecto fue diseñar e implementar cada una de estas etapas correctamente, garantizando que el compilador sea capaz de procesar programas escritos en el lenguaje de programación TDS25, generando como resultado final el código ejecutable.

A continuación, se ofrecerá información sobre la división del trabajo entre los integrantes del equipo, la lista de decisiones de diseño, la descripción del diseño y sus decisiones clave, detalles de implementación interesantes, deficiencias de diseño, problemas conocidos en el proyecto para cada etapa de desarrollo.

2. División de Trabajo

La división del trabajo entre los integrantes fue decidida de la siguiente manera teniendo en cuenta cómo fue dividido el Preproyecto de la asignatura, para que cada integrante pueda comprender el diseño, implementación y funcionamiento de cada etapa.

Se tendrá en cuenta lo desarrollado durante la etapa que se menciona, los cambios que se hicieron posteriormente en alguna implementación de las etapas por razones de modularización, funcionamiento entre etapas o solución de problemas no se tomarán en cuenta en este apartado.

2.1. Análizador Sintáctico y Léxico

- Irigoyen Juan Cruz: Análizador sintáctico completo (En preproyecto desarrolló el análizador léxico completo).
- Gutierrez Marysol: Análizador léxico completo (En preproyecto desarrolló el análizador sintáctico completo).

2.2. Análizador Semántico

- Irigoyen Juan Cruz: Tabla de símbolos completa (En preproyecto desarrolló el árbol sintáctico abstracto completo).
- Gutierrez Marysol: Árbol sintáctico abstracto completo (En preproyecto desarrolló la tabla de símbolos).
- El desarrollo del análizador semántico fue dividido aproximadamente en un 50% para cada integrante, donde se dividieron las reglas 1 a 7 y 8 a 13 de las reglas semánticas (Estas reglas fueron definidas por el equipo docente en la especificación del lenguaje TDS25) para que implemente cada uno.

2.3. Generador Código Intermedio

- El desarrollo del generador de código intermedio fue dividido aproximadamente en un 50% para cada integrante teniendo en cuenta las funciones que fueron decididas para el diseño de la etapa.

2.4. Generador Código Objeto

- El desarrollo del generador de código objeto fue dividido aproximadamente en un 50% para cada integrante teniendo en cuenta las funciones que fueron decididas para el diseño de la etapa.

2.5. Optimizador

- La etapa de optimizaciones fue desarrollada a la par con la etapa del generador de código objeto, por inconvenientes de tiempo durante el desarrollo del proyecto. Cada integrante fue implementando optimizaciones para obtener un código objeto que cumpla con la correctitud y completitud esperada.

3. Decisiones de diseño

Las decisiones de diseño para el compilador fueron decididas en base a la especificación del lenguaje de programación TDS25, provista por el equipo docente. Acompañada de conocimiento obtenido durante el desarrollo del preproyecto, ya que él mismo ofreció un acercamiento a las técnicas, herramientas y la estructura que se utilizaría para la creación del compilador a futuro.

3.1. Análizador Sintáctico y Léxico

- Se utilizaron Flex y Bison para la generación del análizador léxico y sintáctico correspondientemente. Lo que proporcionó una forma clara de lectura y creación durante la etapa actual y futuras, ya que los mismos evolucionaron a medida que se avanzaba en el desarrollo del proyecto.
- Para facilitar la detección de palabras reservadas, literales, identificadores, operaciones (Aritméticas, lógicos y comparativos), delimitadores y funcionalidades simples del lenguaje de programación TDS25 (Por ejemplo comentarios de línea o bloque), se utilizaron tokens para agrupar estas palabras y carácteres.
- Los tokens ayudarían en el análizador sintáctico para brindar una estructuración correcta, legible y concisa de las “frases” que se manejaron para detectar errores de sintaxis del lenguaje TDS25.
- Se tomó en cuenta la gramática proporcionada en la especificación del lenguaje TDS25 para la implementación del análizador sintáctico y la estructuración que tiene el mismo.

3.2. Análizador Semántico

- Para la creación de la estructura del árbol sintáctico abstracto (AST) se tuvo en cuenta decisiones que se decidió: Creación de enumeraciones para los tipos de nodos del AST, tipos de operación y tipo de dato.
- Una estructura llamada “AstValor” donde se guardaría toda la información de valores, tipos, flags y la línea donde aparece.
- Todo esto encapsulado en la estructura del nodo, donde trabajaría como un árbol binario y ternario (En otro apartado entraremos más en detalle).
- La traducción que se genera del árbol sintáctico abstracto es guiada por las reglas gramaticales, es decir mediante el archivo .y del análizador sintáctico.
- Por otro lado, para la tabla de símbolos se decidió utilizar una pila dinámica para representar los diferentes niveles (bloques anidados) con las variables y funciones alcanzadas.

- Teniendo en cuenta estas estructuras, el análizador semántico se mueve utilizando el AST y usando la información de la tabla de símbolos.
- Los chequeos que se decidieron que el compilador proporciona son: Declaración única de identificadores. detectar variables no declaradas, chequeo de tipos básicos y valores (En declaraciones, expresiones, operaciones y asignaciones), chequeo de inicialización de variables, chequeo en condiciones de estructuras de control de flujo, gestión de scopes, chequeo de llamadas a funciones y parámetros (Formales y globales), chequeo de retornos y existencia de la función main.

3.3. Generador Código Intermedio

- Para la generación del código intermedio se siguió un formato de código de tres direcciones, teniendo en cuenta la generación del código objeto a futuro.
- Cómo estrategía para las diferentes instrucciones, se creó una enumeración de los diferentes tipos de instrucciones y se formaría la estructura de instrucción.
- Estas estructuras mencionadas, se complementan a la estructura principal para manejar el código intermedio.
- Además que se aprovecharía de la información brindada por el AST y la tabla de símbolos generada.
- Las funciones decididas para generar el código intermedio fueron pensadas para generar de forma separada los diferentes tipos de declaraciones, sentencias, operaciones y expresiones.

3.4. Generador Código Objeto

- Para el generador del código objeto (assembly x86-64) decidimos que se devolvería un archivo .txt como resultado para la primera instancia de prueba y luego esto se modificaría para que genere el archivo ejecutable.
- Para mejor manipulación de los diferentes registros, se pensaron en funciones para mapear los mismos y otras para cargar, guardar o mover información en los registros para proporcionar mejor claridad y legibilidad en el código.
- Además de utilizar una función auxiliar para gestionar las diferentes traducciones de todas las operaciones que soporta el compilador.
- Primero decidimos usar el formato Microsoft para generar el código objeto, aunque luego lo cambiamos al formato GNU para obtener el código ejecutable. Este último formato fue recomendado por parte de la información dada por el equipo docente.

3.5. Optimizador

- Cómo la etapa de optimización se realizó a la par con la del generador de código objeto. La prioridad fue solucionar errores en casos de testing, por ejemplo la traducción correcta de llamadas a funciones con sus parámetros, ciclos anidados estructurados correctamente, etc. Además de algunas mejoras de diseño de legibilidad, reusabilidad, eficiencia y mantenibilidad en el código.
- También la tabla de símbolos al principio la creamos para que gestionará dentro del análizador sintáctico, luego por temas de consistencia y completitud decidimos para la devolución de errores del análizador semántico al archivo .sem. Dejando así que nuestro compilador primero crea el AST y luego la tabla de símbolos.

4. Descripción del diseño y decisiones claves

Las decisiones de diseño para el compilador fueron decididas en base a la especificación del lenguaje de programación TDS25, provista por el equipo docente. Acompañada de conocimiento obtenido durante el desarrollo del preproyecto, ya que él mismo ofreció un acercamiento a las técnicas, herramientas y la estructura que se utilizaría para la creación del compilador a futuro.

4.1. Análizador Sintáctico y Léxico

- Dentro del análizador léxico, utilizamos los tokens cómo explicamos anteriormente para estructurar de manera limpia, clara y concisa la grámatica del análizador sintáctico. Además, utilizamos yyval (Una variable global para pasar el valor semántico de un token desde el análizador léxico al sintáctico) para que cada token tenga información del tipo, valor, línea o operador.
- Se generó un soporte completo para línea de comentarios, bloque de comentarios y hasta la posibilidad de anidar estos últimos.
- Por parte del análizador sintáctico, se implementó la construcción directa del AST dentro del mismo. Utilizando las funciones de creación de nodos, ya sean binarios, ternarios o hojas (Estás creadas en el archivo ast.c).
- Además de tener un reporte de errores por línea, ofreciendo información sobre el tipo de error.

4.2. Análizador Semántico

- Entre las decisiones claves de esta etapa, consideramos la construcción de la estructura de la tabla de símbolos usando una pila de niveles, teniendo en cuenta las funcionalidades que debe soportar para TDS25 y simplicidad de implementación (Otra alternativa podría ser usar un hashMap). Utilizando para la generación de la misma una tabla con niveles activos y otra con que guarda el historial de los que ya se recorrieron. Con esto se conserva la información completa de los distintos niveles de la tabla para su impresión.
- Para denotar lo que es un símbolo en la tabla, utilizamos la estructura AstValor para copiar la información de cada uno de los campos necesarios (tipo, nombre, flags, etc), aprovechando la reusabilidad de nuestras implementaciones.
- Por parte del árbol semántico abstracto, consideramos hacer su construcción e impresión simple para simplificar el análisis y generar la implementación del código de manera eficiente.
- La misma contiene dentro de la estructura AstValor toda la información necesaria (tipo, nombre, valor, línea) en conjunto con flags para detectar si la

variables fue declarada, si es una función, si es un parámetro y el atributo offset. Brindando una trazabilidad y vínculo entre las diferentes etapas.
- Por parte del análizador semántico en su completitud, el mismo hace los chequeos de manera estructurada dependiendo el tipo del nodo que se analice del AST. Estos cubren muchos aspectos del lenguaje TDS25 cómo: duplicación de variable o funciones, uso de variable antes de declarar, consistencia de tipos en operaciones, inicializaciones, condiciones booleanas y retornos, la existencia única de la función main (sin parámetros) y llamadas a funciones (chequeando también el tipo de los parámetros). Además de gestionar los niveles de anidamiento de los bloques, asegurando una construcción segura para la tabla de símbolos.
- Entre las mejoras que se pueden hacer en el compilador es el soporte de las reglas de conversión (Casting) y polimorfismo, pero teniendo en cuenta la duración del proyecto y las especificaciones dadas del lenguaje TDS25, no es necesaria su implementación.

4.3. Generador Código Intermedio

- Para la generación del código intermedio consideramos representarlo en tres direcciones con la estructura de instrucción la cual contiene el tipo, los argumentos, resultado, label y un puntero a la siguiente instrucción. Estos atributos tienen valores asociados dependiendo del tipo de instrucción. Y luego la estructura principal de código intermedio que cuenta con un puntero a la primera instrucción y a la última generada, un contador para temporales y otro para labels.
- Una implementación modularizada donde generamos la traducción de manera separada para sentencias, expresiones y funciones. Esta última para generar de manera correcta las funciones con sus parámetros en la impresión.
- Teniendo un manejo automático de temporales y offsets mediante funciones auxiliares, incorporación de labels únicos y una conversión de identificadores a referencias de destino teniendo en cuenta si es global, local o parámetro.

4.4. Generador Código Objeto

- El generador del código objeto se genera recorriendo la lista de instrucciones generada en la etapa anterior, ofreciendo una traducción literal.
- Consideramos trabajar con los registros de la siguiente manera: Cargar argumentos de parámetros de funciones (rdi, rsi, rdx, rcx, r8, r9), valor de retorno en funciones (rax), cálculo de operaciones (rax, rbx, r10, r11) y gestión de variables locales y temporales en el stack (rbp, rsp).

- Se crearon auxiliares para cargar y guardar valores de expresiones y mover constantes, memoria o identificadores a un registro determinado para reusabilidad, claridad, calidad y mantenibilidad del código implementado.
- El código objeto generado es devuelto en un archivo .txt para un análisis eficiente y pasar el ensamblado para obtener el ejecutable del mismo.

4.5. Optimizador

- La decisión principal es la modularización de funciones auxiliares para reducir la duplicación del código en todas las etapas.
- Decidimos hacer optimizaciones en el código intermedio en vez del código objeto para seguridad y portabilidad.
- Para mayor claridad en la funcionalidad del compilador y en el código, se tomó la decisión de dejar todo lo que es la construcción del AST dentro del análizador sintáctico (Archivo .y) y pasar lo implementado de la construcción de la tabla de símbolos al análizador semántico. Logrando que se puedan retornar de manera correcta errores semánticos por parte de la interfaz de comandos (La cual puede devolver archivos de la etapa concreta que quiera el usuario o directamente el archivo ejecutable) y tener una generación correcta de las estructuras importantes con las que trabaja el compilador.
- Entre los defectos que encontramos y que se pueden mejorar es la liberación de temporales utilizando una lista la cual guarde los que están en uso y se liberen cuando terminan de estar ocupados.
- Reescritura de impresión del código objeto de formato Microsoft al formato GNU.

5. Detalles de implementación

- Utilizar la técnica de descomposición y modularización de funciones para reutilizar, reducir duplicidad y brindar una lectura clara del código durante todas las etapas.
- Implementación de estructura nodo con punteros a tres hijos, permitiendo representar estructuras binarias y ternarias según el tipo de sentencia.
- Para la gestión de bloques, implementamos la tabla de símbolos basándonos en una pila dinámica de niveles. Cada nivel contiene una lista enlazada de símbolos del bloque y un puntero al anterior. Permitiendo abrir y cerrar los mismos usando funciones simples. Facilitando otras funcionalidades cómo: Buscar el identificador si no se encuentra en el nivel actual, distinción entre variables globales, locales y parámetros y facilitar la generación de offsets para variables locales.
- La estructura del AST nos permitió tener una comunicación eficiente y rápida con las etapas posteriores por la estructura AstValor. Permitiendo que no se necesitará de estructuras similares o duplicación de la misma.
- Para las estructuras del AST y código intermedio, utilizamos enumeraciones para diferenciar los tipos de nodos y tipos de instrucciones. Facilitando la implementación de la impresión en cada una de estas etapas.
- El diseño planteado de código intermedio permite que se generé incrementalmente a medida que se recorre el AST mediante las funciones de generación mencionadas anteriormente, garantizando una traducción directa de la estructura semántica a la representación intermedia del programa.
- En la generación de código intermedio y objeto, las variables locales son asignadas automáticamente en el stack por medio del registro rbp, calculando los offsets a medida que se detectan variables locales o parámetros mientras se gestiona la memoria. Evitando calcular posiciones de memoria manualmente.
- Para el generador de código objeto se crearon funciones auxiliares para mejorar la legibilidad y reducir duplicación para acciones cómo cargar, guardar, mover y traducir.
- Los errores semánticos y sintácticos son mostrados en mensajes ofreciendo precisión de la línea, el tipo de error y la razón del mismo. Mejorando la experiencia para el usuario y la depuración durante el desarrollo.
- Los labels son generados de manera automática para estructuras de control.

6. Lista de problemas

- Dependencia de funciones externas: Estas funciones no tienen implementación dentro del lenguaje, por lo que deben ser provistas por el usuario al enlazar el código objeto generado. Sin estas implementadas, el linkeo finaliza con errores. Esto se detectó por medio de los tests que utilizan este tipo de funciones. Se agregó un archivo llamado “funcionesTests.c” para soportar estas funciones, el usuario debe agregar dentro del mismo las implementaciones necesarias para generar el ejecutable y se modificó la etapa OUT para incluir este archivo durante la ejecución.
- El código objeto generado es correcto, pero con falta de optimizaciones. Ya que se usan siempre rax y rbx, las variables temporales van siempre a memoria, no hay propagación de constantes o eliminación de código muerto. Esto mismo se vio reflejado en la impresión del código objeto y chequeando la ejecución de instrucciones.
- La interfaz de comandos no soporta los comandos de optimizaciones e información de debugging por cuestiones de tiempo durante el desarrollo del proyecto.
- El compilador ante un error muestra un mensaje, el mismo faltan detalles cómo tokens que se esperan y un análisis posterior del mismo. Esto se detectó haciendo pruebas de error. Para mejorar esto se debería ajustar el comportamiento de los errores devueltos por parte de los diferentes analizadores que contiene el compilador. Además de que los mismos pueden contener nuevos casos que no son cubiertos en la versión actual.
- Para la inicialización de variables se debe pasar por asignación un valor para las mismas, ya que el compilador no soporta valores determinados para cada tipo (Integer, Boolean). Por ello, se debe asignar siempre un valor a la variable que se inicialice.