Entrega 1:

Análisis léxico y sintáctico

PROCESADORES DE LENGuAJEs

Carlos Ruiz Ballesteros

Héctor ruiz-poveda coca

GIS-GII

Índice

[AUTORES 2](#_Toc446617678)

[DESCRIPCIÓN DEL CÓDIGO 2](#_Toc446617679)

[Analizador léxico 2](#_Toc446617680)

[Analizador sintáctico 2](#_Toc446617681)

[Interfaz gráfica y su uso 3](#_Toc446617682)

[EJEMPLOS DE EJECUCIÓN 4](#_Toc446617683)

[Control de errores en la entrada de argumentos 4](#_Toc446617684)

[Ejecución de la función *head* 4](#_Toc446617685)

[Ejecución de la función *tail* 5](#_Toc446617686)

[Ejecución de la función *longlines* 6](#_Toc446617687)

[Ejecución de la función *longlines* con un fichero como *stdin* 7](#_Toc446617688)

[Comentarios personales 8](#_Toc446617689)

[Problemas encontrados y decisiones tomadas 8](#_Toc446617690)

# AUTORES

Carlos Ruiz Ballesteros.

Héctor Ruiz-Poveda Coca.

Ambos autores somos del Doble Grado en Ingeniería del Software + Ingeniería Informática.

# DESCRIPCIÓN DEL CÓDIGO

El código se divide en dos partes principales:

## Analizador léxico

Por medio de la herramienta *“Flex”* se ha desarrollado un analizador léxico que permite identificar los distintos elementos: valores numéricos, identificadores, constantes literales, comentarios, retornos de carro…

El analizador léxico transforma estos elementos, así como palabras reservadas, en tokens y símbolos que utilizará el analizador sintáctico.

## Analizador sintáctico

La gramática que se encarga de comprobar si el código es sintácticamente correcto ha sido generada por *“CUP”*.

Este analizador se encarga de comprobar que la estructura y el orden del programa son correctas.

Contiene variables y métodos para el control y detección de errores en la sintáxis del fichero a analizar. Un ejemplo son las variables booleanas ***if\_expression*** y  ***case\_expression***, utilizadas para informar de un token incorrecto en la expresión aritmética de un *IF* o un *CASE*. Por otro lado cuenta con un método que informa la línea y la columna en la que se encuentra el error detectado ( ***syntax\_error(Symbol s)***).

Se encuentran también en el analizador los símbolos terminales, no terminales y las reglas gramaticales necesarias para el correcto análisis gramatical.

Con motivo de una correcta recuperación de errores, se ha incluido la regla ***DEFRANG***, la cual, en caso de un error, busca un corchete cerrado. En caso contrario, se ejecutaría la regla ***ALLTYPES***.

Cabe destacar que se han añadido las modificaciones pertinentes para añadir la gramática que permite el reconocimiento de matrices y registros, así como sentencias*”IF”*, *“WHILE”* y *“FOR”*.

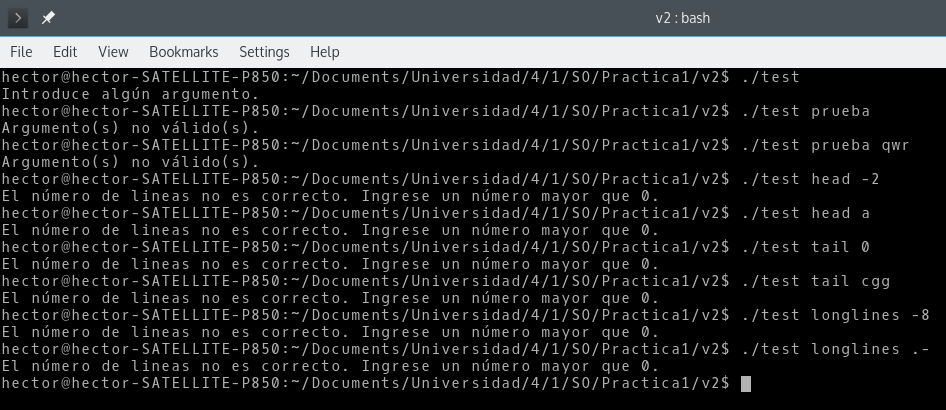
## Interfaz gráfica y su uso

Con motivo de una mejor usabilidad se ha desarrollado una interfaz gráfica, la cual permite seleccionar un fichero, analizarlo, y proporciona un correcto feedback al usuario sobre el resultado de dicho análisis.

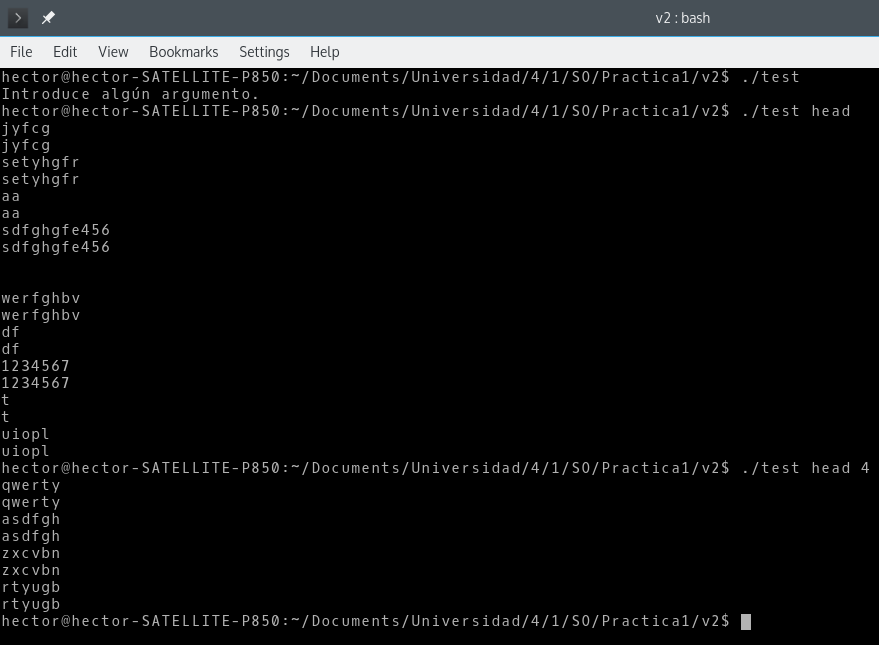
# EJEMPLOS DE EJECUCIÓN

A continuación, se muestran las capturas de pantallas correspondientes a la ejecución del código:

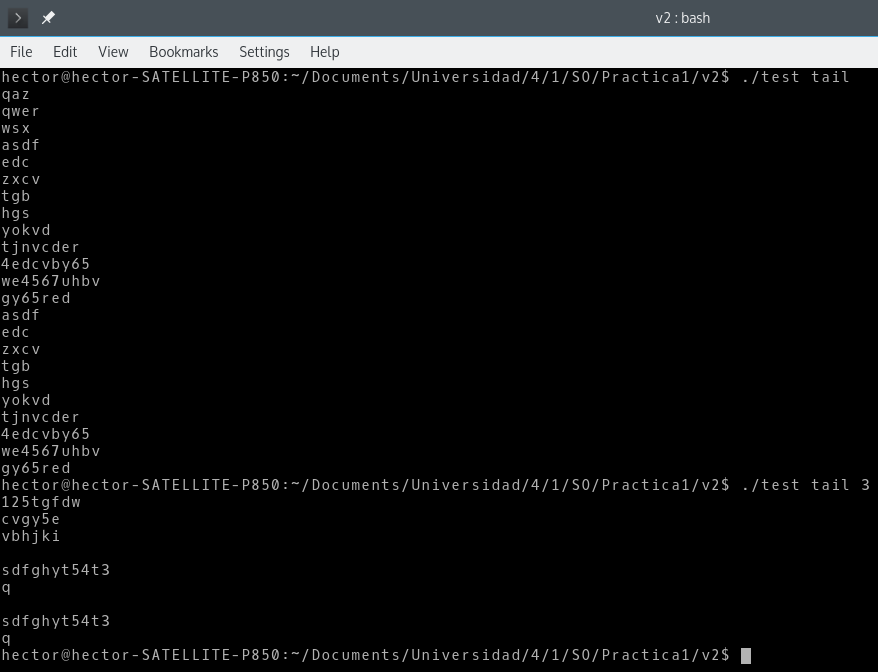
## Control de errores en la entrada de argumentos



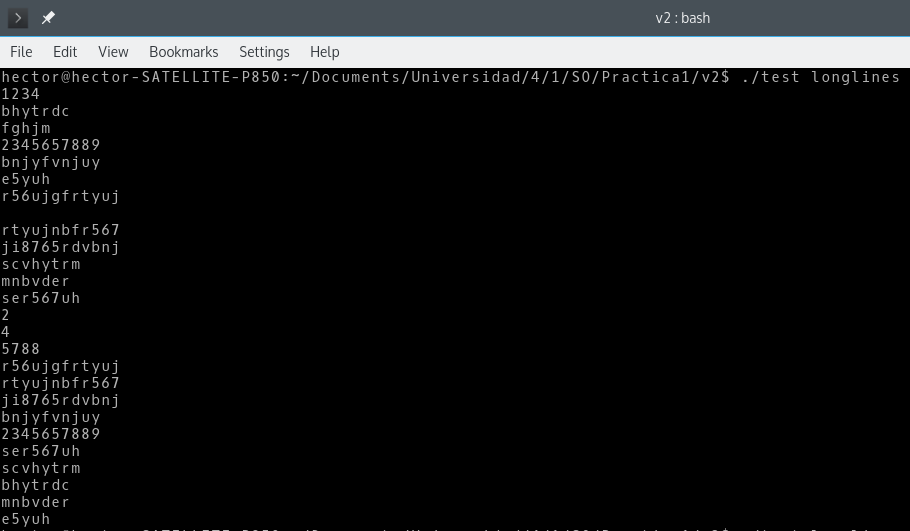
## Ejecución de la función *head*



## Ejecución de la función *tail*

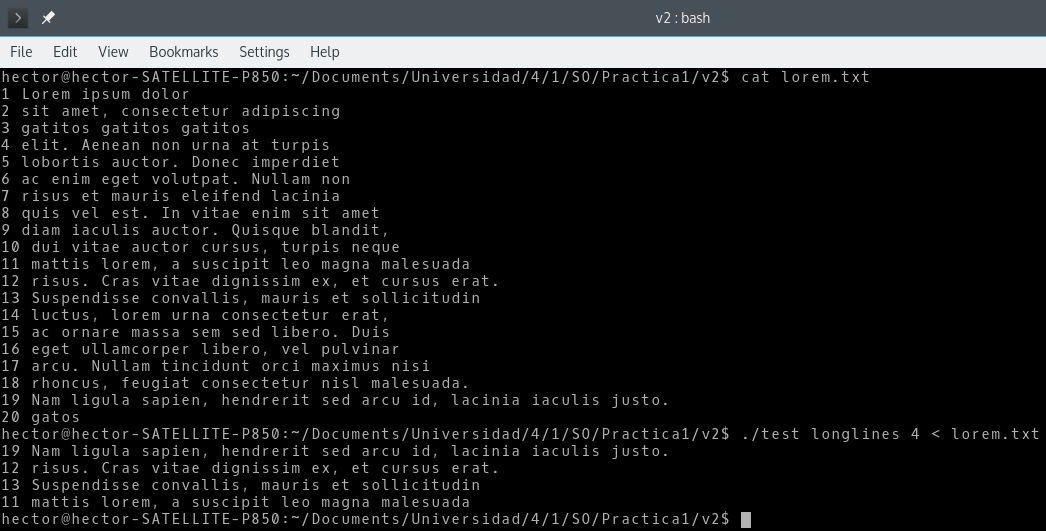


## Ejecución de la función *longlines*





## Ejecución de la función *longlines* con un fichero como *stdin*



# Comentarios personales

## Problemas encontrados y decisiones tomadas

He encontrado grandes dificultades al encontrarme ante un lenguaje que no he utilizado nunca, así como la dificultad añadida de no contar con un compañero en una práctica orientada a su resolución en grupos de dos y, además, escasez de tiempo debido a compaginar los estudios con una beca de colaboración concedida por la universidad.

Sin embargo, he sido capaz de solventar estas dificultades gracias a los apuntes de la asignatura, un esfuerzo extra y conocimientos previos de otros lenguajes como Java.

Durante la realización de la práctica me he encontrado con varios aspectos de la misma sobre los que he tenido que tomar decisiones.

Por ejemplo, la elección de la implementación de las funciones, generalmente buscando la mejor optimización y eficacia del código.

-En la función *tail* se ha elegido esta implementación porque resulta la más eficaz, pues se reserva el menor espacio de memoria posible y en cada iteración la complejidad es O(1).

-En la función *longlines*, por el contrario, la complejidad se eleva a O(n), pues por cada línea recibida tiene que volver a comprobar cuál es la de menor longitud.

Otro ejemplo es la decisión de limitar el tamaño máximo de las líneas de texto, buscando robustecer el código.

El tiempo dedicado a esta práctica me ha servido para asentar las bases de otro lenguaje de programación más, permitiendome tener un punto de partida para seguir evolucionando en el aprendizaje del lenguaje.