Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Algoritmos y Estructuras de Datos
Curso K2055
Ing. Pablo D. Mendez

Trabajo práctico final

Grupo SinGrupo

Fecha de entrega: 28/11/2025

Legajo	Nombre y Apellido	Correo Institucional	Github
160.664-5	Alain Degoumois	adegoumoispasserini@frba.utn.edu.ar	https://github.com/adego25
215.087-6	Carracedo Ignacio Lautaro	isantillan@frba.utn.edu.ar	https://github.com/nachito-cc
203.665-4	Mauro Ghia Feroldi	mghiaferoldi@frba.utn.edu.ar	https://github.com/MauroGhia1

Repo del grupo:
https://github.com/adego25/TP_SySL_Grupo_Olvida2

Informe del Proyecto

1. Descripción de la Solución

Para este TP Final desarrollamos un intérprete para el lenguaje **MICRO**. El programa permite declarar variables, realizar cuentas matemáticas, manejar textos y mostrar resultados por pantalla, cumpliendo con la gramática que nos pasó la cátedra.

Para hacerlo, usamos las herramientas que vimos en la cursada trabajando sobre C:

- **Flex (Scanner):** Lo usamos para la parte léxica. Definimos los tokens usando expresiones regulares. También configuramos que ignore los comentarios con `//` y los espacios en blanco.
- **Bison (Parser):** Acá definimos la gramática del lenguaje. A medida que el parser va reconociendo las reglas, ejecutamos código en C para hacer las sumas, restas y asignaciones en el momento.
- **Tabla de Símbolos:** Implementamos una tabla de símbolos simple en C para guardar las variables. Ahí almacenamos el nombre, el tipo de dato y el valor actual de cada variable mientras corre el programa.

2. Hipótesis y Decisiones de Diseño

Para que el intérprete funcione bien y no explote, tomamos estas decisiones:

1. **Errores:** Si el programa encuentra un error semántico,decidimos cortar la ejecución y avisar el error al usuario.
2. **Strings:** Como pedía la consigna, limitamos los strings a 255 caracteres. Usamos memoria dinámica (`malloc`/`strdup`) para guardarlos en la tabla de símbolos.
3. **Constantes:** Implementamos la sentencia `const`. El valor se asigna sí o sí cuando la declarás. Si después querés cambiarle el valor a una constante, el intérprete te tira error.
4. **Ejecución:** Hicimos que el `main` te deje elegir: podés escribir código en el momento o pasarle el nombre de un archivo `.micro` para que lo lea y ejecute.

3. División de Tareas

Nos organizamos así para llegar con todo:

- **Ignacio Santillan Carracedo y Alain Degoumois:** se encargaron de armar el entorno en Windows, hacer la parte de Flex, la gramática de Bison, programar las funciones de la Tabla de Símbolos en C y el armado de los casos de prueba.
- **Mauro Ghia Feroldi:** Se ocupó de la parte teórica del trabajo, de investigar y redactar las respuestas sobre las fases de compilación, el funcionamiento de los Makefiles y el concepto de espacios de nombres para completar el informe.

Parte Teórica

1. Enumere las fases de compilación de un programa C. Identifique en qué fases interactúa la tabla de símbolos.

La compilación en C pasa por 4 etapas:

1. **Preprocesamiento:** Se resuelven los `#include`, `#define` y se limpian los comentarios.
2. **Compilación:** Pasa el código al ensamblador. Acá es donde se hace el análisis léxico, sintáctico y semántico. Es necesaria la Tabla de Símbolos, porque el compilador necesita chequear tipos y ver si las variables existen.
3. **Ensamblado:** Convierte el ensamblador a código objeto (lenguaje máquina, los famosos.
4. **Enlazado (Linking):** Junta todos los `.o` y las librerías para armar el ejecutable final.

Tabla de Símbolos: Interactúa principalmente en la etapa de Compilación y también sirve para generar el código, ya que ayuda a calcular las direcciones de memoria.

2. Describa el concepto de "espacio de nombres" en C y dé ejemplos. ¿Qué rol cumple la tabla de símbolos con este concepto?

En C no existe la palabra `namespace` como en C++, pero el concepto existe igual mediante el scope y las categorías de los identificadores. Básicamente, un espacio de nombres es un contexto donde un nombre es único.

Ejemplos:

- Podés tener una `struct Nodo` y una variable `int Nodo`. No chocan porque las etiquetas de las structs van por un carril distinto al de las variables.
- Dos funciones distintas pueden tener variables locales con el mismo nombre y no pasa nada.

Rol de la Tabla de Símbolos: La tabla se encarga de diferenciar esto. A cada símbolo que guarda le agrega atributos. Así, el compilador sabe distinguir si estás hablando de la struct o de la variable, aunque se llamen igual.

3. Investigue y describa el makefile de C. Enumere sus parámetros y dé un ejemplo de uso utilizando un programa con bibliotecas.

El `makefile` es un script que usa la herramienta `make` para automatizar la compilación. Sirve para no tener que escribir el comando `gcc` eterno cada vez que cambiamos una línea de código. Se fija qué archivos cambiaste y recompila solo lo necesario.

Parámetros clave:

- **Target:** Lo que querés generar (el `.exe`).
- **Dependencies:** Qué archivos necesitás para generar eso (los `.o` o `.c`).
- **Commands:** El comando de terminal para compilar (ej: `gcc ...`).

Ejemplo simple:

Makefile

```
# Variables para no repetir
CC = gcc
CFLAGS = -Wall
LIBS = -lm  # Acá linkeamos la librería matemática (math.h)

# El objetivo final
programa: main.o calculos.o
    $(CC) -o programa main.o calculos.o $(LIBS)

# Cómo generar los objetos
main.o: main.c
    $(CC) -c main.c $(CFLAGS)

calculos.o: calculos.c
    $(CC) -c calculos.c $(CFLAGS)

clean:
    rm *.o programa
```

4. Describa los pasos que realiza make para llegar al programa ejecutable. Identifique en qué paso se confeccionan los programas objeto y explique qué función cumple el linker.

Pasos de Make:

1. Lee el `Makefile`.
2. Compara la fecha del archivo destino contra sus dependencias.
3. Si alguna dependencia es más nueva que el destino, ejecuta el comando de compilación.
4. Hace esto en cadena hasta llegar al ejecutable final.

Programas Objeto: Los archivos objeto (`.o`) se crean en la etapa de Compilación/Ensamblado, antes de juntar todo.

Función del Linker: El Linker es el que trabaja al final de todo. Agarra todos los archivos objeto (`.o`) que fuiste compilando por separado y las librerías del sistema, resuelve las referencias cruzadas y te entrega el archivo ejecutable listo para correr.