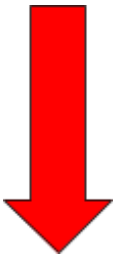
Memoria del Tokenizador

```
Hola Mundo! hoy es 12/03/2021 y hacen +23ºC
```



```
hola
mundo
hoy
es
12
03
2021
y
hacen
+23
C
```

Asignatura: Explotación de la información(EI)

Número de la Práctica: 1

Alumno: Saul Verdu Aparicio

Curso: 2020-2021

Índice

Introducción

En esta práctica se nos ha pedido construir y optimizar un tokenizador que dado un string consigas segmentarlo en diferentes tokens, pudiendo detectar una serie de casos especiales (URLs, eMails, Acronimos y Multipalabras). Esta práctica nos servirá como base para poder implementar un buscador en las futuras prácticas, por ello es de gran importancia el tamaño que ocupa nuestra práctica y la velocidad de la misma. Para reducir estos dos parámetros se han implementado diferentes mejoras, que se comentarán más adelante.

Análisis de la Solución Implementada

Tras varias pruebas con diferentes implementaciones la solución final que se tomó es parecida a la que se nos mostró en el tokenizador visto en clase. En este caso lo que se hace es sacar la posición más cercana de un carácter que no sea delimitador ni espacio en blanco, y la posición del delimitador o espacio en blanco más cercano. Estas se almacenan en las variables 'lastPos' y 'pos'. Ya con estos dos valores almacenados, se pasa a comprobar el contenido del string entre 'lastPos' y 'pos' para saber si pertenece a alguno de los casos especiales y por último se almacena. Esto se hará tantas veces como sean necesarias hasta que se haya tokenizado el sting por completo.

```
    std::string::size_type lastPos = copia_str.find_first_not_of
(delimitersAux, 0); // Posicion del primer caracter del Token
    std::string::size_type pos = copia_str.find_first_of
(delimitersAux, lastPos);   // Posicion del primer delimitador

    while (std::string::npos != pos || std::string::npos != lastPos)
    {
        //Comprobamos si es una URL
        if (esURL(lastPos, copia_str))...

        //Comprobamos si es una Numero
        else if (esNumero(lastPos, copia_str, pos))...

        //Comprobamos si es un Email
        else if (esEmail(lastPos, copia_str, pos))...

        //Comprobamos si es un Acronimo
        else if (esAcronimo(lastPos, copia_str, pos))...

        //Comprobamos si es una Multipalabra
        else if (esMultPalab(lastPos, copia_str, pos))...

        //Si no se mete el token directamente
        else...

        lastPos = copia_str.find_first_not_of(delimitersAux, pos);
        pos = copia_str.find_first_of(delimitersAux, lastPos);
    }
}
```

Las funciones ‘esNumero()’, ‘esAcronimo()’, ‘esMultPalab()’ se encarga de comprobar si el token pertenece a alguna de estas categorías y actualizar la posición de pos. Por otro lado las funciones ‘esURL()’ y ‘esEmail()’ solo comprueban si pertenece a esa categoría.

```
bool Tokenizador::esMultPalab(const std::string::size_type &lastPos, const std::string &str, std::string::size_type &pos) const
{
    if(str[pos] == '-'){
        std::string::size_type posAux;
        posAux = Encontrar_final(lastPos, str, delimitersMul);
        if(
            delimiters.find_first_of("-", 0) != std::string::npos &&    // Comprobamos que se este utilizando el '-'
            posAux > str.find_first_of("-", lastPos)                    // y que haya algo despues del guion
        ){
            pos = posAux;
            return true;
        }
    }
    return false;
}
```

```
bool Tokenizador::esEmail(const std::string::size_type &lasPos, const std::string &str, const std::string::size_type &pos) const
{

    std::string::size_type posAux, posAux2, posAux3;
    posAux = str.find_first_of("@", pos + 1);
    posAux2 = str.find_first_of(" ", pos + 1);
    posAux3 = str.find_first_of(delimitersMail + ".-_", pos + 1);

    return (
        str[pos] == '@' &&                                  //Comprobamos que el delimitador que se ha encontrado es un '@'
        (posAux == std::string::npos || posAux2 < posAux) && //Comprobamos que no hayan '@' despues del primer '@'
        posAux3 != pos + 1                                  //Comprobamos que despues del '@' no venga ningun delimitador
    );
}
```

Dentro del bucle principal en la mayoría de los casos especiales lo único que se hace es añadir el token la lista de tokens, salvo en el caso de que sea un número el cual se comprobará si se debe de añadir un 0 delante de él y si se deben añadir los caracteres ‘%’ y ‘$’ como un nuevo token.

```
//Comprobamos si es una Multipalabra
else if (esMultPalab(lastPos, copia_str, pos))
{
    //std::cout << "Soy una Multipalabra" << std::endl;
    tokens.push_back(copia_str.substr(lastPos, pos - lastPos));
}
```

```
//Comprobamos si es una Numero
else if (esNumero(lastPos, copia_str, pos))
{
    //std::cout << "Soy un Numero" << std::endl;
    if (copia_str[lastPos - 1] == '.' || copia_str[lastPos - 1] == ',')
    {
        tokens.push_back('0' + copia_str.substr(lastPos - 1, pos - (lastPos - 1)));
    }
    else
    {
        tokens.push_back(copia_str.substr(lastPos, pos - lastPos));
    }

    //Si se termina por '%' o '$' se a?ade eso como un token extra
    if (copia_str[pos] == '$' || copia_str[pos] == '%')
    {
        std::string temp;
        temp = copia_str[pos];
        tokens.push_back(temp);
        ++pos;
    }
}
```

Por último, cabe recalcar que para ciertos casos especiales era necesario detectar que no se repitiera dos delimitadores seguidos (aunque el caso especial aceptara ese delimitador) ni que contuviera un delimitador aceptado al final del token, para ello ha sido necesario crear un método que sirve para calcular el valor de ‘pos’ teniendo en cuenta esto, llamado ‘Encontrar_Final()’.

```
std::string::size_type Tokenizador::Encontrar_final(const std::string::size_type &lasPos,
const std::string &str, const std::string &del) const
{
    std::string::size_type pos, aux1, aux2;
    pos = str.find_first_of(del, lasPos);
    aux1 = str.find_first_of(delimitersAux, lasPos);
    aux2 = str.find_first_of(delimitersAux, aux1 + 1);

    while (aux1 <= pos && aux1 != std::string::npos)
    {
        if (aux1 == aux2 - 1)
        {
            pos = aux1;
        }
        aux1 = aux2;
        aux2 = str.find_first_of(delimitersAux, aux1 + 1);
    }
    if  (
            pos == std::string::npos &&
            delimiters.find_first_of(str[str.length() - 1], 0) != std::string::npo
        ){
        pos = str.length() - 1;
    }

    return pos;
}
```

Este método necesita que se le pasen por parámetro los delimitadores que son aceptados por el caso especiales. Estos se calculan en los constructores y las funciones que se usan para el cambio de delimitadores.

Justificación de la Solución Implementada

Antes de llegar a este modelo se probaron otros diferentes pero se acabó optando por este ya que era fácil de comprender, implementar y mantener en caso de que se tuviera que hacer cambios en su estructura.

La primera de las opciones que se barajó fue la de introducir el contenido del string en un StreamString e ir dividiendo el string primeramente por espacios en blanco para su posterior procesamiento. Esta estructura se descartó porque requería demasiado tiempo para sacar los datos del StreamString.

```
std::stringstream strStream(str);
std::string aux;

while(strStream >> aux){
    ...
}
```

Otra de las opciones que se pensó para esta práctica era la de leer uno por uno los caracteres del string y en función de lo que se obtuviera se realizará una acción u otra. Este método también se descartó como solución por la elevada complejidad que suponía su implementación (la cual era más propensa a bugs).

```
char caracter;
std::string buf;

for (size_t i = 0; i < str.legth(); ++i)
{
    caracter = str[i];
    ...
    buf += caracter;
    ...
    tokens.push_back(buf);
}
```

Mejoras Realizadas

Tras realizar la práctica se comprobó la eficiencia de la misma y se implementaron diversas mejoras para el funcionamiento, entre ellas las más importantes son:

- **Reducción de accesos a memoria:** dado que esta práctica realiza accesos a memoria para la lectura y escritura de ficheros, fue necesario disminuir el número de llamadas a memoria que hacía dura la lectura o escritura. Para ello hicimos que los métodos encargados de hacer estas tareas solo tuvieran que hacer una única llamada.

- **Eliminación de acciones repetitivas:** Dentro del algoritmo se detectaron diferentes acciones que se repetían en cada una de las iteraciones (p.e cálculo de las cadenas que acepta cada uno de los casos especiales, calculo del final del Token...) las cuales causaban unos tiempos de ejecución muy elevados.

- **Reducción del número de parámetros en las funciones:** Al principio de la práctica la mayoría de las funciones usadas para detectar si se trataban de casos especiales utilizaban una gran cantidad de parámetros. Tras ver que este paso de parámetros provocaba que el rendimiento de la práctica disminuyese se optó por reducir el número de parámetros que se pasaban y que se pasasen solo como parámetro de lectura.

Complejidad

Para realizar el cálculo de la complejidad temporal debemos observar la función en la que se apoya todo el tokenizador, es decir, la complejidad temporal dependerá de:

```
void Tokenizador::Tokenizar(const std::string &str, std::list<std::string> &tokens) const
```

Para expresar la complejidad temporal se hará en función de **n** que hace referencia a la longitud del string a tokenizar y **d** que hace referencia a el número de delimitadores que tenemos.

- Coste temporal teórico mejor caso : O(n*d)
- Coste temporal teórico peor caso: Ω(5n*d)
- Coste espacial teórico: O(1)

Coste temporal real:

```
saul@saul-VirtualBox:~/Escritorio/EI/P1$ ./practica1
Ha tardado 3.03152 segundos
saul@saul-VirtualBox:~/Escritorio/EI/P1$ ./practica1
Ha tardado 2.95554 segundos
saul@saul-VirtualBox:~/Escritorio/EI/P1$ ./practica1
Ha tardado 2.96547 segundos
saul@saul-VirtualBox:~/Escritorio/EI/P1$ ./practica1
Ha tardado 2.85835 segundos
```

Coste espacial real:

```
Memoria total usada: 740 Kbytes
  "    de datos: 608 Kbytes
  "     de pila: 132 Kbytes
```