

Tokenizador

Alba Carmona Quirante

Febrero/Marzo 2025

Contents

1	Diseño del tokenizador de casos especiales	2
2	Mejoras probadas	4
2.1	Mejoras de estructuras de datos para complejidad constante	4
2.1.1	Mejoras consideradas	4
2.2	Mejoras de saltos condicionales	5
2.2.1	Mejoras consideradas	5
2.3	Mejoras de acceso a archivos	5
3	Complejidad temporal	6
4	Complejidad espacial	7

1 Diseño del tokenizador de casos especiales

El tokenizador de casos especiales ha sido subdividido en varias funciones según los delimitadores más condicionantes dados los requerimientos de algunos casos especiales. Así, si el punto no es delimitador, entonces la tokenización se realizará en una función que no contempla acrónimos ni números decimales ¹.

De este modo, se han creado tantas funciones como combinaciones hay de posibilidad e imposibilidad de que se detecte cada caso especial, resultando en las siguientes:

- TokenizarCasosEspeciales_U
- TokenizarCasosEspeciales_UA
- TokenizarCasosEspeciales_UE
- TokenizarCasosEspeciales_UM
- TokenizarCasosEspeciales_UEM
- TokenizarCasosEspeciales_UAM
- TokenizarCasosEspeciales_UDA
- TokenizarCasosEspeciales_UAM
- TokenizarCasosEspeciales_UAE
- TokenizarCasosEspeciales_UDAE
- TokenizarCasosEspeciales_UMAE
- TokenizarCasosEspeciales_UDEAM

El objetivo de esta configuración ha sido ahorrar saltos condicionales dentro de bucles que están mirando carácter por carácter, en el que se comprobaba, por ejemplo, si el guion es un delimitador antes de empezar a leer como si fuera una multipalabra.

Debido a que estas funciones solamente se llaman una vez, cuando se llama a la función tokenizar, y cada una solo requiere de cuatro saltos condicionales antes de entrar, no se considera una inversión temporal significativa.

Del mismo modo, y esto quizá sea más negativo, también se ha separado el procesamiento de cada caso especial en su propia función. Esto provoca que dentro del bucle de la función superior, que itera por todo el archivo, el control esté entrando y saliendo de funciones muy a menudo (cada vez que se decide que el token podría ser una cosa, y cada vez que se decide que no lo puede ser). Esto, es probable que vaya en detrimento de la eficiencia temporal del programa, pero considero que dada la claridad y facilidad de debug que aporta a las funciones superiores, en un programa que ya de por sí he complicado bastante con mi solución propuesta, vale la pena el gasto.

¹Se ha hecho una interpretación del enunciado en la que solo se admiten números decimales si tanto la coma (,) como el punto (.) están establecidos como delimitadores, aunque bien podría haberse referido a una indiferencia entre uno u otro.

Las funciones creadas para cada caso especial son las siguientes:

- `bool Tokenizar_ftp(const string& str, list<string>& tokens, int& i, string& curr_token) const`
- `bool Tokenizar_http(const string& str, list<string>& tokens, int& i, string& curr_token) const`
- `bool Tokenizar_decimal(const string& str, list<string>& tokens, int& i, string& curr_token, char heading_zero) const`
- `bool Tokenizar_email_O(const string& str, list<string>& tokens, int& i, string& curr_token) const`
- `bool Tokenizar_email_A(const string& str, list<string>& tokens, int& i, string& curr_token, bool &canBeAcronym) const`
- `bool Tokenizar_email_M(const string& str, list<string>& tokens, int& i, string& curr_token, bool &canBeMultiword) const`
- `bool Tokenizar_email_AM(const string& str, list<string>& tokens, int& i, string& curr_token, bool &canBeAcronym, bool &canBeMultiword) const`
- `bool Tokenizar_acronimo(const string& str, list<string>& tokens, int&i, string& curr_token) const`
- `Tokenizar_multipalabra(const string& str, list<string>& tokens, int&i, string& curr_token) const`
- `Tokenizar_token_normal(const string& str, list<string>& tokens, int&i, string& curr_token) const`

Los métodos de control de acceso a archivos funcionan como es esperable: `Tokenizar(str)` llama a `Tokenizar(str, str)` con el nombre de archivo de salida por defecto, `TokenizarDirectorio(str)` genera un archivo mediante un comando `find`, que luego `TokenizarListaFicheros(str)` lee, etc.

2 Mejoras probadas

2.1 Mejoras de estructuras de datos para complejidad constante

Se han definido dos estructuras de datos para lograr que ciertas operaciones que hay que hacer muy a menudo, como la comprobación de delimitadores y la conversión a minúsculas sin acento, tengan complejidad constante.

En primer lugar se define un array de `unsigned char` de 256 posiciones, que define la conversión a minúsculas sin acento. Se establece como `constexpr` para que se defina en tiempo de compilación, ayudando a la eficiencia temporal ligeramente.

Se define adicionalmente otro array no constante de `unsigned char` de 256 posiciones, en el que se guarda qué caracteres están establecidos como delimitadores. Esta tabla se lleva paralelamente al string `this->delimiters`; contienen datos idénticos excepto por el hecho de que el espacio, el salto de línea y el carácter nulo siempre están definidos como delimitadores en la tabla, para facilitar el control en ciertas operaciones a nivel más bajo.

Ambas tablas permiten un acceso tal que:

```
textttchar c = conversion[str[i]];
```

2.1.1 Mejoras consideradas

Se ha considerado usar otro array de `short int` en el que se definan los caracteres que están establecidos como excepciones para cada tipo de caso especial. Esto hubiera servido para reducir lo que en su día eran muchos ifs a uno solo que le pregunte a esta tabla.

Inicialmente hubiera creído que sobraría con asignar un número a cada caso especial y asignárselo a cada char, pero algunos caracteres eran excepciones para varios casos especiales. Definir más números para aquellos caracteres que son excepciones para varios casos me hubiera traído de vuelta el problema, así que probé de otra manera.

Terminé haciendo una tabla de verdad en la que cada dígito se corresponde a un caso especial. Pude entonces asociar cada carácter a una fila de la tabla de verdad, permitiéndome encontrar el mínimo número de estados posibles que representaban correctamente las relaciones. Salió lo siguiente:

UDEAM

00000 // caracteres que no son excepción

01000 // ',' excepción para decimal

10000 // ':/?&=' excepción para URL

10100 // '@' excepción para URL e email

10101 // '-' excepción para URL, email y multipalabra

11111 // '.' excepción para URL, email, decimal, acrónimo y multipalabra

Esto, para empezar, ahora que lo veo no estaba bien, ya que el punto no es un delimitador excepción para multipalabra. En segundo lugar, para el decimal no le era posible dejarlo en una sola comprobación, ya que eran tanto el estado 2 como el 5. De modo que esta idea se descartó por completo en favor de un switch, como hubiera sido obvio para cualquier otra persona.

2.2 Mejoras de saltos condicionales

Relacionado con el sistema de métodos descrito en la sección de diseño, se ha intentado evitar hacer ciertas comprobaciones varias veces cuando es seguro que su resultado no va a cambiar de una iteración a otra. Esto en el caso mencionado ha requerido duplicar métodos masivamente, y en otros simplemente ha hecho que se haga cierta comprobación al principio de un método de caso especial concreto, y se duplique el código (cambiado ligeramente según la condición) en cada parte del `if-else`.

Esto incluye comprobar al principio de los métodos si hay que pasar a minúsculas sin acentos, de modo que en caso negativo el programa se ahorra un `if` en cada caracter de entrada.

También se ha tomado la decisión de comprobar los caracteres especiales de cada caso especial mediante un `switch`, que debido a que internamente genera una `jump table` a posiciones de memoria concretas, es menos costoso que un `if-else` y además a menudo ofrecía más claridad.

2.2.1 Mejoras consideradas

La alternativa a esto hubiera sido hacer que el array `conversion` pudiera apuntar a uno de dos arrays: uno con las conversiones reales y otro con los valores normales de los caracteres del 0 al 255.

Se ha decidido no hacer esto por dos razones. La primera es la absurdidad de tener un array de 256 elementos con los números del 0 al 255 en orden. Y la segunda es que, en caso de que no haya que pasar a minúsculas, en cada caracter se estaría haciendo una llamada innecesaria a la sobrecarga del operador `[]` del array, que aunque es poco costosa, es esperable que se note un poco la diferencia con grandes volúmenes de datos.

2.3 Mejoras de acceso a archivos

De lejos la mejora que más ha afectado a la eficiencia temporal y espacial del tokenizador ha sido usar **memory mapped files** en lugar de la librería `fstream` de C++.

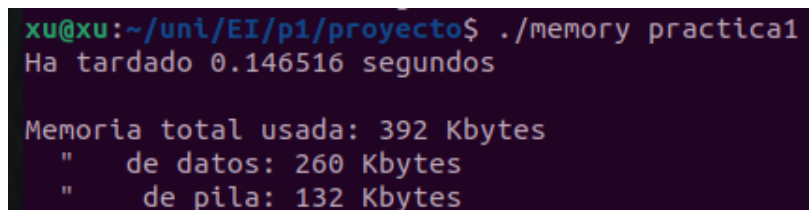
Esta librería (`sys/mmap.h`) permite mapear una serie de direcciones de memoria asociadas a un archivo a un array dentro del programa, permitiendo acceso directo con poco overhead tanto a los archivos de entrada como a los de salida.

En primer lugar, esta mejora ha permitido no tener que cargar todo el contenido del archivo en un string, proporcionando una mejora masiva en la eficiencia espacial del programa, ya que todo eso se convierte en un solo puntero. Al usar memory mapping también para el archivo de salida, también ha permitido escribir y rectificar los caracteres resultantes a medida que se van obteniendo, con un coste de operaciones muy bajo (ahorrando `resizes`, `substrings`... en la versión con strings).

Esto ha ahorrado un recorrido lineal (temporal) entero, mientras que introduce únicamente el control de un par de contadores para el archivo de entrada y el de salida.

También, al estar escribiendo directamente en el archivo de salida, la lista de tokens y el string de entrada dados en el prototipo de la clase quedan obsoletos y pueden no manejarse cuando se está accediendo a archivos, ahorrando de nuevo dos recorridos lineales (espaciales) enteros. Desafortunadamente, esto también implica tener que definir un nuevo ejército de métodos que reciben los punteros en lugar de los strings y las listas, lo cual ha complicado bastante el proceso de implementación y ha llenado el header de cosas. En un futuro, definitivamente le dedicaré más tiempo a la fase de diseño para pensar en maneras de que no se me vaya de las manos la complejidad del programa como ha ocurrido en esta práctica.

3 Complejidad temporal

A terminal window with a dark background and light-colored text. The prompt is 'xu@xu:~/uni/EI/p1/proyecto\$'. The command executed is './memory practica1'. The output shows the execution time as 'Ha tardado 0.146516 segundos' and memory usage as 'Memoria total usada: 392 Kbytes', with sub-entries for data ('de datos: 260 Kbytes') and stack ('de pila: 132 Kbytes').

```
xu@xu:~/uni/EI/p1/proyecto$ ./memory practica1
Ha tardado 0.146516 segundos

Memoria total usada: 392 Kbytes
"   de datos: 260 Kbytes
"   de pila: 132 Kbytes
```

Figure 1: Coste temporal/espacial empírico

En mi portátil, con un procesador 12th Gen Intel(R) Core(TM) i7-12700H, el programa ofrecía los resultados mostrados.

Las funciones descritas en el apartado de diseño iteran por todo el archivo de manera lineal, llamando por el camino a funciones a nivel de token que técnicamente también tienen coste lineal, ya que también iteran por el archivo. Esto no significa que la complejidad resultante sea extracuadrática (ya que llama a varias funciones lineales a nivel de token), ya que como se ve al final de dicho apartado, reciben el iterador y lo avanzan internamente, de manera que en un caso ideal se está leyendo el archivo una sola vez, con complejidad lineal.

Sin embargo, mi algoritmo en casos concretos encuentra la necesidad de volver atrás para volver a leer. Este puede ser el caso de toparse con 123.123,123.123,123a. Siendo un caso relativamente rebuscado, no está en los tests dados, y no he llegado a saber si este token debería partirse en la última coma, dejando un decimal por detrás y un token normal por delante, o si debería partirse desde la primera coma y desmoronarse a partir de ahí.

Finalmente decidí hacer que cuando el decimal falla simplemente se vuelva al caracter inicial, una solución bastante ineficiente, pero no supe mejorarla de manera segura. Como este hay varios ejemplos, pero es suficiente para ilustrar que la complejidad temporal de mi programa será, como mínimo, extralineal con las dimensiones del archivo de entrada.

En el peor caso de que tokens como estos pueblen el archivo entero, el programa terminaría haciendo entre dos y tres pasadas por cada caracter, suponiendo una **complejidad temporal en el caso peor de $O(3N)$** .

En el mejor caso, no se tiene que hacer más de una pasada y la **complejidad temporal resultante en el mejor caso $\Omega(N)$** .

Adicionalmente, con las mejoras introducidas gracias al memory mapping, no es necesario hacer una pasada final para escribir en el archivo ni redimensionar strings sistemáticamente cada vez que se completa un token, de modo que esas operaciones, que hubieran sido de coste lineal cada una, son ahorradas.

4 Complejidad espacial

La complejidad espacial viene dada directamente por el memory mapping realizado sobre los archivos, mediante el cual reserva espacio para el archivo de entrada un array y para el archivo de salida en otro array. Aunque al final se trunca la longitud del de salida para ajustarse a su contenido, lo esperable es que la memoria que requiera sea más o menos la misma que el archivo de entrada.

Dado que nos hemos librado de tener que llevar la lista de tokens y el string de entrada, esto nos lleva a concluir que el programa tiene una complejidad espacial en el **peor caso de hasta $O(2N)$** (el caso en el que el archivo de salida es igual de grande que el de entrada) y **$\Omega(N)$ en el mejor caso** (en el caso en el que, por ejemplo, el archivo de entrada solo contenía delimitadores y el archivo de salida queda vacío).