Trabajos de Sintaxis y Semántica de los Lenguajes

Esp. Ing. José María Sola, profesor.

Revisión 1.1.1
2017-06-27

Tabla de contenidos

1

Introducción

El objetivo de los trabajos es afianzar los conocimientos y evaluar su comprensión. Deben resolverse en equipo, salvo que se indique que es individual, como es el caso del trabajo #0.

Los trabajos prefijados con **?** son opcionales y sirven como introducción a otros trabajos más complejos; pueden resolverse en equipo o individualmente. Pueden enviar la resolución para una devolución no prioritaria.

Los trabajos prefijados con **@** son comunes para todos los cursos.

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 descriptos en el enunciado de cada trabajo.

2.1. Requisitos de Tiempo

Cada trabajo establece la 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.

2.2. Requisitos de Forma

Requisitos de forma del repostiorio, las carpetas de las resoluciones, y los encabezados de los archivos fuente.

2.2.1. Repositorio del Equipo en GitHub

La entrega de cada resolución debe realizarse a través de *GitHub*, para ello, a cada equipo se le asigna un **repositorio privado**. Lugo, el equipo debe crear una carpeta particular para cada resolución.

2.2.2. Front Page del Equipo

En la raíz del repositorio cada equipo debe diseñar un archivo `readme.md` que actúe como front page del equipo. Debe estar escrito en notación *Markdown* y debe tener, como mínimo, la siguiente información:

- Asignatura.
- Curso.
- Año de cursada, y cuatrimestre si corresponde.
- Número de equipo.
- Nombre del equipo (opcional).
- Integrantes del equipo actualizados, ya que, durante la transcurso de la cursada el equipo puede cambiar:
  - Usuario *GitHub*.
  - Legajo.
  - Apellido.
  - Nombre.

2.2.3. Carpetas para cada Resolución

La resolución de cada trabajo debe tener su propia carpeta en el repositorio del equipo.

Además de los productos solicitados por cada trabajo, la carpeta **sí debe incluir**:

- un archivo `readme.md` que actúe como front page de la resolución.

Para facilitar el desarrollo se **recomienda incluir**:

- un archivo `.gitignore`.
- un archivo `Makefile`.
- archivos tests.

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

- archivos ejecutables.

- archivos intermedios producto del proceso de compilación o similar.

El sigueinte patrón como la carpeta se **debe nombrar**:

> ÚltimosTresDígitosDelCurso-DosDelEquipo-DosDígitosNúmeroTrabajo-NombreTrabajo

Ejemplo 2.1. Nombre de carpeta

051-02-00-Hello

2.2.4. Header Comments (Comentarios Encabezado)

Todo archivo fuente de 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
 */
```

2.2.5. Front Page de la Resolución

Cada de resolución debe contar con un archivo `readme.md`, escrito en *Markdown* que contenga, como mínimo, la siguiente información:

- Número de equipo.

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

3

@ Trabajo #0 — "Hello, World!" en C

Este es un trabajo individual.

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 la resolución de los trabajos.

3.2. Tareas

1. Solicitar inscripción al Grupo Yahoo.
2. Registrarse en GitHub.
3. Crear un repositorio público con el nombre `CHelloWorld`.
4. Seleccionar, instalar, y configurar un compilador **C11**.
5. Probar el compilador con un programa `hello.c` que envíe a `stdout` la línea `Hello, World!` o similar.
6. Ejecutar el programa, y capturar su salida en un archivo de texto `output.txt` .
7. Publicar `hello.c` y `output.txt` en GitHub.
8. Escribir y publicar el archivo `readme.md` del repositorio.
9. Enviar a UTNFRBASSL@yahoogroups.com[1] usuario GitHub.

[1] mailto:UTNFRBASSL@yahoogroups.com

3.3. Productos

- Repositorio público `CHelloWorld`.
- `readme.md`.
- `hello.c`
- `output.txt`.

3.4. Entrega

- Mar 20 (segunda clase), 13hs.

4

Trabajo #1 — Conversor de Temperaturas

Este trabajo está basado en el ejercicio 1-15 de [KR1988]:

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

4.1. Objetivos

- Realizar el primer trabajo en equipo en el repositorio privado del equipo en **GitHub**.
- Demostrar conocimiento de:
  - Funciones.
  - Archivos header (.h).
  - Interfaces e Implementación.
  - Uso de `make`.

4.2. Restricciones

- Utilizar `const`.
- Utilizar `for` con declaración (C99).

4.3. Productos

- Sufijo del nombre de la carpeta: `Temperatura`.
- `FahrCel.c`.
- `Conversion.h`.
- `Conversion.c`.
- `Makefile`.

4.4. Entrega

- Abr 17, 13hs

5

? Trabajo #2 — 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."

5.1. Temas

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

5.2. Tareas

1. Árboles de Expresión
    a. Dibuje el árbol de expresión para la inicialización de los contadores: `nl = 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.

5.3. 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`.

6

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

6.1. 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.

6.2. 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 generar 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.

6.3. 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`

6.4. Entrega

- Jun 5, 13hs

7

? Trabajo #4 — Módulo Stack

7.1. Objetivos

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

7.2. 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`.

7.3. 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. Escrbir 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.

7.4. 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*.

7.5. Productos

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

7.6. Entrega

Opcional.

8

@ Trabajo #5 — Léxico de la Calculadora Polaca

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

8.1. 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`.

8.2. Temas

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

8.3. Tareas

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

    - Calculator
        - 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`
    - StackOfDoublesModule
        - Qué exporta:
            - `StackItem`
            - `Push`
            - `Pop`

        - `IsEmpty`
        - `IsFull`
    - Scanner
        - Qué hace: Obtiene operadores y operandos.
        - Qué usa:
            - Biblioteca Estándar
                - `getchar`
                - `EOF`
                - `isdigit`
                - `ungetc`
        - Qué exporta:
            - `GetNextToken`
            - `Token`
            - `TokenType`
            - `TokenValue`
3. Diagramar en *Dot* las dependencias entre los componentes e interfaces.
4. Definir formalmente y con digrafo en *Dot* la máquina de estados que implementa `GetNextToken`, utilizar estados finales para diferentes para cada clase de tokens.
5. Escribir un archivo `expresiones.txt` para probar la calculadora.
6. Construir el programa `Calculator`.
7. Ejecutar `Calculator < expresiones.txt`.
8. 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.

8.4. 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`

8.5. 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`

8.6. Entrega

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

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.