

Trabajos de Sintaxis y Semántica de los Lenguajes

Esp. Ing. José María Sola, profesor.

Revisión 3.1.0

2018-05-07

Tabla de contenidos

1. Introducción	1
2. Requisitos Generales para las Entregas de las Resoluciones	3
2.1. Requisitos de Forma	3
2.1.1. Repositorios	3
2.1.2. Lenguaje de Programación	6
2.1.3. Header Comments (Comentarios Encabezado)	6
2.2. Requisitos de Tiempo	7
3. "Hello, World!" en C	9
3.1. Objetivos	9
3.2. Temas	9
3.3. Tareas	9
3.4. Restricciones	10
3.5. Productos	10
4. Fases de la Traducción y Errores	11
4.1. Objetivos	11
4.2. Temas	11
4.3. Tareas	11
4.3.1. Secuencia de Pasos	11
4.4. Restricciones	13
4.5. Productos	13
5. Interfaces & Makefile — Temperaturas	15
5.1. Objetivos	15
5.2. Temas	15
5.3. Tareas	16
5.4. Restricciones	16
5.5. Productos	16
6. Trabajo #2 — Contador de Palabras (?)	17
6.1. Temas	17
6.2. Tareas	17
6.3. Productos	18
7. Trabajo #3 — Removedor de Comentarios	21
7.1. Objetivo	21
7.2. Restricciones	21
7.3. Productos	22

7.4. Entrega	23
8. Trabajo #4 — Módulo Stack (?)	25
8.1. Objetivos	25
8.2. Temas	25
8.3. Tareas	26
8.4. Restricciones	27
8.5. Productos	27
8.6. Entrega	27
9. Trabajo #5 — Léxico de la Calculadora Polaca (@)	29
9.1. Objetivos	29
9.2. Temas	29
9.3. Tareas	30
9.4. Restricciones	31
9.5. Productos	32
9.6. Entrega	33
10. Trabajo #6 — Parser Simple	35
10.1. Objetivo	35
10.2. Restricciones	35
10.3. Productos	36
10.4. Entrega	37
11. Trabajo #7 — Calculadora Polaca con Lex (@)	39
11.1. Objetivo	39
11.2. Restricciones	39
11.3. Productos	39
11.4. Entrega	39
12. Trabajo #8 — Calculadora Infija con RDP (?)	41
12.1. Objetivo	41
12.2. Restricciones	41
12.3. Entrega	42
13. Trabajo #9 — Calculadora Infija con Yacc (?)	43
14. Trabajo #10 — DCL con Lex	45
15. Trabajo #11 — DCL con Lex y con Yacc	47
Bibliografía	49

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/>

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 descritos, 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.

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 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.

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

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

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

Ejemplo de Estructura de Repositorios

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
 */
```

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

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](https://josemariasola.wordpress.com/ssl/calendar/)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. Tareas

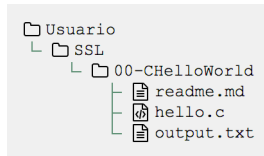
1. Solicitar inscripción al Grupo Yahoo, la aprobación demora un par de días.
2. Si no posee una cuenta *GitHub*, crearla.
3. Crear un repositorio público llamado *SSL*.
4. Escribir el archivo `readme.md` que actúa como *front page* del repositorio personal.
5. Crear la carpeta `00-HelloWorld`.
6. Escribir el archivo `readme.md` que actúa como *front page* de la resolución.
7. Seleccionar, instalar, y configurar un compilador **C11**.
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 de texto `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 por email a UTNFRBASSL@yahoogroups.com¹ el usuario *GitHub*.

3.4. Restricciones

- Ninguna.

3.5. Productos



¹ <mailto:UTNFRBASSL@yahoogroups.com>

Fases de la Traducción y Errores

4.1. Objetivos

- Identificar las fases de traducción y errores.

4.2. Temas

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

4.3. Tareas

1. Investigar las funcionalidades y opciones que su compilador presenta para limitar el inicio y fin de las fases de traducción.
2. Para la siguiente secuencia de pasos:
 - a. Transcribir en `readme.md` cada **comando ejecutado** y
 - b. Describir en `readme.md` el **resultado** u **error** obtenidos para cada paso.

4.3.1. Secuencia de Pasos

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

```
#include <stdio.h>

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

2. Preprocesar hello2.c, no compilar, y generar hello2.i. Analizar su contenido.
3. Escribir hello3.c, una nueva variante:

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

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

4. Investigar la semántica de la primera línea.
5. Preprocesar hello3.c, no compilar, y generar hello3.i. Buscar diferencias entre hello3.c y hello3.i.
6. Compilar el resultado y generar hello3.s, no ensamblar.
7. Corregir en el nuevo archivo hello4.c y empezar de nuevo, generar hello4.s, no ensamblar.
8. Investigar hello4.s.
9. Ensamblar hello4.s en hello4.o, no vincular.
- 10.Vincular hello4.o con la biblioteca estándar y generar el ejecutable.
- 11.Corregir en hello5.c y generar el ejecutable.
- 12Ejecutar y analizar el resultado.
- 13Corregir en hello6.c y empezar de nuevo.
- 14Escribir hello7.c, una nueva variante:

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


15Explicar porqué funciona.

4.4. Restricciones

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

4.5. Productos

5

Interfaces & Makefile — Temperaturas

Este trabajo está basado en los ejercicios 1-4 y 1-15 de [\[KR1988\]](#):

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.

5.1. Objetivos

- Realizar el primer trabajo en equipo en el repositorio privado del equipo en **GitHub**.
- Aplicar el uso de interfaces y de `makefile`.

5.2. Temas

- `makefile`.
- Archivos header (`.h`).

- Funciones.
- Pruebas unitarias.
- `assert`.
- Tipo de dato `double`.
- Interfaces e Implementación.

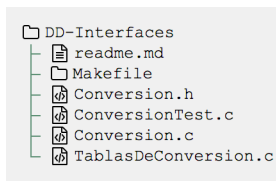
5.3. Tareas

1. Escribir el `Makefile`.
2. Escribir `conversion.h`
3. Escribir `conversionTest.h`
4. Escribir `conversion.c`
5. Escribir `tablasDeConversion.c`.

5.4. Restricciones

- Las funciones deben llamarse `celsius` y `fahrenheit`.
- Utilizar `assert`.
- Utilizar `const`.
- Utilizar `for` con declaración (C99).

5.5. Productos



6

Trabajo #2 — Contador de Palabras (?)

Este trabajo está basado en el ejemplo de la 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."

6.1. Temas

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

6.2. Tareas

1. Árboles de Expresión

- a. Dibuje el árbol de expresión para la inicialización de los contadores: `n1 = nw = nc = 0.`
- b. Dibuje el árbol de expresión para la expresión de control del segundo `if`:
`c == ' ' || c == '\n' || c == '\t'`

2. Enum y Switch

- a. Escriba una segunda versión del programa, `wc-enum-switch.c`, que:
 - i. Utilice `typedef` y `enum` en vez de `define`, de tal modo que la variable estado se pueda declarar de la siguiente manera: `state s = out;`
 - ii. Utilice `switch` en vez de `if`.
 - b. Responda: ¿Cómo implementa los estados este programa? ¿Y las transiciones?
3. Funciones Recursivas
- a. Escriba una tercera versión del programa, `wc-enum-switch.c.c`, que, en vez de una variable, utilice funciones recursivas para representar los estados.
4. Sentencias `goto` (sí, el infame *goto*)
- a. Lea la sección 3.8 *Goto and labels* de [\[KR1988\]](#)
 - b. Lea *Go To Statement Considered Harmful* de [\[DIJ1968\]](#).
 - c. Responda: ¿Tiene alguna aplicación *go to* hoy en día? ¿Algún lenguaje moderno lo utiliza?
 - d. Escriba una cuarta versión del programa, `wc-goto.c`, que, en vez de variable o funciones, utilice etiquetas para representar los estados y sentencias `goto` para las transiciones.
5. Eficiencia
- a. Construya una tabla comparativa a modo de *benchmark* que muestre el tiempo de procesamiento para cada una de las tres 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.

6.3. Productos

- Sufijo del nombre de la carpeta: `contador`.
- `readme.md`:
 - Árboles de expresión.

- Respuestas.
- Tabla de mediciones.
- `wc-enum-switch.c`.
- `wc-rec.c`.
- `wc-goto.c`.
- `Makefile`.

7

Trabajo #3 — Removedor de 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

7.1. 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.

7.2. 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 cada 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`.
- En el caso que sea necesario, utilizar `switch`, y no `if`.
- Realizar una prueba funcional y tres pruebas de volumen.
- Construir una tabla comparativa a modo de *benchmark* que muestre el tiempo de procesamiento para cada una de las dos 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.
- Crear a mano el archivo de test funcional: `Test.c`.
- Construir el programa `GenerateTest.c` que genere automáticamente los tres archivos para pruebas de volumen: `Testkilo.c`, `Testmega.c`, y `Testgiga.c`.
- No incorporar al repositorio los archivos de prueba de volumen, sí el de prueba funcional.
- Diseñar el archivo `Makefile` para que construya una, otra o ambas implementaciones, y para que ejecute las pruebas.

7.3. Productos

- Sufijo del nombre de la carpeta: `sincomentarios`.
- `/Readme.md`
 - Autómata finito para cada lenguaje.
 - Diagrama de transiciones.
 - Definición Formal.
 - Expresión regular para cada lenguaje.
 - Máquina de Estados del programa.

- Descripción de la implementación A: rc-a.c.
- Descripción de la implementación B: rc-a.c.
- Benchmark.
- /rc-a.c
- /rc-b.c
- /tests/Test.c
- /tests/GenerateTest.c
- /Makefile

7.4. Entrega

- Jun 5, 13hs

Trabajo #4 — Módulo Stack (?)

8.1. Objetivos

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

8.2. 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`.

8.3. 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.

8.4. 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`.

8.5. Productos

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

8.6. Entrega

Opcional.

9

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

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

9.1. 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`.

9.2. Temas

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

9.3. Tareas

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

Calculator	StackOfDoublesModule	Scanner
<ul style="list-style-type: none">• Qué hace: Procesa entrada y muestra resultado.• Qué usa:<ul style="list-style-type: none">◦ Biblioteca Estándar<ul style="list-style-type: none">▪ EOF▪ printf▪ atof◦ StackOfDoublesModule<ul style="list-style-type: none">▪ StackItem▪ Push▪ Pop▪ IsEmpty▪ IsFull◦ Scanner<ul style="list-style-type: none">▪ GetNextToken▪ Token▪ TokenType▪ TokenValue	<ul style="list-style-type: none">• Qué exporta:<ul style="list-style-type: none">◦ StackItem◦ Push◦ Pop◦ IsEmpty◦ IsFull	<ul style="list-style-type: none">• Qué hace: Obtiene operadores y operandos.• Qué usa:<ul style="list-style-type: none">◦ Biblioteca Estándar<ul style="list-style-type: none">▪ getchar▪ EOF▪ isdigit▪ ungetc• Qué exporta:<ul style="list-style-type: none">◦ 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.

9.4. 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='*',
    Subtraction='-',
    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 leído.

```

- 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

9.5. Productos

- Sufijo del nombre de la carpeta: Po1calc.
- /Readme.md

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

9.6. Entrega

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

10

Trabajo #6 — 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, apóstrofes, secuencias de escape, y comentarios. (Este programa es difícil si lo hace en su completa generalidad.)

10.1. 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: `{[]}()`
- Apóstrofes y comillas, secuencias de escape:
 - Válido: `"[{"`
 - Inválido: `"{}`

10.2. Restricciones

- Primero diseñar y el APD y luego derivar una implementación.

- Utilizar el lenguaje dot para dibujar el digrafo.
- Resolver con máquina de estados, para eso leer Capítulo #2 del Volumen #2 de [\[MUCH2012\]](#).
- Utilizar el símbolo \$ para la pila vacío.
- 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 | Parse
- Diseñar pushString("xyz") para que sea equivalente a push('z'), push('y'), push('x')
- Para la implementación indicar cómo se representan los estados y cómo las transiciones.
- Respetar la máquina de estado especificada, en la implementación utilizar los mismos nombres de estado y cantidad de transiciones.
- En el caso que sea necesario, utilizar enum, y no define.
- En el caso que sea necesario, utilizar switch, y no if.
- Crear a mano el archivo de test funcional: Test.c.
- Buscar la mejor forma para reutilizar la implemenación de *stack* de problemas anteriores.

10.3. Productos

- Sufijo del nombre de la carpeta: simpleParser.
- /Readme.md
 - Definición formal del APD.
 - Especificación de *PushString* basada en operaciones de cadenas de lenguajes formales.
- /StackOfCharsModule.h
- /StackOfCharsModule.c
- /Parser.gv
- /Parser.c

- /Makefile

10.4. Entrega

- Aug 22, 13hs.

11

Trabajo #7 — Calculadora Polaca con Lex (@)

Este trabajo está es una segunda iteración de [Capítulo 9, Trabajo #5 — Léxico de la Calculadora Polaca \(@\)](#), en la cual el *scanner* se implementa con *lex* y no con una máquina de estados.

11.1. Objetivo

Aplicar *lex* para el análisis lexicográfico.

11.2. 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`.

11.3. Productos

- Sufijo del nombre de la carpeta: `Polca1Lex`.
- Los mismos que [Capítulo 9, Trabajo #5 — Léxico de la Calculadora Polaca \(@\)](#) con la adición de `/Scanner.1`

11.4. Entrega

- Sep 6, 13hs.

- Preentrega: `scanner.1` con `main` que informa por `stdout` los tokens encontrados en `stdin`
- Sep 11, 13hs
 - Entrega final completa.

12

Trabajo #8 — Calculadora Infija con RDP (?)

Este trabajo es la versión infija de [Capítulo 11, 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
```

12.1. 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).

12.2. Restricciones

- Implementar getNextToken con Lex, basado en el getNextToken de [Capítulo 11, Trabajo #7 — Calculadora Polaca con Lex \(@\)](#)
- Agregar los tokens LParen y RParen.

12.3. Entrega

- Opcional

13

Trabajo #9 — Calculadora Infija con Yacc (?)

Esta vez, el parser lo construye Yacc por nosotros.

14

Trabajo #10 — DCL con Lex

Esta es la versión con Lex del programa de [\[KR1988\]](#).

15

Trabajo #11 — DCL con Lex y con Yacc

Esta es la versión con Lex y con Yacc del programa de [\[KR1988\]](#).

Bibliografía

[KR1988] Brian W. Kernighan and Dennis Ritchie. The C Programming Language, 2nd Edition. 1988.

[DIJ1968] Edsger W. Dijkstra. Go To Statement Considered Harmful. Reprinted from Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.

[MUCH2012] Jorge Muchnik y Ana María Díaz Bott. SSL, 2da Edición. 2012.

