Trabajos de Sintaxis y Semántica de los Lenguajes

Esp. Ing. José María Sola, profesor.

Revisión 3.23.0-rc.1
2021-08-24

Tabla de contenidos

Lista de ejemplos

1

Introducción

El objetivo de los trabajos es afianzar los conocimientos y evaluar su comprensión.

En la sección "Trabajos" de la página del curso[1] se indican cuales de los trabajos acá definidos que son **obligatorios** y cuales **opcionales**, como así también si se deben resolver **individualmente** o en **equipo**.

En el sección "Calendario" de la página del curso[2] se establece cuando es la **fecha y hora límite de entrega**,

Hay trabajos opcionales que son introducción a otros trabajos más complejos, también pueden enviar la resolución para que sea evaluada.

Cada trabajo tiene un **número** y un **nombre**, y su enunciado tiene las siguientes secciones:

1. **Objetivos**: Descripción general de los objetivos y requisitos del trabajo.
2. **Temas**: Temas que aborda el trabajo.
3. **Problema**: *Descripción* del problema a resolver, la *definición completa y sin ambigüedades* es parte del trabajo.
4. **Tareas**: Plan de tareas a realizar.
5. **Restricciones**: Restricciones que deben cumplirse.
6. **Productos**: Productos que se deben entregar para la resolución del trabajo.

[1] https://josemariasola.wordpress.com/ssl/assignments/
[2] https://josemariasola.wordpress.com/ssl/calendar/

2

Requisitos Generales para las Entregas de las Resoluciones

Cada trabajo tiene sus requisitos particulares de entrega de resoluciones, esta sección indica los requisitos generales, mientras que, cada trabajo define sus requisitos particulares.

Una resolución se considera **entregada** cuando cumple con los **requisitos de tiempo y forma** generales, acá descriptos, sumados a los particulares definidos en el enunciado de cada trabajo.

La entrega de cada resolución debe realizarse a través de *GitHub*, por eso, cada estudiante tiene poseer una cuenta en esta plataforma.

2.1. Requisitos de Forma

2.1.1. Repositorios

En el curso usamos repositorios *GitHub*. Uno público y personal y otro privado para del equipo.

Repositorios público y privado.

```
Usuario
`-- Repositorio público personal para la asignatura
Repositorio privado del equipo
```

Repositorio Personal para Trabajos Individuales

Cada estudiante debe crear un repositorio público dónde publicar las resoluciones de los trabajos individuales. El nombre del repositorio debe ser el de la asignatura. En la raíz del mismo debe publicarse un archivo `readme.md` que actúe como *front page* de la persona. El mismo debe estar escrito en notación *Markdown* y debe contener, como mínimo, la siguiente información:

- Sintaxis y Semántica de los Lenguajes
- Curso.
- Año de cursada, y cuatrimestre si corresponde.
- Legajo.
- Apellido.
- Nombre.

Repositorio personal para la asignatura.

```
Usuario
`-- Repositorio público personal para la asignatura
    `-- readme.md // Front page del usuario
```

Repositorio de Equipo para Trabajos Grupales

A cada equipo se le asigna un **repositorio privado**. En la raíz del mismo debe publicarse un archivo `readme.md` que actúe como *front page* del equipo. El mismo debe estar escrito en notación *Markdown* y debe contener, como mínimo, la siguiente información:

- Sintaxis y Semántica de los Lenguajes
- Curso.
- Año de cursada, y cuatrimestre si corresponde.
- Número de equipo.
- Nombre del equipo (opcional).
- Integrantes del equipo actualizados, ya que, durante el transcurso de la cursada el equipo puede cambiar:

- Usuario *GitHub*.
- Legajo.
- Apellido.
- Nombre.

Repositorio privado del equipo.

```
Repositorio privado del equipo
`-- readme.md // Front page del equipo.
```

Carpetas para cada Resolución

La resolución de cada trabajo debe tener su propia carpeta, ya sea en el repositorio personal, si es un trabajo individual, o en el del equipo, si es un trabajo grupal. El nombre de la carpeta debe seguir el siguiente formato:

```
DosDígitosNúmeroTrabajo-NombreTrabajo
```

O en notación *regex*:

```
[0-9]{2}"-"[a-zA-Z]+
```

Ejemplo 2.1. Nombre de carpeta

00-Hello

En los enunciados de cada trabajo, el número de trabajo para utilizar en el nombre de la carpeta está generalizado con "DD", se debe reemplazar por los dos dígitos del trabajo establecidos en el curso.

Adicionalmente a los productos solicitados para la resolución de cada trabajo, la carpeta debe incluir su propio archivo `readme.md` que actúe como *front page* de la resolución El mismo debe estar escrito en notación *Markdown* y debe contener, como mínimo, la siguiente información:

- Número de equipo.
- Nombre del equipo (opcional).
- Autores de la resolución:
  - Usuario github.
  - Legajo.
  - Apellido.
  - Nombre.
- Número y título del trabajo.
- Transcripción del enunciado.
- Hipótesis de trabajo que surgen luego de leer el enunciado.

Opcionalmente, para facilitar el desarrollo se **recomienda incluir**:

- un archivo `.gitignore`.
- un archivo `Makefile`.[1]
- archivos tests.[1]

Carpeta de resolución de trabajo.

```
Carpeta de resolución de trabajo
|-- .gitignore
|-- Makefile
|-- readme.md // Front page de la resolución
`-- Archivos de resolución
```

Por último, la carpeta **no debe incluir**:

- archivos ejecutables.
- archivos intermedios producto del proceso de compilación o similar.

Ejemplo de Estructura de Repositorios

Ejemplo completo.

[1] Para algunos trabajos, el archivo `Makefile` y los tests son obligatorios, de ser así, se indica en el enunciado del trabajo.

```
usuario // Usuario GithHub
`-- Asignatura // Repositorio personal público para a la asignatura
    |-- readme.md // Front page del usuario
    |-- 00-Hello // Carperta de resolución de trabajo
    |    |-- .gitignore
    |    |-- readme.md // Front page de la resolución
    |    |-- Makefile
    |    |-- hello.cpp
    |    `-- output.txt
    `-- 01-Otro-trabajo
2019-051-02 // Repositorio privado del equipo
|-- redme.md // Front page del equipo
|-- 04-Stack // Carperta de resolución de trabajo
|    |-- .gitignore
|    |-- readme.md // Front page de la resolución
|    |-- Makefile
|    |-- StackTest.cpp
|    |-- Stack.h
|    |-- Stack.cpp
|    `-- StackApp.cpp
`-- 01-Otro-trabajo
```

2.1.2. Lenguaje de Programación

En el curso se establece la versión del estándar del lenguaje de programación que debe utilizarse en la resolución.

2.1.3. Header Comments (Comentarios Encabezado)

Todo archivo fuente debe comenzar con un comentario que indique el "Qué", "Quiénes", "Cuándo" :

```
/* Qué: Nombre
 * Breve descripción
 * Quiénes: Autores
 * Cuando: Fecha de última modificación
 */
```

Ejemplo 2.2. Header comments

```
/* Stack.h
 * Interface for a stack of ints
 * JMS
 * 20150920
 */
```

2.2. Requisitos de Tiempo

Cada trabajo tiene una **fecha y hora límite de entrega**, los *commits* realizados luego de ese instante no son tomados en cuenta para la evaluación de la resolución del trabajo.

En el calendario del curso[2] se publican cuando es la fecha y hora límite de entrega de cada trabajo.

[2] https://josemariasola.wordpress.com/ssl/calendar/

3

"Hello, World!" en C

3.1. Objetivos

- Demostrar con, un programa simple, que se está en capacidad de editar, compilar, y ejecutar un programa C.
- Contar con las herramientas necesarias para abordar la resolución de los trabajos posteriores.

3.2. Temas

- Sistema de control de versiones.
- Lenguaje de programación C.
- Proceso de compilación.
- Pruebas.

3.3. Problema

Adquirir y preparar los recursos necesarias para resolver los trabajos del curso.

3.4. Restricciones

- Ninguna.

3.5. Tareas

1. Si no posee una cuenta *GitHub*, crearla.
2. Crear un repositorio público llamado `ssl`.

3. Escribir el archivo `readme.md` que actúa como *front page* del repositorio personal.
4. Crear la carpeta `00-CHelloWorld`.
5. Escribir el archivo `readme.md` que actúa como *front page* de la resolución.
6. Seleccionar, instalar, y configurar un compilador **C11** ó **C18**.
7. Indicar en `readme.md` el compilador seleccionado.
8. Probar el compilador con un programa `hello.c` que envíe a `stdout` la línea `Hello, World!` o similar.
9. Ejecutar el programa, y capturar su salida en un archivo `output.txt` .
10. Publicar en el repositorio personal `SSL` la carpeta `00-CHelloWorld` con `readme.md`, `hello.c`, y `output.txt` .
11. La última tarea es informar el usuario *GitHub* en la lista indicada en el curso.

3.6. Productos

```
Usuario
`-- SSL
    `-- 00-CHelloWorld
        |-- readme.md
        |-- hello.c
        `-- output.txt
```

3.7. Referencia

- [Git101]
- [CompiladoresInstalacion]
- ??? § 1.1 Comenzado

4

Uso del Lenguaje C en mi Día a Día

4.1. Objetivos

- Identificar tecnologías basadas en el Lenguaje C y que usamos en nuestro día a día para estimar el nivel de adopción de C.

4.2. Temas

- Lenguaje C.

4.3. Tareas

1. Listar entre tres y diez tecnologías digitales que usamos en nuestro día a día.
2. Indicar para cada tecnología el repositorio público donde se la desarrolla, si es que lo tiene.
3. Indicar para cada una de esas tecnologías si se desarrollan en C o no.

5

Niveles del Lenguaje: Hello.cpp v Hello.c

5.1. Objetivos

- Analizar e identificar las diferencias entre `hello.cpp` y `hello.c`, en los tres niveles: léxico, sintáctico, y semántico.

5.2. Temas

- Lenguaje C++.
- Lenguaje C.
- Niveles del Lenguaje.
- Léxico.
- Sintaxis.
- Semántica.

5.3. Tareas

1. Armar una tabla con similitudes y diferencias para cada uno de los tres niveles del lenguaje, que compare ambas versiones de `hello`.
2. Opcional: Agregar una tercera versión en otro lenguaje de programación.

5.4. Restricciones

- Ninguna.

5.5. Productos

```
DD-HelloCppVHelloC
`-- HelloCppVHelloC.md
```

6

Interfaces & Makefile — Temperaturas

Este trabajo está basado en los ejercicios 1-4 y 1-15 de [KR1988] y aplica los conceptos presentados en [Interfaces-Make]:

> 1-4. Escriba un programa para imprimir la tabla correspondiente de Celsius a Fahrenheit

> 1-15. Reescriba el programa de conversión de temperatura de la sección 1.2 para que use una función de conversión.

Desarrollar un programa que imprima dos tablas de conversión, una de Fahrenheit a Celsius y otra de Celsius a Fahrenheit.

6.1. Objetivos

- Aplicar el uso de interfaces y módulos.
- Construir un programa formado por más de una unidad de traducción.
- Comprender el proceso de traducción o *Build* cuando intervienen varios archivos fuente.
- Aplicar el uso de `Makefile`.

6.2. Temas

- `Makefile`.
- Archivos header (.h).
- Tipo de dato `double`.
- Funciones.
- Pruebas unitarias.
- `assert`.

> La comparación de los tipos flotantes puede ser no trivial debido a su representación y precisión.

- Interfaces e Implementación.

6.3. Tareas

1. Escribir el `Makefile`.
2. Escribir `Conversion.h`
3. Escribir `ConversionTest.c`
4. Escribir `Conversion.c`
5. Escribir `TablasDeConversion.c`.

6.4. Restricciones

- Las dos funciones públicas deben llamarse `Celsius` y `Farenheit`.
- Utilizar `assert`.
- Utilizar `const` y no `define`.
- Utilizar `for` con declaración (C99).

6.5. Productos

```
DD-Interfaces
|-- readme.md
|-- Makefile
|-- Conversion.h
```

```
|-- ConversionTest.c
|-- Conversion.c
`-- TablasDeConversion.c.
```

Crédito extra

Desarrolle `TablasDeConversion.c` para que use funciones del estilo `PrintTablas`, `PrintTablaCelsius`, `PrintTablaFahrenheit`, `PrintFilas`, `PrintFila`.

Los límites inferior y superior, y el incremento deben ser parámetros.

Crédito extra

Desarrollar la función `PrintFilas` para que sea genérica, es decir, pueda invocarse desde `PrintTablaFahrenheit` y desde `PrintTablaCelsius`. `PrintFilas` debe invocar a `PrintFila`.

Considere el uso de punteros a función.

7

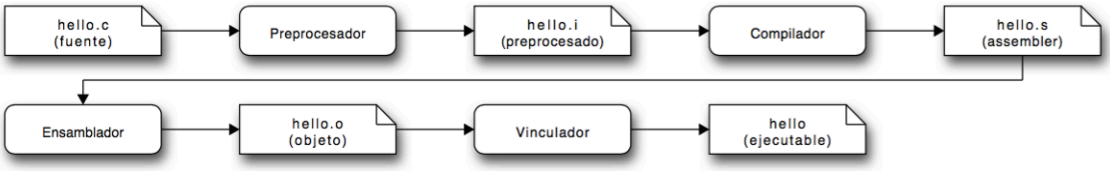
Fases de la Traducción y Errores

7.1. Objetivos

Este trabajo tiene como objetivo identificar las fases del proceso de traducción o *Build* y los posibles errores asociados a cada fase.

Para lograr esa identificación se ejecutan las fases de traducción una a una, se detectan y corrigen errores, y se registran las conclusiones en `readme.md`.

No es un trabajo de desarrollo; es más, el programa que usamos como ejemplo es simple, similar a `hello.c` pero con errores que se deben corregir. La complejidad está en la identificación y comprensión de las etapas y sus productos.



7.2. Temas

- Fases de traducción.
- Preprocesamiento.
- Compilación.
- Ensamblado.
- Vinculación (Link).
- Errores en cada fase.

- Compilación separada.

7.3. Tareas

1. La primera tarea es investigar las funcionalidades y opciones que su compilador presenta para limitar el inicio y fin de las fases de traducción.
2. La siguiente tarea es poner en uso lo que se encontró. Para eso se debe transcribir al `readme.md` cada **comando ejecutado** y su **resultado** o **error** correspondiente a la siguiente secuencia de pasos. También en `readme.md` se vuelcan las conclusiones y se resuelven los puntos solicitados. Para claridad, mantener en `readme.md` la misma numeración de la secuencia de pasos.

7.3.1. Secuencia de Pasos

Se parte de un archivo fuente que es corregido y refinado en sucesivos pasos. Es importante no saltearse pasos para mantener la correlación, ya que el estado dejado por el paso anterior es necesario para el siguiente.

1. Preprocesador

   a. Escribir `hello2.c`, que es una variante de `hello.c`:

   ```
   #include <stdio.h>

   int/*medio*/main(void){
    int i=42;
    prontf("La respuesta es %d\n");
   ```

   b. Preprocesar `hello2.c`, no compilar, y generar `hello2.i`. Analizar su contenido. ¿Qué conclusiones saca?

   c. Escribir `hello3.c`, una nueva variante:

   ```
   int printf(const char * restrict s, ...);

   int main(void){
    int i=42;
    prontf("La respuesta es %d\n");
   ```

   d. Investigar e indicar la semántica de la primera línea.

e. Preprocesar `hello3.c`, no compilar, y generar `hello3.i`. Buscar diferencias entre `hello3.c` y `hello3.i`.

2. Compilación

   a. Compilar el resultado y generar `hello3.s`, no ensamblar.

   b. Corregir solo los errores, no los *warnings*, en el nuevo archivo `hello4.c` y empezar de nuevo, generar `hello4.s`, no ensamblar.

   c. Leer `hello4.s`, investigar sobre lenguaje ensamblador, e indicar de formar sintética cual es el objetivo de ese código.

   d. Ensamblar `hello4.s` en `hello4.o`, no vincular.

3. Vinculación

   a. Vincular `hello4.o` con la biblioteca estándar y generar el ejecutable.

   b. Corregir en `hello5.c` y generar el ejecutable. Solo corregir lo necesario para que vincule.

   c. Ejecutar y analizar el resultado.

4. Corrección de Bug

   a. Corregir en `hello6.c` y empezar de nuevo; verificar que funciona como se espera.

5. Remoción de prototipo

   a. Escribir `hello7.c`, una nueva variante:

```
int main(void){
    int i=42;
    printf("La respuesta es %d\n", i);
}
```

   b. Explicar porqué funciona.

6. Compilación Separada: Contratos y Módulos

   a. Escribir `studio1.c` (sí, `studio1`, no `stdio`) y `hello8.c`.

La unidad de traducción `studio1.c` tiene una implementación de la función `prontf`, que es solo un wrappwer[1] de la función estándar `printf`:

```
void prontf(const char* s, int i){
 printf("La respuesta es %d\n", i);
}
```

La unidad de traducción `hello8.c`, muy similar a `hello4.c`, invoca a `prontf`, pero no incluye ningún header.

```
int main(void){
 int i=42;
 prontf("La respuesta es %d\n", i);
}
```

b. Investigar como en su entorno de desarrollo puede generar un programa ejecutable que se base en las dos unidades de traducción (i.e., archivos fuente, archivos con extensión `.c`).
   Luego generar ese ejecutable y probarlo.

c. Responder ¿qué ocurre si eliminamos o agregamos argumentos a la invocación de `prontf`? Justifique.

d. Revisitar el punto anterior, esta vez utilizando un contrato de interfaz en un archivo header.

   i. Escribir el contrato en `studio.h`.

```
#ifndef _STUDIO_H_INCULDED_
#define _STUDIO_H_INCULDED_

void prontf(const char*, int);

#endif
```

   ii. Escribir `hello9.c`, un cliente que sí incluye el contrato.

```
#include "studio.h" // Interfaz que importa
```

[1] https://en.wikipedia.org/wiki/Wrapper_function

```
int main(void){
 int i=42;
 prontf("La respuesta es %d\n", i);
}
```

iii. Escribir `studio2.c`, el proveedor que sí incluye el contrato.

```
#include "studio.h" // Interfaz que exporta
#include <stdio.h>  // Interfaz que importa

void prontf(const char* s, int i){
 printf("La respuesta es %d\n", i);
}
```

iv. Responder: ¿Qué ventaja da incluir el contrato en los clientes y en el proveedor.

Crédito extra

> Investigue sobre *bibliotecas*. ¿Qué son? ¿Se puden distribuir? ¿Son portables? ¿Cuáles son sus ventajas y desventajas?.
> Desarrolle y utilice la biblioteca `studio`.

7.4. Restricciones

- El programa ejemplo debe enviar por `stdout` la frase `La respuesta es 42`, el valor 42 debe surgir de una variable.

7.5. Productos

```
DD-FasesErrores
|-- readme.md
|-- hello2.c
|-- hello3.c
|-- hello4.c
|-- hello5.c
|-- hello6.c
|-- hello7.c
|-- hello8.c
|-- studio1.c
|-- studio.h
```

```
`-- studio2.c
```

8

Operaciones de Strings

Este trabajo tiene dos partes, una de análisis comparativo y otra de desarrollo.

El análisis comparativo es sobre el tipo de dato String en el lenguaje de programación C versus otro lenguaje de programación a elección; mientras que el desarrollo está basado en los ejercicios 20 y 21 del Capítulo #1 del Volumen #1 de [MUCH2012], que a continuación transcribe:

> Investigue y construya, en LENGUAJE C, la función que realiza cada operación solicitada:
>
> * Ejercicio 20 *
> (a) Calcula la longitud de una cadena;
> (b) Determina si una cadena dada es vacía.
> (c) Concatena dos cadenas.
>
> * Ejercicio 20 *
> Construya un programa de testeo para cada función del ejercicio anterior.

8.1. Objetivos

1. Parte I — Análisis Comparativo del tipo String en Lenguajes de Programación: Realizar un análisis comparativo de dato String en el lenguaje C versus un lenguaje de programación a elección. El análisis debe contener, por lo menos, los siguientes ítems:

    a. ¿El tipo es parte del lenguaje en algún nivel?

b. ¿El tipo es parte de la biblioteca?

c. ¿Qué alfabeto usa?

d. ¿Cómo se resuelve la alocación de memoria?

e. ¿El tipo tiene mutabilidad o es inmutable?

f. ¿El tipo es un *first class citizen*?

g. ¿Cuál es la mecánica para ese tipo cuando se los pasa como argumentos?

h. ¿Y cuando son retornados por una función?

Las anteriores preguntas son disparadores para realizar una análisis profundo.

2. Parte II — Biblioteca para el Tipo String: Desarrollar una biblioteca con las siguientes operaciones de strings:

   a. *GetLength* ó *GetLongitud*

   b. *IsEmpty* ó *IsVacía*

   c. *Power* ó *Potenciar*

   d. Una operación a definir libremente.

   Notar que en vez de la operación concatenar que propone [MUCH2012] se debe desarrollar *Power* ó *Potenciar* que repite un string n veces.

   La parte pública de la biblioteca se desarrolla en el header `"String.h"`, el cual no debe incluir `<string.h>`. El programa que prueba la biblioteca, por supuesto, incluye a `"String.h"`, pero sí puede incluir `<string.h>` para facilitar las comparaciones.

8.2. Temas

- Strings.
- Alocación.
- Tipos.

8.3. Tareas

1. Parte I

a. Escribir el `AnálisisComparativo.md` con la comparación de strings en C versus otro lenguaje de programación a elección.

2. Parte II

a. Para cada operación, escribir en `Strings.md` la especificación matemática de la operación, con conjuntos de salida y de llegada, y con especificación de la operación.

b. Escribir el `Makefile`.

c. Por cada operación:

i. Escribir las pruebas en `StringsTest.c`.

ii. Escribir los prototipos en `String.h`.

iii. Escribir en `String.h` comentarios con las precondiciones y poscondiciones de cada función, arriba de cada prototipo.

iv. Escribir las implementaciones en `Strings.c`.

8.4. Restricciones

- Las pruebas deben utilizar `assert`.
- Los proptotipos de utilizar `const` cuando corresponde.
- Por lo menos una operación debe implementarse con recursividad.
- Las implementaciones no deben utilizar funciones estándar, declaradas en `<string.h>`

8.5. Productos

```
DD-Strings
|-- readme.md
|-- AnálisisComparativo.md
|-- String.md
|-- Makefile
|-- StringTest.c
|-- String.h
`-- String.c.
```

9

Strings en *Go* (*golang*)

El trabajo consta de la *especificación* del tipo *String* con una selección de operaciones de *String* del lenguaje *Go*, y de le *implementación* del ese tipo con representación en memoria igual a la de ese lenguaje, con la facilidad de alocar strings en el heap, y de liberar esa memoria reservada ante el pedido del *garbage collector*.

Las operaciones del tipo son:

- Len
- Count
- New
- At
- Delete

Recordar que el tipo string es inmutable en Go.

El trabajo incluye también un ejemplo el uso del tipo en un programa que haga uso de las de las operaciones y el desarrollo de una función que reciba un string como parámetro.

9.1. Objetivos

1. Especificación del tipo String que incluya una selección de operaciones de Go.
2. Programa ejemplo en Go.
3. Desarrollo del tipo String de Go en C.

4. Programa ejemplo en C que usa el tipo String de Go.

9.2. Temas

- Strings.
- Alocación.
- Tipos.
- Heap
- Garbage Collector.

9.3. Tareas

a. Especificar el tipo en `GoString.md`.

b. Escribir y ejecutar un programa Go ejemplo de uso con una función que recibe un string en `GoStringExample.go`.

c. Escribir el `Makefile`.

d. Escribir las pruebas en `GoStringTest.c`.

e. Escribir las declaraciones públicas en `GoString.h`.

f. Escribir en `GoString.h` comentarios con las precondiciones, poscondiciones e invariantes.

g. Escribir un ejemplo de uso de tipo, que incluya una funciópn que recibe un string en `GoStringExample.c`.

h. Escribir la implementación en `GoString.c`.

9.4. Restricciones

- Las pruebas deben utilizar `assert`.
- Los proptotipos de utilizar `const` cuando corresponde.
- La operación `at` debe implementar el mismo compartamiento `panic` que tien Go; para eso debe desarrollarse la función `panic` que es invocada por `at` cuando el índice es inválido.

9.5. Productos

```
DD-GoStrings
```

```
|-- readme.md
|-- GoString.md
|-- GoStringExample.go
|-- Makefile
|-- GoStringTest.c
|-- GoString.h
|-- GoStringExample.c
`-- GoString.c
```

10

Máquinas de Estado — Palabras en Líneas

Este trabajo está basado en el ejercicio 1-12 de [KR1988]:

> 1-12. Escriba un programa que imprima su entrada una palabra por línea.

Problema: Imprimir cada palabra de la entrada en su propia línea. La cantidad de líneas en la salida coincide con la cantidad de palabras en la entrada. Cada línea tiene solo una palabra.

10.1. Objetivos

- Aplicar máquinas de estado para el procesamiento de texto.
- Implementar máquinas de estado con diferentes métodos.

10.2. Temas

- Árboles de expresión.
- Representación de máquinas de estado.
- Implementación de máquinas de estado.

10.3. Tareas

1. Árboles de Expresión

a. Estudiar el programa del ejemplo las sección *1.5.4 Conteo de Palabras* de [KR1988].

b. Dibujar el árbol de expresión para la inicialización de los contadores: `nl = nw = nc = 0`.

c. Dibujar el árbol de expresión para la expresión de control del segundo if: `c == ' ' || c == '\n' || c == '\t'`.

2. Máquina de Estado:

   a. Describir en lenguaje `dot` ??? y dentro del archivo `wl.gv` la máquina de estado que resuelve el problema planteado.

   b. Formalizar la máquina de estados como una *n-upla*, basarse en el Capítulo #1 del Volumen #3 de [MUCH2012].

3. Implementaciones de Máquinas de Estado:
   Las implementaciones varían en los conceptos que utilizan para representaar los estados y las transiciones.

   a. Implementación #1: Una variable para el estado actual.

      i. Escribir el programa `wl-1-enum-switch.c` que siga la Implementación #1, variante `enum` y `switch`.
      Esta implementación es la *regularización* de la implementación de la sección 1.5.4 de [KR1988]. Los *estados* son valores de una variable y las *transiciones* son la selección estructurada y la actualización de esa variable. Esta versión es menos eficiente que la versión de [KR1988], pero su regularidad permite la automatización de la construcción del programa que implementa la máquina de estados. Además de la regularidad, esta versión debe:

         - Utilizar `typedef` y `enum` en vez de `define`, de tal modo que la variable estado se pueda declarar de la siguiente manera: `State s = Out;`
         - Utilizar `switch` en vez de `if`.

      ii. Responder en `readme.md`: Indicar ventajas y desventajas de la versión de [KR1988] y de esta implementción.

   b. Implementación #2: Sentencias `goto` (sí, el infame *goto*)

i. Leer la sección *3.8 Goto and labels* de [KR1988]

ii. Leer *Go To Statement Considered Harmful* de [DIJ1968].

iii. Leer *"GOTO Considered Harmful" Considered Harmful* de [RUB1987].

iv. Responder en `readme.md`: ¿Tiene alguna aplicación *go to* hoy en día? ¿Algún lenguaje moderno lo utiliza?

v. Escribir el programa `wl-2-goto.c` que siga la Implementación #2.
En esta implementación los *estados* son *etiquetas* y las *transiciones* son la selección estructurada y el salto incondicional con la sentencia `goto`.

c. Implementación #3: Funciones Recursivas

i. Leer la sección *4.10 Recursividad* de [KR1988].

ii. Responder en `readme.md`: ¿Es necesario que las funciones accedan a a contadores? Si es así, ¿cómo hacerlo?.
Leer la sección *1.10 Variables Externas y Alcance* y *4.3 Variables Externas* de [KR1988].

iii. Escribir el programa, `wl-3-rec.c` que siga la implementación #3.
En esta implementación los *estados* son *funciones recursivas* y las *transiciones* son la selección estructurada y la invocación recursiva.

d. Implementación #X:
Es posible diseñar más implementaciones. Por ejemplo, una basada en una tabla que defina las transiciones de la máquina. En ese caso, el programa usaría la tabla para lograr el comportamiento deseado. El objetivo de este punto es diseñar una implementación **diferente** a las implementaciones #1, #2, y #3.

i. Diseñar una nueva implementación e indicar en `Readme.md` cómo esa implementación representa los estados y cómo las transiciones.

ii. Escribir el programa, `wl-x.c` que siga la nueva implementación.

4. Eficiencia del uso del Tiempo:
Construir una tabla comparativa a modo de *benchmark* que muestre el tiempo de procesamiento para cada una de las cuatro implementaciones, para tres archivos diferentes de tamaños diferentes, el primero en el orden de los

kilobytes, el segundo en el orden de los megabytes, y el tercero en el orden de los gigabytes.
La tabla tiene en las filas las cuatro implementación, en las columnas los tres archivos, y en la intersección la duración para una implementación para un archivo.

10.4. Restricciones

- Ninguna.

10.5. Productos

```
DD-wl
|-- readme.md
|   |-- Árboles de expresión.
|   |-- Respuestas.
|   `-- Benchmark.
|-- wl.gv
|-- Makefile
|-- wl-1-enum-switch.c
|-- wl-2-goto.c
|-- wl-3-rec.c
`-- wl-x.c
```

11

Máquinas de Estado — Contador de Palabras

Este trabajo está basado en el ejemplo de las sección *1.5.4 Conteo de Palabras* de [KR1988]:

> "… cuenta líneas, palabras, y caracteres, con la definición ligera que una palabra es cualquier secuencia de caracteres que no contienen un blanco, tabulado o nueva línea."

Problema: Determinar la cantidad de líneas, palabra, y caracteres que se reciben por la entrada estándar.

11.1. Objetivos

- Aplicar máquinas de estado para el procesamiento de texto.
- Implementar máquinas de estado con diferentes métodos.

11.2. Temas

- Árboles de expresión.
- Representación de máquinas de estado.
- Implementación de máquinas de estado.

11.3. Tareas

1. Árboles de Expresión

    a. Estudiar el programa del ejemplo las sección *1.5.4 Conteo de Palabras* de [KR1988].

    b. Dibujar el árbol de expresión para la inicialización de los contadores: `nl = nw = nc = 0`.

    c. Dibujar el árbol de expresión para la expresión de control del segundo if: `c == ' ' || c == '\n' || c == '\t'`.

2. Máquina de Estado:

    a. Describir en lenguaje `dot` ??? y dentro del archivo `wc.gv` la máquina de estado que resuelve el problema planteado.

    b. Formalizar la máquina de estados como una *n-upla*, basarse en el Capítulo #1 del Volumen #3 de [MUCH2012].

3. Implementaciones de Máquinas de Estado:
   Las implementaciones varían en los conceptos que utilizan para representaar los estados y las transiciones.

    a. Implementación #1: Una variable para el estado actual.

        i. Escribir el programa `wc-1-enum-switch.c` que siga la Implementación #1, variante `enum` y `switch`.
        Esta implementación es la *regularización* de la implementación de la sección 1.5.4 de [KR1988]. Los *estados* son valores de una variable y las *transiciones* son la selección estructurada y la actualización de esa variable. Esta versión es menos eficiente que la versión de [KR1988], pero su regularidad permite la automatización de la construcción del programa que implementa la máquina de estados. Además de la regularidad, esta versión debe:

            - Utilizar `typedef` y `enum` en vez de `define`, de tal modo que la variable estado se pueda declarar de la siguiente manera: `State s = Out;`
            - Utilizar `switch` en vez de `if`.

   ii. Responder en `readme.md`: Indicar ventajas y desventajas de la versión de [KR1988] y de esta implementción.

b. Implementación #2: Sentencias `goto` (sí, el infame *goto*)

   i. Leer la sección *3.8 Goto and labels* de [KR1988]

   ii. Leer *Go To Statement Considered Harmful* de [DIJ1968].

   iii. Leer *"GOTO Considered Harmful" Considered Harmful* de [RUB1987].

   iv. Responder en `readme.md`: ¿Tiene alguna aplicación *go to* hoy en día? ¿Algún lenguaje moderno lo utiliza?

   v. Escribir el programa `wc-2-goto.c` que siga la Implementación #2.
   En esta implementación los *estados* son *etiquetas* y las *transiciones* son la selección estructurada y el salto incondicional con la sentencia `goto`.

c. Implementación #3: Funciones Recursivas

   i. Leer la sección *4.10 Recursividad* de [KR1988].

   ii. Responder en `readme.md`: ¿Es necesario que las funciones accedan a a contadores? Si es así, ¿cómo hacerlo?.
   Leer la sección *1.10 Variables Externas y Alcance* y *4.3 Variables Externas* de [KR1988].

   iii. Escribir el programa, `wc-3-rec.c` que siga la implementación #3.
   En esta implementación los *estados* son *funciones recursivas* y las *transiciones* son la selección estructurada y la invocación recursiva.

d. Implementación #X:
   Es posible diseñar más implementaciones. Por ejemplo, una basada en una tabla que defina las transiciones de la máquina. En ese caso, el programa usaría la tabla para lograr el comportamiento deseado. El objetivo de este punto es diseñar una implementación **diferente** a las implementaciones #1, #2, y #3.

   i. Diseñar una nueva implementación e indicar en `Readme.md` cómo esa implementación representa los estados y cómo las transiciones.

   ii. Escribir el programa, `wc-x.c` que siga la nueva implementación.

4. Eficiencia del uso del Tiempo:

Construir una tabla comparativa a modo de *benchmark* que muestre el tiempo de procesamiento para cada una de las cuatro implementaciones, para tres archivos diferentes de tamaños diferentes, el primero en el orden de los kilobytes, el segundo en el orden de los megabytes, y el tercero en el orden de los gigabytes.
La tabla tiene en las filas las cuatro implementación, en las columnas los tres archivos, y en la intersección la duración para una implementación para un archivo.

11.4. Restricciones

- Ninguna.

11.5. Productos

```
DD-wc
|-- readme.md
|   |-- Árboles de expresión.
|   |-- Respuestas.
|   `-- Benchmark.
|-- wc.gv
|-- Makefile
|-- wc-1-enum-switch.c
|-- wc-2-goto.c
|-- wc-3-rec.c
`-- wc-x.c
```

12

Máquinas de Estado — Histograma de longitud de palabras

Este trabajo está basado en el ejercicio 1-13 de [KR1988] de la sección arreglos:

> Ejercicio 1-13. Escriba un programa que imprima el histograma de las longitudes de las palabras de su entrada. Es fácil dibujar el histograma con las barras horizontales; la orientación vertical es un reto más interesante.

Problema: Imprimir un histograma de las longitudes de las palabras de en la entrada estándar.

12.1. Objetivos

- Aplicar los conceptos de modularización
- Utilizar las herramientas de compilación y construcción de ejecutables estudiadas
- Aplicar máquinas de estado para el procesamiento de texto.
- Implementar máquinas de estado con diferentes métodos.

12.2. Temas

- Árboles de expresión.
- Representación de máquinas de estado.

- Implementación de máquinas de estado.
- Arreglos
- Flujos
- Modularización

12.3. Tareas

1. Árboles de Expresión

   a. Estudiar el programa del ejemplo las sección *1.5.4 Conteo de Palabras* de [KR1988].

   b. Dibujar el árbol de expresión para la inicialización de los contadores: `nl = nw = nc = 0`.

   c. Dibujar el árbol de expresión para la expresión de control del segundo if: `c == ' ' || c == '\n' || c == '\t'`.

2. Máquina de Estado:

   a. Describir en lenguaje `dot` ??? y dentro del archivo `histograma.gv` la máquina de estado que resuelve el problema planteado.

   b. Formalizar la máquina de estados como una *n-upla*, basarse en el Capítulo #1 del Volumen #3 de [MUCH2012].

3. Implementaciones de Máquinas de Estado:
   Las implementaciones varían en los conceptos que utilizan para representaar los estados y las transiciones.

   a. Implementación #1: Una variable para el estado actual.

      i. Escribir el programa `histograma-1-enum-switch.c` que siga la Implementación #1, variante `enum` y `switch`.
      Esta implementación es la *regularización* de la implementación de la sección 1.5.4 de [KR1988]. Los *estados* son valores de una variable y las *transiciones* son la selección estructurada y la actualización de esa variable. Esta versión es menos eficiente que la versión de [KR1988], pero su regularidad permite la automatización de la construcción del programa que implementa la máquina de estados. Además de la regularidad, esta versión debe:

- Utilizar `typedef` y `enum` en vez de `define`, de tal modo que la variable estado se pueda declarar de la siguiente manera: `State s = Out;`
- Utilizar `switch` en vez de `if`.

ii. Responder en `readme.md`: Indicar ventajas y desventajas de la versión de [KR1988] y de esta implementción.

b. Implementación #2: Sentencias `goto` (sí, el infame *goto*)

i. Leer la sección *3.8 Goto and labels* de [KR1988]

ii. Leer *Go To Statement Considered Harmful* de [DIJ1968].

iii. Leer *"GOTO Considered Harmful" Considered Harmful* de [RUB1987].

iv. Responder en `readme.md`: ¿Tiene alguna aplicación *go to* hoy en día? ¿Algún lenguaje moderno lo utiliza?

v. Escribir el programa `histograma-2-goto.c` que siga la Implementación #2.
En esta implementación los *estados* son *etiquetas* y las *transiciones* son la selección estructurada y el salto incondicional con la sentencia `goto`.

c. Implementación #3: Funciones Recursivas

i. Leer la sección *4.10 Recursividad* de [KR1988].

ii. Responder en `readme.md`: ¿Es necesario que las funciones accedan a a contadores? Si es así, ¿cómo hacerlo?.
Leer la sección *1.10 Variables Externas y Alcance* y *4.3 Variables Externas* de [KR1988].

iii. Escribir el programa, `histograma-3-rec.c` que siga la implementación #3.
En esta implementación los *estados* son *funciones recursivas* y las *transiciones* son la selección estructurada y la invocación recursiva.

d. Implementación #X:
Es posible diseñar más implementaciones. Por ejemplo, una basada en una tabla que defina las transiciones de la máquina. En ese caso, el programa usaría la tabla para lograr el comportamiento deseado. El

objetivo de este punto es diseñar una implementación **diferente** a las implementaciones #1, #2, y #3.

i. Diseñar una nueva implementación e indicar en `Readme.md` cómo esa implementación representa los estados y cómo las transiciones.

ii. Escribir el programa, `histograma-x.c` que siga la nueva implementación.

4. Escribir un único programa de prueba para las cuatro implementaciones.
5. (Opcional) Construir una tabla comparativa a modo de *benchmark* que muestre el tiempo de procesamiento para cada una de las cuatro implementaciones, para tres archivos diferentes de tamaños diferentes, el primero en el orden de los kilobytes, el segundo en el orden de los megabytes, y el tercero en el orden de los gigabytes.
   Eficiencia del uso del Tiempo:
   La tabla tiene en las filas las cuatro implementación, en las columnas los tres archivos, y en la intersección la duración para una implementación para un archivo.

12.4. Restricciones

- La implementación de la máquina de estado debe ser "seleccionable". Algunas formas posibles de implementar la selección son:
  - En tiempo de traducción desde el makefile.
  - En de tiempo de ejecución mediante reemplazlo de dynamic link library.
  - En de tiempo de ejecución mediante argumentos de la línea de comandos.
- La solución debe estar modularizada: las máquinas de estado no deben conocer del graficador y viceversa.
- Desde main.c se coordina todo.

Crédito extra

> Parametrizar si el histograma se dibuja vertical u horizontalmente.

12.5. Productos

```
DD-histograma
|-- readme.md
|   |-- Árboles de expresión.
|   |-- Respuestas.
|   `-- Benchmark.
|-- histograma.gv
|-- Makefile
|-- main.c
|-- Graficador.h
|-- Graficador.c
|-- Test.c
|-- histograma.h
|-- histograma-1-enum-switch.c
|-- histograma-2-goto.c
|-- histograma-3-rec.c
`-- histograma-x.c
```

13

Máquinas de Estado — Sin Comentarios

Este trabajo está basado en el ejercicio *1-23* de [KR1988]:

> Escriba un programa que remueva todos los comentarios de un programa C. Los comentarios en C no se anidan. No se olvide de tratar correctamente las cadenas y los caracteres literales

13.1. Objetivo

El objetivo es diseñar una máquina de estado que remueva comentarios, implementar dos versiones, e informar cual es la más eficiente mediante un benchmark.

13.2. Restricciones

- Primero diseñar y especificar la máquina de estado y luego derivar dos implementaciones.
- Utilizar el lenguaje `dot` para dibujar los digrafos.
- Incluir comentarios de una sola línea (`//`).
- Considerar las variantes no comunes de literales carácter y de literales cadenas que son parte del estándar de C.
- Diseñar el programa para que pueda invocarse de la siguiente manera: `RemoveComments < Test.c > NoComments.c`

- Ninguna de las implementaciones debe ser la *Implementación #1: estado como variable y transiciones con selección estructurada*.
- Indicar para la implementación cómo se representan los estados y cómo las transiciones.
- Respetar la máquina de estado especificada, en cada implementación utilizar los mismos nombres de estado y cantidad de transiciones.
- En el caso que sea necesario, utilizar `enum`, y no `define`.
- Diseñar el archivo `Makefile` para que construya una, otra o ambas implementaciones, y para que ejecute las pruebas.

13.3. Productos

```
DD-SinComentarios
|-- readme.md
|-- RemoveComments.gv
|-- RemoveComments.c
`-- Makefile
```

14

Preprocesador Simple

Este trabajo está basado en el ejercicio *6-6* de [KR1988]:

> Implemente una versión simple de la directiva de preprocesador `#define` (i.e., sin argumentos) aplicable a programas C, basados en las rutinas de esta sección. Puede encontrar útiles a `getch` y `ungetch`

14.1. Objetivo

El objetivo es diseñar e implementar una versión simple del preprocesador que:

- atienda directivas `#define` sin argumentos.
- atienda directivas `#undef`.
- (*opcionalmente*) atienda directivas `#include "` *filename* `"`.
- Reemplace comentarios por un espacio.

14.2. Temas

- Tabla de Símbolos.
- Máquinas de estado.
- `#define` con y sin argumentos.
- `#include` con comillas (`"..."`) y con corchetes angulares (`<...>`)

14.3. Restricciones

- La máquina de estados y la implementación deben ser una evolución del trabajo *"Máquinas de Estado — Sin Comentarios"*
- Respetar la máquina de estado especificada, en la implementación utilizar los mismos nombres de estado y cantidad de transiciones.
- Utilizar el lenguaje `dot` para dibujar el digrafo.
- La definición formal de la máquina de estados debe estar en `readme.md`.
- Diseñar el programa para que pueda invocarse de la siguiente manera: `> Preprocess < Test.c`
- En el caso que sea necesario, utilizar `enum` o `const`, y no `define`.
- Diseñar y aplicar un módulo *SymbolTable* que se base en la interfaz de la sección *6.6 Búsqueda en Tabla* de [KR1988]. Esta es una propuesta de interfaz para un módulo *SymbolTable* donde la tabla es única y está encapsulada:

Operación	Nombre en K&R	Una alternativa	Comentario
Agregar	install(name,text)	Set(name,text)	Si ya existe redefine, si no, agrega. Retorna el texto o `NULL` si no puedo agregarlo.
Buscar	lookup(name)	Get(name)	Si existe retorna el texto, si no, `NULL`.
Sacar	undef(name)	Remove(name)	Si existe remueve la definición, si no, `NULL`.

Propuesta de prototipos en `SymbolTable.h`:

```
const char* Set(const char* name, const char* text);
const char* Get(const char* name);
```

```
const char* Remove(const char* name);
```

14.4. Tareas

1. Escribir el archivo de test funcional: `Test.c`.
2. Especificar máquina de estado en `readme.md` y dibujar su digrafo en `Preprocess.gv`.
3. Desarrollar el módulo *SymbolTable*
    a. Diseñar interfaz: `SymbolTable.h`
    b. Escribir pruebas: `SymbolTableTest.c`
    c. Implementar: `SymbolTable.c`
4. Implementar máquina de estado: `Preprocess.c`
5. Probar: `> Prepocess < Test.c`

14.5. Productos

```
DD-SimplePreprocessor
|-- readme.md
|-- SymbolTable.h
|-- SymbolTable.c
|-- SymbolTableTest.c
|-- Preprocess.gv
|-- Preprocess.c
|-- Test.c
`-- Makefile
```

15

Parser Simple

Este trabajo está basado en el ejercicio *1-24* de [KR1988]:

> Escriba un programa para verificar errores sintácticos rudimentarios de un programa C, como paréntesis, corchetes, y llaves sin par. No se olvide de las comillas, apóstrofos, secuencias de escape, y comentarios. (Este programa es difícil si lo hace en su completa generalidad.)

15.1. Objetivo

El objetivo es diseñar e implementar un autómata de pila (APD) que verifique el balanceo de los paréntesis, corchetes, y llaves; en un programa C pueden estar anidados. La solución debe validar:

- Paréntesis, corchetes y llaves desbalanceados:
  - Válido: `{[()]}`
  - Inválido: `{[}(])`
- Apóstrofos y comillas, secuencias de escape:
  - Válido: `"[("`
  - Inválido: `"{}`

15.2. Temas

- Autómata de Pila (Push down Automata).

- Stacks.

15.3. Tareas

1. Escribir el archivo de test funcional: `Test.c`.
2. Especificar el formalmente el APD en `readme.md` y dibujar su digrafo en `Parser.gv`.
3. Desarrollar el módulo *Stack* que disponibilice una pila de caracteres.
    a. Diseñar interfaz: `StackOfCharsModule.h`
    b. Escribir pruebas: `StackOfCharsModuleTest.c`
    c. Implementar: `StackOfCharsModule.c`
4. Implementar el parser mediante el APD definido: `Parse.c`
5. Probar: `> Parse < Test.c`

15.4. Restricciones

- Diseñar el programa para que pueda invocarse de la siguiente maneras:
    - `> Preprocess < Test.c | Parse` Ó
    - `> RemoveComments < Test.c | Parse`
- Resolver APD según para eso leer Capítulo #2 del Volumen #2 de [MUCH2012].
- (*Opcional*) Considerar las variantes no comunes de literales carácter y de literales cadenas que son parte del estándar de C.
- Utilizar el símbolo **$** para la pila vacía.
- Respetar la máquina de estado especificada, en la implementación utilizar los mismos nombres de estado y cantidad de transiciones.
- Utilizar el lenguaje `dot` para dibujar el digrafo.
- La definición formal de la máquina de estados debe estar en `readme.md`.
- En `readme.md` indicar cómo se representan los estados y cómo las transiciones en la implementación del APD.
- Especificación en `readme.md` de *PushString* basada en operaciones de cadenas de lenguajes formales.

- Diseñar `PushString("xyz")` para que sea equivalente a `Push('z'), Push('y'), Push('x')`
- Diseñar el programa para que pueda invocarse de la siguiente manera: `> Parse < Test.c`
- En el caso que sea necesario, utilizar `enum` o `const`, y no `define`.
- Diseñar y aplicar un módulo *StackOfCharsModule*, insipirarse en las versiones de *stack* de [KR1988]

15.5. Productos

```
DD-SimpleParser
|-- readme.md
|-- StackOfCharsModule.h
|-- StackOfCharsModule.c
|-- Parser.gv
|-- Parser.c
`-- Makefile
```

16

Calculadora Infija: Construcción Manual — Iteración #1

16.1. Objetivos

- Experimentar el diseño de la especificación de lenguajes a nivel léxico y sintáctico.
- Experimentar la implementación manual del nivel léxico y sintáctico de lenguajes.

16.2. Temas

- Especificación del nivel Léxico y Sintáctico.
- Implementación del nivel Léxico y Sintáctico.
- Implementación de Scanner
- Implementación de Parser.

16.3. Problema

Análisis de expresiones aritméticas infijas simples que incluya:

- Números naturales con representación literal en base 10.
- Identificadores de variables.
- Adición.
- Multiplicación.

Ejemplos de expresiones incorrectas:

```
A+2*3
2*A+3
A
2
```

Ejemplos de expresiones incorrectas:

```
+
42+
+A
```

16.4. Solución

Especificar e implementar los niveles léxicos y sintácticos del lenguaje.

16.5. Restricciones

- El scanner y el parser deben estar lógicamente separados.
- El parser se comunica con el scanner con la operación `GetNextToken`, el scanner toma los caracteres de `stdin` con `getchar`.

Crédito Extra

> Estructurar la solución con separación física entre scanner y parser.

16.6. Tareas

1. Diseñar el nivel léxico del lenguaje.
2. Diseñar el nivel sintáctico del lenguaje.
3. Implementar el scanner.
4. Implementar el parser.
5. Probar.

16.7. Productos

```
DD-CalcInfManual
```

```
|-- Calc.md
|-- Makefile
|-- Scanner.h   // Opcional
|-- Parser.h    // Opcional
|-- Parser.c    // Opcional
|-- Scanner.c   // Opcional
`-- Calc.c
```

17

Calculadora Infija: Construcción Manual — Iteración #2

Extender la *calculadora* para que las expresiones puedan incluir *paréntesis* y para que las expresiones puedan *evaluarse*.

18

Calculadora Infija: Automática — Iteración #1

Implementar el *scanner* con *lex/flex*, mantener la interfaz establecida en `Scanner.h`.

19

Calculadora Infija: Automática — Iteración #2

Implementar el *parser* con *yacc/bison*, mantener la interfaz establecida en `Parser.h`.

20

Traductor de Declaraciones C a LN

Este trabajo está basado en el programa `dcl` ejemplo de las sección *5.12 Declaraciones Complicadas* de [KR1988].

20.1. Restricciones

- Aplicar los conceptos de modularización, componentes, e interfaces.
- Codificar `Scanner.h` y `Scanner.c`, para declarar y definir las siguientes declaraciones:

```
typedef enum {
    ...,
    LexError
}TokenType;

typedef ... TokenValue; // ¿Cuál es el valor de un token?

typedef struct {
    TokenType  type;
    TokenValue val;
} Token;

bool GetNextToken(Token *t /*out*/); // Retorna si pudo leer,
 almacena en t el token leido.
```

21

Traductor de Declaraciones C a LN con Lex

Este trabajo está basado en el programa `dcl` ejemplo de las sección *5.12 Declaraciones Complicadas* de [KR1988].

Esta versión se basa en un scanner generado por Lex.

22

Traductor de Declaraciones C a LN con Lex & Yacc

Este trabajo está basado en el programa `dcl` ejemplo de las sección *5.12 Declaraciones Complicadas* de [KR1988].

Esta versión se basa en un scanner generado por Lex y un parser generado por Yacc.

23

Trabajo #4 — Módulo Stack (?)

23.1. Objetivos

Construir dos implementaciones del Módulo Stack de `int`s.

23.2. Temas

- Módulos.
- Interfaz.
- Stack.
- Unit tests.
- `assert`
- Reserva estática de memoria.
- Ocultamiento de información.
- Encapsulamiento.
- Precondiciones.
- Poscondiciones.
- Call stack.
- heap.
- Reserva dinámica de memoria.
- Punteros.
- `malloc`.
- `free`.

23.3. Tareas

1. Analizar el stack de la sección 4.3 de [KR1988].
2. Codificar la interfaz `StackModule.h` para que incluya las operaciones:
    a. `Push`.
    b. `Pop`.
    c. `IsEmpty`.
    d. `IsFull`.
3. Escribir en la interfaz `StackModule.h` comentarios que incluya *especificaciones* y *pre* y *poscondiciones* de las operaciones.
4. Codificar los unit tests en `StackModuleTest.c`.
5. Codificar una implementación contigua y estática en `StackModuleContiguousStatic.c`.
6. Probar `StackModuleContiguousStatic.c` con `StackModuleTest.c`.
7. Codificar una implementación enlazada y dinámica en `StackModuleLinkedDynamic.c`.
8. Probar `StackModuleLinkedDynamic.c` con `StackModuleTest.c`.
9. Probar StackDynamic.c con StackTest.
10. Construir una tabla comparativa a modo de *benchmark* que muestre el tiempo de procesamiento para cada una de las dos implementaciones.
11. Diseñar el archivo `Makefile` para que construya una, otra o ambas implementaciones, y para que ejecute las pruebas.
12. Responder:
    a. ¿Cuál es la mejor implementación? Justifique.
    b. ¿Qué cambios haría para que no haya precondiciones? ¿Qué implicancia tiene el cambio?
    c. ¿Qué cambios haría en el diseño para que el stack sea genérico, es decir permita elementos de otros tipos que no sean `int`? ¿Qué implicancia tiene el cambio?
    d. Proponga un nuevo diseño para que el módulo pase a ser un *tipo de dato*, es decir, permita a un programa utilizar más de un stack.

23.4. Restricciones

- En `StackModule.h`:
  - Aplicar guardas de inclusión.
  - Declarar `typedef int StackItem;`
- En `StackModuleTest.c` incluir `assert.h` y aplicar `assert`.
- En ambas implementaciones utilizar `static` para aplicar encapsulamiento.
- En la implementación contigua y estática:
  - No utilizar índices, sí aritmética punteros.
  - Aplicar el *idiom* para stacks.
- En la implementación enlazada y dinámica:
  - Invocar a `malloc` y a `free`.
  - No utilizar el operador `sizeof(` *tipo* `)`, sí `sizeof` *expresión*.

23.5. Productos

- Sufijo del nombre de la carpeta: `StackModule`.
- `/Readme.md`
  - Benchmark.
  - Preguntas y Respuestas.
- `/StackModule.h`.
- `/StackModuleTest.c`
- `/StackModuleContiguousStatic.c`
- `/StackModuleLinkedDynamic.c`
- `/Makefile`

23.6. Entrega

Opcional.

24

Trabajo #5 — Léxico de la Calculadora Polaca (@)

Este trabajo está basado en el la sección 4.3 de [KR1988]: *Calculadora con notación polaca inversa.*

24.1. Objetivos

- Estudiar los fundamentos de los scanner aplicados a una calculadora con notación polaca inversa que utiliza un stack.
- Implementar modularización mediante los módulos `Calculator`, `StackOfDoublesModule`, y `Scanner`.

24.2. Temas

- Módulos.
- Interfaz.
- Stack.
- Ocultamiento de información.
- Encapsulamiento.
- Análisis léxico.
- Lexema.
- Token.
- Scanner.
- `enum`.

24.3. Tareas

1. Estudiar la implementación de las sección 4.3 de [KR1988].
2. Construir los siguientes componentes, con las siguientes entidades públicas:

Calculator	StackOfDoublesModule	Scanner
• Qué hace: Procesa entrada y muestra resultado. • Qué usa: ◦ Biblioteca Estándar ▪ `EOF` ▪ `printf` ▪ `atof` ◦ StackOfDoublesModule ▪ `StackItem` ▪ `Push` ▪ `Pop` ▪ `IsEmpty` ▪ `IsFull` ◦ Scanner ▪ `GetNextToken` ▪ `Token` ▪ `TokenType` ▪ `TokenValue`	• Qué exporta: ◦ `StackItem` ◦ `Push` ◦ `Pop` ◦ `IsEmpty` ◦ `IsFull`	• Qué hace: Obtiene operadores y operandos. • Qué usa: ◦ Biblioteca Estándar ▪ `getchar` ▪ `EOF` ▪ `isdigit` ▪ `ungetc` • Qué exporta: ◦ `GetNextToken` ◦ `Token` ◦ `TokenType` ◦ `TokenValue`

1. Diagramar en *Dot* las dependencias entre los componentes e interfaces.

2. Definir formalmente y con digrafo en *Dot* la máquina de estados que implementa `GetNextToken`, utilizar estados finales para diferentes para cada clase de tokens.
3. Escribir un archivo `expresiones.txt` para probar la calculadora.
4. Construir el programa `Calculator`.
5. Ejecutar `Calculator < expresiones.txt`.
6. Responder:

    a. ¿Es necesario modificar `StackModule.h`? ¿Por qué?

    b. ¿Es necesario recompilar la implementación de Stack? ¿Por qué?

    c. ¿Es necesario que `Calculator` muestre el lexema que originó el error léxico? Justifique su decisión.

        i. Si decide hacerlo, ¿de qué forma debería exponerse el lexema? Algunas opciones:

            - Tercer componente `lexeme` en `Token`¿De qué tipo de dato es aplicable?
            - Cambiar el tipo de `val` para que sea un `union` que pueda representar el valor para `Number` y valor `LexError`.

        ii. Implemente la solución según su decisión.

24.4. Restricciones

- Aplicar los conceptos de modularización, componentes, e interfaces.
- En `Calculator.c` la variable `token` del tipo `Token`, que es asignada por `GetNextToken`.
- Codificar `StackOfDoublesModule.h` a partir de la implementación contigua y estática de `StackModule`, `StackModuleContiguousStatic.c`, del trabajo #4, y modificar `StackItem`.
- Codificar `Scanner.h` y `Scanner.c`, para que usen las siguientes declaraciones:

```
enum TokenType {
    Number,
    Addition='+',
```

```
    Multiplication='*',
    Substraction='-',
    Division='/',
    PopResult='\n',
    LexError
};
typedef enum TokenType TokenType;
typedef double TokenValue;
struct Token{
    TokenType type;
    TokenValue val;
};
bool GetNextToken(Token *t /*out*/); // Retorna si pudo leer,
 almacena en t el token leido.
```

- `GetNextToken` debe usar una variable llamada `lexeme` para almacenar el lexema leído.
- Usar las siguientes entidades de la biblioteca estándar:
  - `stdio.h`
    - `getchar`
    - `EOF`
    - `stdin`
    - `printf`
    - `stdout`
    - `getchar`
    - `ungetc`
  - `ctype.h`
    - `isdigit`
  - `stdlib.h`
    - `atof`

24.5. Productos

- Sufijo del nombre de la carpeta: `PolCalc`.
- `/Readme.md`

    - Preguntas y Respuestas.
- `/expresiones.txt`
- `/Dependencias.gv`
- `/Calculator.c`
- `/StackOfDoublesModule.h`
- `/StackOfDoublesModule.c`
- `/Scanner.gv`
- `/Scanner.h`
- `/Scanner.c`
- `/Makefile`

24.6. Entrega

- Jul 3, 13hs.
    - Preentrega:
        - `StackOfDoublesModule.h`
        - `StackOfDoublesModule.c`
        - `Scanner.h`
        - `Scanner.gv`
- Jul 31, 13hs
    - Entrega final completa.

25

Trabajo #7 — Calculadora Polaca con Lex (@)

Este trabajo está es una segunda iteración de Capítulo 24, *Trabajo #5 — Léxico de la Calculadora Polaca (@)*, en la cual el *scanner* se implementa con *lex* y no con una máquina de estados.

25.1. Objetivo

Aplicar *lex* para el análisis lexicográfico.

25.2. Restricciones

- No cambiar `Scanner.h`, implica recompilar solo `Scanner.c` y volver a vincular.
- Utilizar *make* para construir el hacer uso de *lex*.
- La única diferencia está en `Scanner.c`, en el cual la función `GetNextToken` debe invocar a la función `yylex`.

25.3. Productos

- Sufijo del nombre de la carpeta: `PolCalLex`.
- Los mismos que Capítulo 24, *Trabajo #5 — Léxico de la Calculadora Polaca (@)* con la adición de `/Scanner.l`

25.4. Entrega

- Sep 6, 13hs.

    - Preentrega: `Scanner.l` con `main` que informa por `stdout` los tokens encontrados en `stdin`
- Sep 11, 13hs
    - Entrega final completa.

26

Trabajo #8 — Calculadora Infija con RDP (?)

Este trabajo es la versión infija de Capítulo 25, *Trabajo #7 — Calculadora Polaca con Lex (@)*; es decir en vez de procesar:

```
1 2 - 4 5 + *
-9
```

el programa debe procesar correctamente:

```
(1 - 2) * (4 + 5)
-9
```

26.1. Objetivo

- Diseñar una gramática independiente de contexto que represente la asociatividad y precedencia de las operaciones.
- Las operaciones son: `+ - * / ()`.
- Implementar un Parser Descendente Recursivo (RDP).

26.2. Restricciones

- Implementar `GetNextToken` con Lex, basado en el `GetNextToken` de Capítulo 25, *Trabajo #7 — Calculadora Polaca con Lex (@)*
- Agregar los tokens LParen y RParen.

26.3. Entrega

- Opcional

27

Trabajo #9 — Calculadora Infija con Yacc (?)

Esta vez, el parser lo construye Yacc por nosotros.

Bibliografía

Emden R. Gansner and Eleftherios Koutsofios and Stephen North. *Drawing graphs with dot* (2015) Retrived 2018-06-19 from https://www.graphviz.org/pdf/dotguide.pdf

Interfaz Fluida https://en.wikipedia.org/wiki/Fluent_interface

Git 101 https://josemariasola.wordpress.com/papers#Git101

Compiladores, Editores y Entornos de Desarrollo: Instalación, Configuración y Prueba https://josemariasola.wordpress.com/papers/#CompiladoresInstalacion

José María Sola. *Interfaces & Make* (2017) https://josemariasola.wordpress.com/ssl/papers#Interfaces-Make

Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, 2nd Edition* (1988)

Jorge Muchnik y Ana María Díaz Bott. *SSL, 2da Edición* (tres volúmenes) (2012)

Edsger W. Dijkstra. *Go To Statement Considered Harmful.* Reprinted from Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148. http://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf

Frank Rubin. *"Go To Statement Considered Harmful" Considered Harmful.* Reprinted from Communications of the ACM, Vol. 30, No. 3, March 1987, pp. 195-196. http://web.archive.org/web/20090320002214/http://www.ecn.purdue.edu/ParaMount/papers/rubin87goto.pdf

José María Sola. *Abstracciones, Listas Enlazadas, y For* https://josemariasola.wordpress.com/papers#ArraysPointersPrePosIncrement#AbstractionsLinkedListsAndForInCandCp

José María Sola. *Cadenas, Arreglos, Punteros, Pre, y Pos Incremento* https://josemariasola.wordpress.com/papers#ArraysPointersPrePosIncrement

José María Sola. *Niveles del Lenguaje: Léxico, Sintáctico, Semántico & Pragmático* (2011) https://josemariasola.wordpress.com/papers#LanguageLevels

Changelog

3.23.0+2021-08-24

- Trabajos *Preprocesador Simple* y *Parser Simple*: mejoras y correcciones para la legibilidad y guías para facilitar la resolución.