

# **EDKIT: EDITOR KIT**

José E. Marchesi  
Diciembre 2004

UNIVERSIDAD POLITÉCNICA DE MADRID  
ESCUELA UNIVERSITARIA DE INFORMÁTICA  
TRABAJO DE FIN DE CARRERA

EDKIT, version 1.0

Copyright © 2003 José E. Marchesi, Juan Garbajosa Sopena.

Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la GNU Free Documentation License, Version 1.1 o cualquier versión posterior publicada por la Free Software Foundation; sin secciones invariantes, siendo el texto de la portada "Edkit". Se incluye una copia de la licencia en la sección titulada "GNU Free Documentation License" en este mismo documento.

# **EDKIT: EDITOR KIT**

José E. Marchesi  
Juán Garbajosa Sopena  
Diciembre 2004

UNIVERSIDAD POLITÉCNICA DE MADRID  
ESCUELA UNIVERSITARIA DE INFORMÁTICA  
TRABAJO DE FIN DE CARRERA

## Resumen del Contenido

Lista de figuras . . . . .	1
Lista de tablas . . . . .	2
Prefacio . . . . .	3
1 Edición de texto y editores . . . . .	4
2 Codificación del texto . . . . .	26
3 Algorítmica del texto . . . . .	44
4 Buffers de texto . . . . .	55
5 EDKIT: Editor Kit . . . . .	89
6 Conclusiones y líneas futuras . . . . .	104
A GNU Free Documentation License . . . . .	105
Índice de funciones . . . . .	112
Índice conceptual . . . . .	113
Bibliografía . . . . .	118

# Índice General

Lista de figuras .....	1
Lista de tablas .....	2
Prefacio .....	3
<b>1 Edición de texto y editores .....</b>	<b>4</b>
1.1 El proceso de edición .....	4
1.2 El modelo de edición del usuario .....	5
1.2.1 Modelos de la edición .....	5
1.2.1.1 Array unidimensional de caracteres .....	5
1.2.1.2 Array bidimensional de caracteres .....	6
1.2.1.3 Lista de líneas .....	7
1.2.1.4 Modelos paginados .....	8
1.2.1.5 Modelos estructurados .....	8
1.2.2 Interfaz del usuario .....	8
1.2.2.1 Dispositivos de entrada .....	8
1.2.2.2 Dispositivos de salida .....	9
1.2.2.3 El lenguaje de interacción .....	10
1.3 Taxonomía de los editores de texto .....	13
1.4 Descomposición conceptual de un editor .....	15
1.4.1 El proceso de edición .....	16
1.4.1.1 El procesador del lenguaje de comandos .....	17
1.4.1.2 El componente de edición .....	17
1.4.2 El proceso de visualización .....	18
1.5 Descomposición funcional de un editor .....	18
1.6 Origen y evolución de algunos editores .....	21
1.6.1 Tape/Text Editor and Corrector: TECO .....	21
1.6.2 QED .....	21
1.6.3 El editor estándar de Unix: ED .....	22
1.6.4 EMACS .....	23
<b>2 Codificación del texto .....</b>	<b>26</b>
2.1 Introducción .....	26
2.2 Definiciones y conceptos .....	26
2.2.1 Repertorios de caracteres .....	26
2.2.2 Códigos de caracteres .....	27
2.2.3 Codificaciones de caracteres .....	28
2.2.4 CCS + CES .....	29
2.3 La familia ISO-8859 .....	29
2.4 Páginas de códigos en MS Windows .....	31
2.5 ISO 10646, UCS y Unicode .....	32

2.5.1	ISO 10646: UCS	33
2.5.2	Unicode	35
2.5.3	Codificaciones de UCS/Unicode	36
2.5.4	Codificaciones nativas: UCS-2 y UCS-4	36
2.5.5	Formatos de transformación	37
2.5.6	UTF-16	37
2.5.7	UTF-8	38
2.6	Aspectos de implementación	40
2.6.1	Representaciones internas y externas	41
2.6.2	La librería C estándar	41
2.7	El caso GNU Emacs	42
<b>3</b>	<b>Algorítmica del texto</b>	<b>44</b>
3.1	Introducción	44
3.2	Notación y definiciones previas	44
3.3	Búsqueda de patrones fijos	45
3.3.1	Fuerza bruta	46
3.3.2	Knuth-Morris-Pratt	47
3.3.3	Boyer-Moore	52
3.3.4	Adecuación a codificaciones multi-octeto	53
<b>4</b>	<b>Buffers de texto</b>	<b>55</b>
4.1	Introducción	55
4.2	Estructuración interna	57
4.2.1	Secuencias editables	57
4.2.2	Vectores	58
4.2.3	Buffer gap	61
4.2.4	Listas enlazadas	63
4.2.5	Líneas en tramos	64
4.2.6	Buffers de tamaño fijo	66
4.2.7	Tablas de piezas	67
4.2.8	Evaluación de los métodos	70
4.2.8.1	Comparativa de Finseth	70
4.2.8.2	Comparativa de Crowley	76
4.3	Estructuración externa	78
4.3.1	Localizaciones	79
4.3.2	Punteros y marcas	79
4.3.3	Tramos	80
4.3.4	Listas de tramos	81
4.3.5	Árboles de tramos	82
4.3.6	Grafos de tramos	83
4.3.7	Widening y Narrowing	84
4.3.8	Vistas	85
4.4	El caso GNU Emacs	86

<b>5</b>	<b>EDKIT: Editor Kit .....</b>	<b>89</b>
5.1	Motivación .....	89
5.2	Editores cliente .....	92
5.3	Arquitectura .....	94
5.4	Funcionalidades .....	95
5.4.1	Codificación de caracteres .....	95
5.4.2	Buffers .....	96
5.4.3	Facilidades básicas de búsqueda .....	97
5.4.4	Tablas de caracteres .....	97
5.5	Interfaz de programación .....	99
5.5.1	Interfaz para secuencias de texto .....	99
5.5.2	Inicialización y finalización .....	100
5.5.3	Gestión de buffers .....	100
5.5.4	Gestión de localizaciones .....	101
5.5.5	Gestión del puntero .....	101
5.5.6	Gestión de marcas .....	101
5.5.7	Gestión del contenido del buffer .....	101
5.5.8	Búsqueda de patrones fijos .....	102
5.5.9	Gestión de tablas de caracteres .....	102
	Obteniendo y almacenando información .....	102
	Creación y destrucción de tablas .....	102
	Narrowing .....	102
	Predicados .....	102
<b>6</b>	<b>Conclusiones y líneas futuras .....</b>	<b>104</b>
	<b>Apéndice A GNU Free Documentation License</b>	
	.....	<b>105</b>
	A.0.1 ADDENDUM: How to use this License for your documents	
	.....	111
	<b>Índice de funciones .....</b>	<b>112</b>
	<b>Índice conceptual .....</b>	<b>113</b>
	<b>Bibliografía .....</b>	<b>118</b>

## Lista de figuras

Figura 1.1: Array unidimensional de caracteres .....	6
Figura 1.2: Array bidimensional de caracteres .....	7
Figura 1.3: Lista de líneas .....	8
Figura 1.4: Descomposición conceptual de un editor .....	16
Figura 1.5: Descomposición funcional de un editor .....	19
Figura 2.1: Codificación ISO-8859 .....	29
Figura 2.2: Codificación MS Windows .....	32
Figura 2.3: Espacio UCS .....	34
Figura 2.4: Carácter en un buffer Emacs .....	42
Figura 3.1: Fuerza Bruta .....	46
Figura 3.2: Knuth-Morris-Pratt .....	49
Figura 3.3: Inicialización simple de la tabla <code>next</code> .....	50
Figura 3.4: Inicialización la tabla <code>next</code> .....	51
Figura 3.5: Boyer-Moore .....	53
Figura 4.1: Estructura compuesta para buffers .....	56
Figura 4.2: Jerarquía de Crowley para conjuntos ordenados .....	58
Figura 4.3: Secuencia en un vector .....	59
Figura 4.4: Secuencia en un vector con espacio al final .....	60
Figura 4.5: Secuencia en buffer gap .....	61
Figura 4.6: Coordenadas en buffer gap .....	62
Figura 4.7: Secuencia en lista enlazada .....	64
Figura 4.8: Secuencia con líneas en tramos .....	65
Figura 4.9: Secuencia en buffers de tamaño fijo .....	66
Figura 4.10: Secuencia en tabla de piezas .....	68
Figura 4.11: Estructura de la tabla de piezas .....	68
Figura 4.12: Operación de borrado en tabla de piezas .....	69
Figura 4.13: Operación de inserción en tabla de piezas .....	70
Figura 4.14: Localizaciones en una secuencia .....	79
Figura 4.15: Regiones definidas por marcas .....	80
Figura 4.16: Estructura de un tramo .....	81
Figura 4.17: Lista de tramos .....	82
Figura 4.18: Árbol de tramos .....	82
Figura 4.19: Grafo de tramos .....	83
Figura 4.20: Widening y Narrowing .....	84
Figura 4.21: Dos vistas de un buffer .....	85
Figura 5.1: Dos editores en un entorno CASE .....	90
Figura 5.2: Dos editores en un entorno CASE, con EDKIT .....	91
Figura 5.3: EDKIT y un editor cliente .....	92
Figura 5.4: EDKIT y dos editores cliente .....	93
Figura 5.5: Aportación funcional del EDKIT .....	93
Figura 5.6: Arquitectura del kit .....	94
Figura 5.7: Estructura lógica de una tabla de caracteres .....	97
Figura 5.8: Una jerarquía de tablas de caracteres .....	99

## Lista de tablas

Tabla 2.1: Familia ISO 8859 .....	31
Tabla 2.2: Equivalencias Unicode-UCS.....	35
Tabla 2.3: Codificación UTF-8 .....	39
Tabla 4.1: Comparación Finseth en almacenamiento requerido.....	71
Tabla 4.2: Comparación Finseth en recuperación de errores .....	72
Tabla 4.3: Comparación Finseth en la eficiencia de edición .....	73
Tabla 4.4: Comparación de Crowley .....	78

## Prefacio

Antes de nada, en calidad de autor de este libro y de los contenidos que lo componen, deseo aclarar los motivos principales que lo justifican. ¿Qué hace un libro que trata de la edición del texto como un trabajo de fin de carrera?. ¿Dónde está la “tecnología puntera” que suele caracterizar estos documentos?.

Este libro trata acerca de una tecnología que, si bien está presente en prácticamente todos los campos de la computación desde hace cuarenta años, continúa constituyendo las bases de casi cualquier actividad desarrollada delante de una computadora. El hecho de que cualquier usuario de computadoras (desde el mero usuario, pasando por el profesional y culminando por el investigador mas avezado) invierta al menos el noventa por ciento de su tiempo de cómputo en editar documentos convierte a los editores de texto en el mayor interactor con el usuario.

Cualquier persona interesada en comprender cómo funciona un editor de texto y las tecnologías involucradas en su esquema funcional encontrará este libro útil.

Claramente nadie pretende encontrar algun tipo de *innovación* en un trabajo de fin de carrera. La culminación de los estudios de la ciencia/ingeniería informática únicamente requiere de cierto grado de aportación y una gran capacidad de entendimiento y condensación del área de elección de cada alumno. Sin embargo, resulta muy difícil investigar algun área de tecnología sin sacar una serie de conclusiones, parte de las cuales siempre son “innovadoras”. Luego es posible encontrar ideas y conceptos innovadores en este documento.

El objetivo principal del trabajo consiste en afrontar los nuevos desafíos al área de la implementación de editores de texto que suponen las nuevas plataformas de aplicaciones, como Java. Estas plataformas fomentan la creación de editores de propósito específico: planteamientos que hace no mucho tiempo hubieran parecido desproporcionados. Para ello se propone una arquitectura que separa la implementación de las funcionalidades comunes de cualquier editor de las mas especializadas de un editor de propósito específico. Para cubrir este objetivo es necesario repasar los fundamentos y los avances acontecidos en el área tecnológica correspondiente a la edición de texto.

No quiero dar paso al contenido del libro sin mencionar antes su calidad de *documentación libre*. En virtud de la licencia de distribución que el lector podrá encontrar al final del libro, el usuario ve satisfechos sus derechos de lectura, copia y redistribución. Incluso es lícito modificar aquellas secciones del libro no catalogadas como invariantes en la notificación de derechos de copia.

José E. Marchesi

Madrid, 11 de Septiembre, 2004

# 1 Edición de texto y editores

[...] *Existen muchas formas de descomponer la implementación de un editor de texto en piezas mas pequeñas. Este libro analiza una de ellas: un sub-editor que gestiona el texto que se edita, la actualización de pantalla (redisplay), y el módulo de gestión de los comandos introducidos por el usuario (no hay mas piezas: cuando han sido ensambladas, el editor está completado). Esta descomposición ha sido escogida por dos motivos. Primero, es muy natural y disfruta de interfaces relativamente simples. Segundo, se ha puesto en práctica en muchas implementaciones: es por tanto una descomposición que se sabe trabaja bien.*

- Craig A. Finseth: The Craft of Text Editing

## 1.1 El proceso de edición

Un *editor interactivo* es un programa de ordenador que permite al usuario crear y/o revisar un *documento objetivo*. Utilizamos la palabra *documento* para indicar objetos como páginas de texto y programas en su forma escrita, pero tambien imágenes, jerarquías de pantallas, piezas musicales, etc. A lo largo de este libro nos limitaremos a la *edición de texto*, en donde el documento objetivo está compuesto de información textual organizada como una secuencia de caracteres. En sucesivos capítulos se desarrollan los conceptos de *secuencia* y *caracter*.

Luego estamos hablando de *editores de texto*, utilizados para editar documentos de texto tales como novelas o programas.

El *proceso de edición de texto* responde a las características usuales de un diálogo interactivo humano-máquina:

1. Selección de la parte del documento que deseamos visualizar o cambiar.
2. Visualización en pantalla (u otro dispositivo de salida, como una impresora) de dicha parte del documento en pantalla.
3. Especificación y ejecución de las operaciones de edición pertinentes, que modifican el documento bajo edición.
4. Actualización de la vista en pantalla al documento modificado en el paso anterior.

La selección de la parte del documento a editar requiere una **localización** (que requiere muchas veces de un desplazamiento) para determinar las áreas del documento que van a editarse. Esta localización se lleva a cabo con las operaciones típicas de movimiento entre caracteres, línea, siguiente pantalla, etc. A continuación, habiendo determinado ya **donde** está el área de interés, es necesario determinar **qué** partes del documento van a editarse. Esto se realiza mediante una fase de **filtrado**, que determina las porciones del documento objetivo que son editadas y visualizadas. Estas porciones de documento deben ser entonces **formateadas** para generar la **vista** y presentarla al usuario utilizando algun dispositivo de salida, como una pantalla o una impresora. Es entonces cuando, ya en la fase de edición propiamente dicha, el contenido del documento es alterado mediante al utilización de operaciones de edición como “borrar caracter” o “insertar la letra a”. Un buen conjunto

de operaciones de edición suele incluir operaciones para insertar, borrar, reemplazar, mover y copiar información dentro del documento bajo edición.

Todo lo anterior determina el “proceso de edición” expresado en términos de diálogo entre el usuario y el editor. En un caso sencillo un usuario podría **localizar** (o viajar) la última línea de un documento. Se generaría una **vista** de dicha línea en la pantalla, producto de una operación de filtrado (hay que extraer la línea del resto del documento) y formateado (hay que añadir un fin de línea en la pantalla). Finalmente el usuario podría borrar la primera palabra de la línea mediante operaciones de edición.

## 1.2 El modelo de edición del usuario

El usuario de un editor de texto tiene acceso a un *modelo conceptual* del sistema de edición (que es la visión del diseñador del proceso de edición), que manipula via una *interfaz de usuario* (que proporciona los mecanismos de interacción apropiados).

El modelo conceptual proporciona al usuario una visión abstracta del documento objetivo y los objetos que lo componen, así como una serie de normas y reglas que permiten anticipar los efectos en el documento de las operaciones de edición.

Por su parte la interfaz de usuario se compone de los dispositivos de entrada, los dispositivos de salida y un lenguaje de interacción con el modelo conceptual.

### 1.2.1 Modelos de la edición

#### *modelo de edición*

Un *modelo de edición* es la visión del objeto bajo edición que un editor presenta al usuario. Esta sección describe algunos modelos comunes y utilizados en la práctica.

El modelo de edición que presenta un editor al usuario no tiene relación (en principio) con las *funcionalidades* exportadas. Pese a que, como veremos, en un modelo paginado es muy probable que el usuario disponga de algunos comandos relacionados con páginas, esto no implica que otro usuario que utilice un editor que siga otro modelo de edición (orientado a líneas, por ejemplo) no pueda trabajar con páginas (aunque si, probablemente, en peores condiciones).

Finalmente hay que destacar que los modelos presentados aquí son específicamente modelos de edición de **texto**.

#### 1.2.1.1 Array unidimensional de caracteres

La forma mas general de modelar un texto es mediante un array unidimensional de caracteres. En este modelo los caracteres que conforman el texto codificado se presentan al usuario tal cual, sin ninguna traducción intermedia.

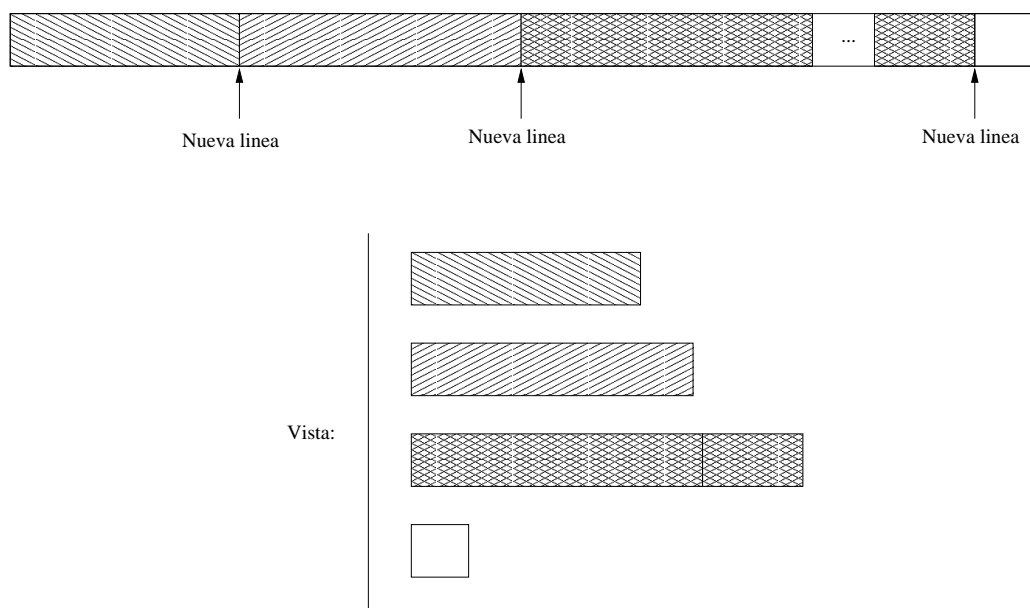


Figura 1.1: Array unidimensional de caracteres

Las operaciones de edición básicas de este modelo son *insertar* y *borrar* un caracter.

Este modelo es de muy bajo nivel y presenta ciertas dificultades al usuario: no existe una visión de estructuración mas allá de la secuencia. Por eso los editores de texto que siguen este modelo traducen algunos caracteres (el caracter de nueva línea o los tabuladores, principalmente) antes de mostrar el texto bajo edición en pantalla. De esta forma el usuario puede “ver” las líneas o el sangrado.

### 1.2.1.2 Array bidimensional de caracteres

Este modelo presenta el documento bajo edición en la forma de una tabla bidimensional de caracteres. Normalmente el origen de coordenadas de la tabla está en la esquina superior izquierda, aumentando las coordenadas de izquierda a derecha y de arriba a abajo.

En teoría la movilidad del usuario es total en todo el plano, sin estar restringida en ninguna de las dos coordenadas: la extensión del plano es infinita. El concepto de línea está implícitamente soportado, correspondiéndose con la “porción útil” de una fila de la tabla. Puede resultar complejo determinar cual es esa porción útil, ya que es habitual “rellenar” las partes no utilizadas de la tabla mediante caracteres blancos o neutros en la visualización.

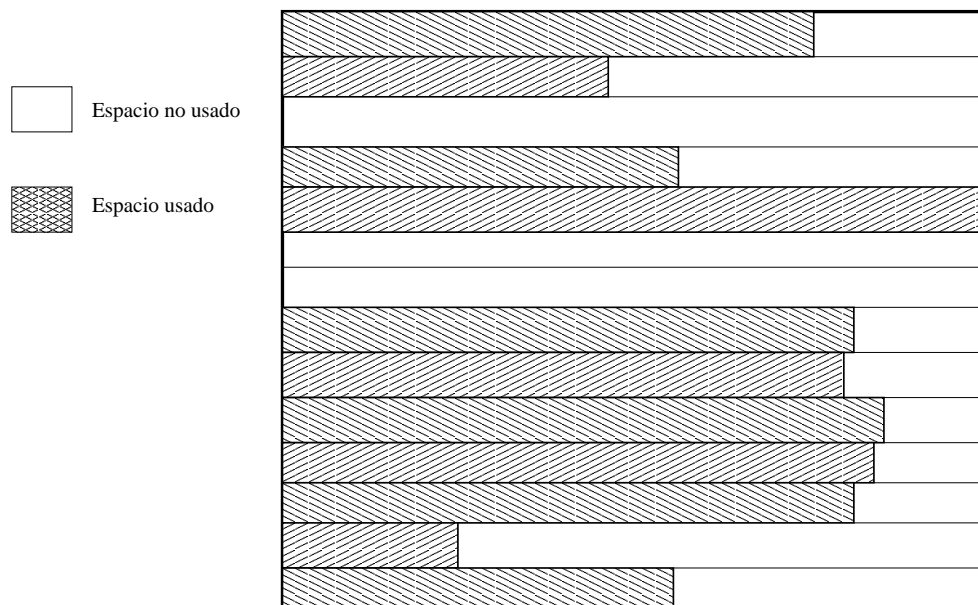


Figura 1.2: Array bidimensional de caracteres

Estos modelos suelen incluir un conjunto de operaciones de edición rico en acciones sobre filas y columnas, y edición de secciones rectangulares de texto en general.

El modelo bidireccional no suele ser implementado en la práctica debido a los costes en memoria y la (a veces) compleja lógica requerida para mantener un plano infinito en máquinas de memoria finita. No obstante es un modelo muy interesante desde un punto de vista conceptual.

### 1.2.1.3 Lista de líneas

Este modelo puede considerarse como una mezcla entre los dos primeros: se compone de un array unidimensional de líneas. Cada línea, por su parte, se implementa como un array unidimensional de caracteres.

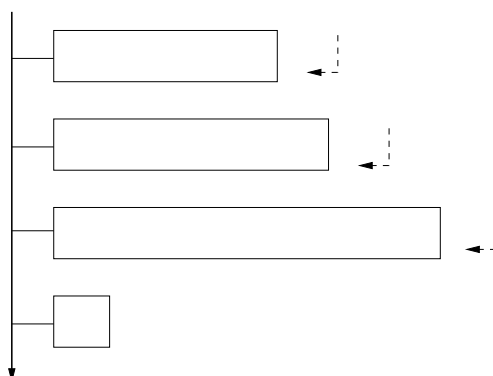


Figura 1.3: Lista de líneas

La diferencia con el array bidimensional de caracteres mas evidente, desde el punto de vista del usuario, consiste en que en este modelo solo existe el texto que ha sido explícitamente insertado (no hay “relleno con blancos”).

Este modelo es con mucho el mas utilizado por los editores existentes.

#### 1.2.1.4 Modelos paginados

Hubo un tiempo en el que era muy popular limitar las actividades de edición a “páginas”. Era necesario utilizar comandos explícitos para moverse entre páginas o repaginar texto.

Cualquiera de los modelos vistos hasta ahora podían utilizarse para editar el contenido de cada una de las páginas.

Esta división en dos niveles de la estructura del documento (páginas y líneas enlazadas, por ejemplo) resultaba muy natural y conveniente, dado que facilitaba la implementación de los editores en las máquinas de entonces, tremendamente limitadas en memoria.

#### 1.2.1.5 Modelos estructurados

Los modelos vistos hasta ahora se centran en la edición de texto lógicamente estructurado en líneas. Sin embargo es muy común dotar de mas estructura a los documentos: un libro dividido en capítulos, secciones, subsecciones, párrafos, frases y palabras; un programa compuesto de paquetes, funciones, procedimientos, parte declarativa; etc.

*editor, de estructura* Algunos editores, llamados *de estructura*, soportan estas nociones de forma directa, incorporándolas al modelo de edición que exportan al usuario.

Los modelos de edición estructurados suelen proporcionar operaciones de edición adecuadas para manipular las estructuras de forma directa y eficiente.

### 1.2.2 Interfaz del usuario

#### 1.2.2.1 Dispositivos de entrada

Desde el punto de vista de un editor los dispositivos de entrada se utilizan para realizar tres funciones principales: introducir elementos (caracteres), introducir comandos (acciones) y designar elementos editables.

Podemos clasificar estos dispositivos en tres categorías:

- Dispositivos de *texto*
- Dispositivos de *estado*
- Dispositivos *localizadores*

Los **dispositivos de texto** consisten típicamente en teclados alfanuméricos en los que el usuario pulsa y libera las teclas, generando códigos únicos que son enviados a la CPU via un controlador de entrada-salida. Existen varias formas de disponer las teclas en estos dispositivos. El mapeo *QWERTY* es el mas común, pese a que existen alternativas como la disposición *Dvorak*, que presenta ciertas ventajas ergonómicas.

Un tipo de teclado alternativo interesante es el llamado *teclado activado por contacto de Montgomery*, en el que, en lugar de presionar teclas, el usuario toca levemente la superficie del teclado con un lapiz o stick.

*dispositivo de estado dispositivo de elección tecla de función*

Los **dispositivos de estado** (tambien llamados **dispositivos de elección**) generan una interrupción cuando se activan, pasando a estado de “encendido”. Es entonces cuando fijan un flag o bandera en el editor, afectando algunas funcionalidades. Usualmente se implementan como *teclas de función* en los teclados, o simulados en la pantalla (el concepto de botón de estado o “checkbox”).

Los **dispositivos localizadores** son dispositivos traductores de coordenadas  $(x, y)$  analógicas a una posición determinada del cursor dentro del editor. Un manejador mapea continuamente pares de valores del dispositivo, mientras que el editor actualiza el puntero o cursor con los nuevos valores traducidos. Estos dispositivos incluyen ratones, joysticks, trackballs, paneles de contacto y tabletas.

### 1.2.2.2 Dispositivos de salida

Los dispositivos de salida permiten al usuario obtener información acerca de los elementos bajo edición y de los resultados de las operaciones.

**TTY** La primera generación de dispositivos de salida (ahora obsoletos) estaba formada por las impresoras y los teletipos (TTYs), que utilizaban papel para plasmar los caracteres. La segunda generación fueron los “teletipos de cristal” (glass teletypes) basados en tubos de rayos catódicos. Estos teletipos utilizaban las pantallas CRT del mismo modo que el papel, aunque algunos modelos innovadores implementaban caracteres como el backspace de forma mas elegante que sus ascendientes en papel.

#### **Pantallas básicas**

Las pantallas CRT modernas conformaron lo que podemos denominar la tercera generación de dispositivos de salida. Estas pantallas disponen de hardware especializado que permite refrescar el contenido de la pantalla (una operación de redisplay) en muy poco tiempo. Operaciones como mover el cursor, insertar y borrar caracteres y borrar líneas estan implementadas directamente en el hardware. Estos dispositivos todavía están orientados a caracteres, no obstante.

### Pantallas mapeadas en memoria

Casi todas las pantallas estaban conectadas a líneas serie de comunicaciones con velocidades típicas de 300, 1200 y 9600 baudios. Por ello la velocidad de actualización del contenido de la pantalla resultaba una operación crítica.

Las bajas velocidades en la transmisión de datos a las pantallas se traducían en dos consecuencias en el diseño y la implementación de los editores de texto:

- El mantenimiento en tiempo real del contenido del texto bajo edición en toda la pantalla podía ser prohibitivo. Por esa razón los editores que trabajaban con estas pantallas usualmente solo efectuaban labores de redisplay (actualización del contenido de la pantalla con el contenido bajo edición) bajo demanda y no en tiempo real.
- Los algoritmos de redisplay eran muy sofisticados y complejos con el objetivo de optimizar al máximo las operaciones de reescritura en la pantalla.

Las nuevas pantallas mapeadas en memoria cambiaron este esquema. Estas pantallas no están conectadas via canales de comunicaciones serie, sino por buses de alta velocidad (ISA, EISA, PCI, etc). Las operaciones de redisplay *memoria principal-memoria de vídeo* son prácticamente instantáneas. El impacto que supusieron estas pantallas en el diseño de los editores de texto es muy grande, ya que los algoritmos de redisplay fueron radicalmente simplificados.

### Pantallas gráficas

La última generación de dispositivos de salida lo tenemos en las pantallas orientadas a gráficos de las estaciones de trabajo modernas. Estas pantallas abandonan la orientación a caracteres para ofrecer posibilidades como mostrar fotografías y otros gráficos, y fuentes de tamaño variable. Estas capacidades permiten al usuario observar (y trabajar sobre) documentos muy similares a como se imprimirán ya tipografiados.

Los primeros editores de texto estaban preparados para trabajar sobre impresoras y teletipos. La mayoría de ellos fueron posteriormente adaptados para trabajar con pantallas. Los editores modernos, por su parte, hacen un uso intensivo de las capacidades de las pantallas modernas.

#### 1.2.2.3 El lenguaje de interacción

Como cualquier otro lenguaje, el lenguaje de interacción de un editor de texto con el usuario puede dividirse en tres partes: un **componente semántico**, un **componente sintáctico** y un **componente léxico**.

El componente semántico del lenguaje especifica su funcionalidad: las distintas construcciones del lenguaje implican determinadas respuestas en el editor (borrar una línea, insertar un caracter, etc).

El componente sintáctico, por su parte, determina la forma de las construcciones del lenguaje: la operación de *borrar región* podría requerir dos argumentos enteros que determinan la región. Teclear dos veces seguidas la tecla *d* pudiera ser una secuencia de comando válida que ejecutaría una acción, pero no teclearla una vez, etc. La sintaxis de este lenguaje debe ser fácil de aprender por parte del usuario y debe ser coherente y natural al modelo conceptual implementado por el editor.

El componente léxico especifica como pueden combinarse los *lexemas* generados por el dispositivo de entrada (o salida) para formar los tokens utilizados por el componente sintáctico.

Podemos clasificar los distintos tipos de lenguajes de interacción encontrados en editores de texto en base a la forma que tienen de generar los lexemas:

### Interfaz orientada a texto

Esta es la forma de interacción con el usuario mas antigua utilizada por editores de texto.

Consiste en un intérprete que va leyendo comandos del usuario en forma de un lenguaje escrito. El usuario debe expresar de forma escrita tanto el nombre de los comandos como sus argumentos. El editor puede procesar un comando cada vez o un número variado de ellos. En el segundo caso el usuario debe indicar explícitamente que desea ejecutar los comandos ya introducidos.

Este tipo de interfaces presentan una serie de desventajas: el usuario debe recordar el lenguaje con detalle (características léxicas y sintácticas). Además el teclear los comandos puede resultar lento, sobre todo para los principiantes. La gran ventaja de las interfaces orientadas a texto es, como era de esperar, una gran flexibilidad y potencia expresiva.

Veamos algunos ejemplos de este tipo de interfaz implementada en editores reales. El comando para sustituir cualquier aparición de 'a' por 'b' en la línea diez con el editor **ed** es el siguiente:

```
10s/a/b/g
```

por su parte, el editor **vi** utilizaría el siguiente comando para escribir el buffer actual en su fichero y salir del editor:

```
wq!
```

Por último, esta sería la instrucción **Emacs** para avanzar el cursor hasta la primera aparición de la palabra "Ridicule":

```
(search-forward 'Ridicule')
```

### Interfaz orientada a teclas

La dificultad de recordar los detalles de los lenguajes de interacción hacían que la utilización de los editores resultara difícil para ciertas clases de usuarios. Ciertamente la curva de aprendizaje podía ser bastante pronunciada, en especial para usuarios que nunca hubieran utilizado este tipo de interfaz para comunicarse con otro programa.

*tecla modificadora*

Esto llevó a la invención de otro tipo de interfaz, que asocia la pulsación de una tecla con la ejecución de una rutina semántica. Dado que los teclados contienen una cantidad limitada de teclas, es muy comun utilizar teclas *modificadoras* como la tecla Control y la tecla Meta, presentes en muchos teclados. De esta forma pueden encadenarse pulsaciones de teclas para formar lexemas del lenguaje de interacción.

Existen estudios (aunque no muchos y no muy extensos) que discuten la forma de asociar secuencias de teclas a rutinas semánticas, en base a ciertos criterios (longitud de la secuencia de teclas, como de frecuente es la utilización de la

rutina semántica, etc). Muchas de las conclusiones son evidentes: asociar a rutinas semánticas muy utilizadas secuencias de teclas cortas o asociar secuencias con igual prefijo a rutinas semánticas de propósito similar (esto permite construir reglas fáciles de recordar como: “las operaciones sobre líneas comienzan con C-l”).

Íntimamente ligado a la interfaz orientada a teclas está el concepto de *modo operacional modal*. Un editor de texto opera de forma modal cuando en un momento dado puede estar en uno de varios *modos de operación*. Muchas veces se utilizan estos modos de operación para tener un *modo comando*, donde todas y cada una de las teclas del teclado están asociadas a rutinas semánticas arbitrarias, y un *modo edición*, donde las teclas alfanuméricas están asociadas a rutinas semánticas que insertan el carácter asignado a cada tecla. **vi** es un ejemplo de editor modal. **Emacs** es un ejemplo de editor no modal, donde se utilizan teclas modificadoras para comenzar secuencias de teclas.

### Interfaz orientada a menús

Con el advenimiento de las interfaces gráficas se hizo posible otro tipo de interfaz para lanzar la ejecución de las rutinas semánticas del editor: los menús y otros componentes gráficos.

Un menú es un conjunto de entradas agrupadas en la pantalla una detrás de la otra. Cuando el usuario selecciona una entrada (mediante la utilización de algún dispositivo localizador) se ejecuta la rutina semántica asociada.

Muchas veces los menús pueden anidarse, apareciendo así el concepto de “submenú”. Gracias al anidamiento de menús pueden agruparse y jerarquizarse las entradas en base a las características de sus rutinas semánticas (todas las entradas que afectan a líneas en un submenú, por ejemplo).

Los tipos de interfaz descritos no tienen por qué presentarse de forma pura: muchos editores combinan varios tipos de interfaz para satisfacer a distintos tipos de usuarios (mas o menos experimentados) o para efectuar distintos tipos de tareas (mas o menos complejas, o repetitivas). Por ejemplo, el editor **vi**

A pesar de que los componentes semánticos de los lenguajes de interacción de editores distintos suelen ser similares (suelen implementarse similares rutinas semánticas) esto no se cumple para los componentes sintácticos o léxicos. Por ejemplo, pensemos en las rutinas semánticas para borrar la palabra “picola”. Un editor con interfaz de texto podría requerir introducir un comando como

```
s/picola/d
```

que puede traducirse a lenguaje natural como “Busca la siguiente aparición de *picola* y bórrala”. En un editor con interfaz de teclas pueden utilizarse las teclas cursor (asociadas a rutinas semánticas de movimiento del puntero de edición) para posicionar el puntero sobre la palabra a borrar, y a continuación pulsar las teclas **dw**. En definitiva, léxicos y sintáxis muy distintas que provocan la misma respuesta semántica en los editores.

Nos queda por destacar una propiedad muy interesante de los lenguajes de interacción con el usuario: pueden contemplarse como una arquitectura de capas. Varias semánticas pueden reflejarse en varias sintáxis, que a su vez pueden reflejarse en varios léxicos distintos. Por ejemplo, dadas las rutinas semánticas para insertar o borrar caracteres en una secuencia,

los implementadores pueden proporcionar sintáxis prefija o postfija para representar dichas semánticas. Por su parte, dada una sintáxis (sea prefija o sufija) los implementadores pueden proporcionar entradas orientadas a texto, a teclas o a menús para formar los lexemas. Esta arquitectura en capas facilita independencia de dispositivos e independencia sintáctica respecto a las semánticas que realmente implementa el editor.

### 1.3 Taxonomía de los editores de texto

En función de las características de los usuarios, las necesidades cubiertas, el hardware disponible y el estado del arte, se han desarrollado muchos tipos de editores de texto a lo largo del tiempo.

Es difícil dividir en categorías disjuntas los cientos (¿tal vez miles?) de editores que se han construido. Muy a menudo comparten ciertas características y difieren en otras. La mejor forma de catalogar un editor de texto es en base a dichas características, de las cuales exponemos aquí un resumen.

#### Qué editan...

Pese a que el objeto básico bajo edición de cualquier editor de textos es información textual (compuesta de una secuencia de caracteres) muchos editores asumen una estructuración adicional. Los motivos para ello son variados, ya que si bien un editor de estructuras añade estructuración adicional para facilitar la manipulación abstracta del texto, los editores de líneas suelen hacerlo para facilitar cuestiones de implementación.

#### Orientados a líneas

Los editores orientados a líneas dividen la información bajo edición en líneas por motivos de eficiencia y capacidad en implementación. Un documento, compuesto de varias líneas, se almacena en memoria secundaria y solo es trasladado a un buffer en memoria principal (ya en los dominios del editor) una línea por vez. Así aparece el concepto de *línea actual*.

La estructuración impuesta por estos editores hace que las operaciones de edición (búsqueda, insertar información, borrar información, etc) estén restringidas a la línea actual, exceptuando las primitivas que almacenan la línea actual en memoria secundaria (*store*) y que cargan una línea desde memoria secundaria (*load*).

Al requerir de las operaciones explícitas de carga, podemos denominar a esta estructuración *fuerte* o *impositiva*.

Pese a que ya no forman parte del mundo de los programas de consumo, estos editores se siguen utilizando para realizar labores de edición no interactivas, pese a que para dichas labores están siendo sustituidos por los *stream editors*.

#### Orientados a páginas

Del mismo modo que los editores de líneas estructuran el texto bajo edición en líneas, algunos editores hacen lo propio con las *páginas*.

La definición de página en estos casos a menudo difiere de nuestra concepción de “página” presente en textos tipografiados. Por

ejemplo, en el editor **TECO**, una página está definida como el texto contenido entre dos caracteres  $\sim L$ .

Esta estructuración es fuerte, requiriendo por tanto de instrucciones explícitas de carga. Es de notar que estos editores (en particular TECO) precedieron en el tiempo a los editores de líneas.

Nadie utiliza ya editores orientados a páginas ya que, una vez desaparecidas las restricciones impuestas por limitaciones hardware, el particionado en páginas parece arbitrario y poco útil.

### **Stream editors**

Los editores que contemplan todo el texto bajo edición como una secuencia y por tanto no presentan ninguna estructuración fuerte que requiera instrucciones explícitas de carga son conocidos como *stream editors*.

Esto es independiente del ámbito de las operaciones de edición, que pueden estar restringidas a líneas, páginas o párrafos.

Como ya se ha mencionado en el apartado de los editores de líneas, estos editores se utilizan de forma generalizada para llevar a cabo trabajos de edición no interactivos.

### **De estructura**

Algunos editores de propósito específico (tales como orientados a escribir programas en algún lenguaje de programación, o a editar secuencias de ADN) llevan a cabo una tarea intensiva de estructuración del contenido textual bajo edición.

En este tipo de editores las operaciones de edición están íntimamente relacionadas con los elementos que componen esta estructura (funciones, declaraciones de variables, números, expresiones, etc) y permiten al usuario expresarse en términos muy abstractos.

Estos editores no son muy comunes y la mayoría son experimentales (“de laboratorio”). Su diseño e implementación requiere de tareas de investigación además de las propias de desarrollo, ya que el manejo en tiempo real de árboles y otras representaciones de sintáxis o semántica es un campo todavía en pleno desarrollo.

### **Como interactúan con el usuario...**

Las formas de comunicación con el usuario varían de editor en editor. La clasificación más general a este respecto es la de editores interactivos y no interactivos.

#### **Batch**

*editor batch*

Los editores de texto no interactivos, o editores *batch*, utilizan una interacción con el usuario propia de los procesos secuenciales: el usuario introduce una serie de operaciones junto con los datos a procesar, el editor los procesa y ofrece al usuario el texto editado.

Estos editores pueden utilizar varios métodos para presentar los resultados al usuario. Si bien pueden utilizar una pantalla, la naturaleza no interactiva del proceso hace más común la utilización de dispositivos como impresoras o ficheros.

De forma independiente al texto bajo edición, estos editores suelen dejar un registro de las operaciones que realizan (notablemente incluyendo informes de errores) en dispositivos alternativos.

Una consecuencia importante de este tipo de operación consiste en que la capacidad de corrección en la edición es nula mientras el proceso está en marcha, siendo necesaria otra operación completa de edición para llevarlos a cabo. Por ello las operaciones especificadas al editor suelen estar generadas por otros programas y no directamente por el usuario.

### **Interactivos**

En contraposición a los editores no interactivos, estos editores están equipados con funcionalidades y dispositivos que permiten una comunicación precisa y continuada con el usuario. Las operaciones de edición son introducidas a todo lo largo del proceso, y por tanto la capacidad de reflexión y de corrección está presente en todo momento.

Además, el usuario es capaz de cambiar el contenido bajo edición en cualquier momento, pudiendo cargar ficheros o líneas a voluntad.

## **1.4 Descomposición conceptual de un editor**

Como hemos visto la taxonomía de los editores de texto es amplia. En función de características como el objeto bajo edición, la máquina en la que se ejecuta el editor o los usuarios que lo utilizan existen infinitas clases de editores, cada una de ellas con sus peculiaridades en cuanto a funcionamiento y manejo de la información.

Por eso conviene en este punto un ejercicio de abstracción y formular un modelo conceptual que englobe cualquier proceso de edición llevado a cabo por un programa (esto es, un editor). De esta forma estaremos en condiciones de pasar al siguiente estadio: formular un modelo funcional que nos permita implementar dichos programas.

Basándonos en el trabajo de Meyrowitz y Andries van Dam podemos ofrecer una visión conceptual de un editor de texto, o dicho de otra forma, del proceso general de edición. Véase Figura 1.4.

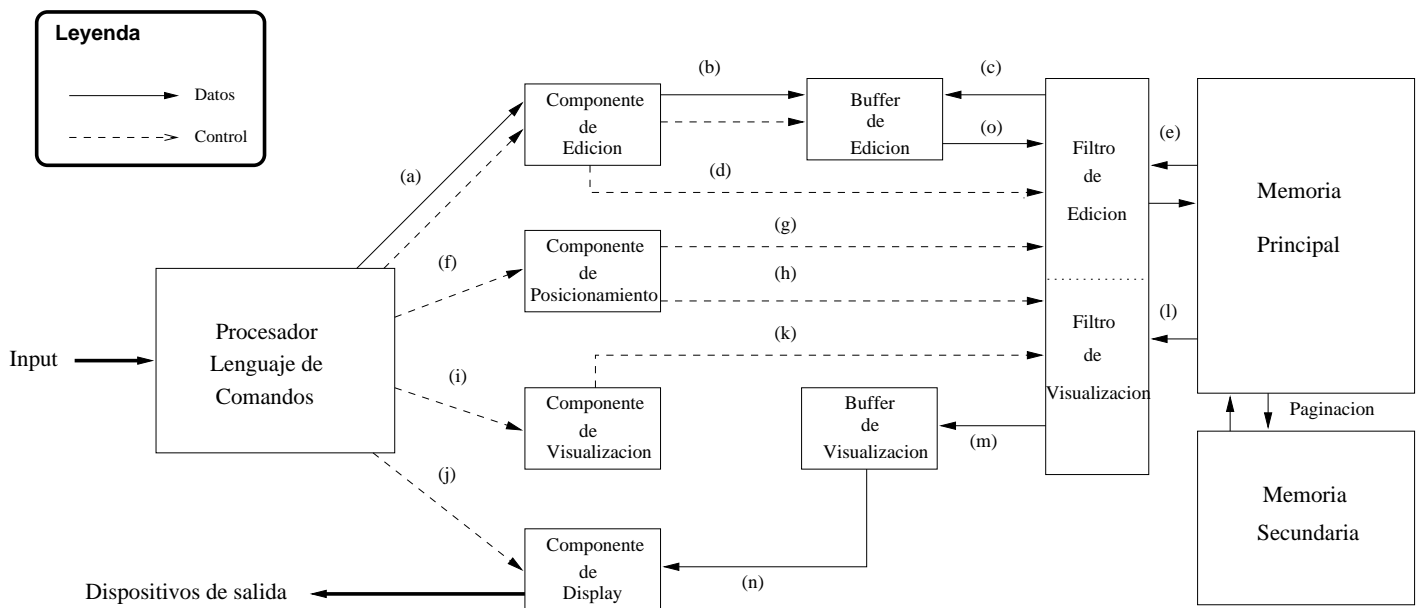


Figura 1.4: Descomposición conceptual de un editor

Nótese que en este esquema las cajas representan *entidades conceptuales*. Esto no determina absolutamente ningún aspecto concreto de implementación: varias de las cajas pueden corresponder a un solo módulo funcional, y una sola caja puede corresponder a varios módulos funcionales.

### 1.4.1 El proceso de edición

Consideremos el proceso que realiza una persona cuando se dispone a editar parte de un texto. Supongamos que para la tarea ha escogido un editor de texto interactivo orientado a pantalla.

En primer lugar hay que localizar la porción del texto que se desea cambiar. Por ejemplo, si nuestro objetivo es cambiar la primera palabra del segundo párrafo de la página diez.

A continuación se debe *viajar* hacia dicha posición o área. En la mayoría de editores de texto esto se consigue moviendo el *puntero de edición* a la posición deseada, bien mediante el teclado o utilizando otro dispositivo de señalización. Finalmente se invocan las operaciones de edición (por ejemplo, sustituir una palabra por otra).

Luego parece que podemos dividir cualquier proceso de edición en tres fases bien diferenciadas:

1. **Localizar** el área del texto donde se va a realizar las operaciones de edición.
2. **Viajar** a dicha área.
3. **Editar** el texto.

Este esquema es recursivo: consideremos el proceso de edición (de alto nivel) “Cambiar la palabra **valla** por **vaya**”. Esto requerirá *viajar* hacia la palabra *valla* y *editarla* para cambiar la **ll** por **y**. Pero esto último consiste en un proceso de edición (de menor nivel

de abstracción) consistente en *viajar* hacia el tercer caracter de la palabra **valla** y sustituir (*editar*) ambas letras por **y**. Y este último proceso de edición también se descompone en un *viaje* a la segunda **l**, un borrado (*edición*), otro *viaje* a la primera **l**, y sustituirla (*edición*) por la **y**. Llega un momento en el que no hay mas descomposición: las operaciones de edición *borrar caracter* y *sustituir caracter* son primitivas y detienen el proceso recursivo.

Esta naturaleza abstracta del proceso de edición nos permite aplicar el mismo esquema sea cual sea el objeto bajo edición: líneas, párrafos, imágenes o caracteres.

### 1.4.1.1 El procesador del lenguaje de comandos

Este componente acepta directivas de los dispositivos de entrada del usuario del editor en forma de un lenguaje de comandos. Una vez analizada léxica y sintácticamente una sentencia en dicho lenguaje de invoca a las rutinas semánticas correspondientes.

Las rutinas semánticas pueden estar escritas en el lenguaje de implementación del editor, o bien en otro lenguaje de mayor nivel.

Ejemplos:

```
:w!q
        write_buffer_contents (FORCE_P);
        quit ();

<Control>-f
        prompt_search_subject ();
        search_forward ();
```

En lugar de invocar directamente a las rutinas semánticas, es posible generar una sentencia en un lenguaje intermedio (representación intermedia) cuyo intérprete es el que invoca a las rutinas. De esta forma se independiza el grupo de rutinas semánticas de la sintáxis de un lenguaje de comandos.

Caben distinguir dos tipos de lenguajes de comandos:

- Lenguajes implícitos (tecla -> invocación a rutina nativa del editor)
- Lenguajes explícitos (tecla -> rutina, prompt)

Una composición legal de tokens (es decir, una sentencia del lenguaje) provoca la invocación de las rutinas semánticas correspondientes. A su vez dichas rutinas semánticas invocan operaciones de desplazamiento, edición, visualización y display.

### 1.4.1.2 El componente de edición

Este componente consiste en una colección de módulos que gestionan las operaciones de edición. Esto incluye el mantenimiento del puntero y las marcas, y por tanto es el encargado de determinar la localización de las operaciones de edición.

Cuando el usuario invoca una operación de edición, el componente de edición invoca al filtro de edición. El buffer en memoria que contiene el documento es entonces filtrado para generar un nuevo *buffer de edición* basado por una parte en la posición actual del puntero, y por otra en los *parámetros de filtrado* ya activos en el filtro.

La configuración del filtro de edición puede ser fijada tanto por el componente de edición como por el componente de posicionamiento. En el primer caso puede asumirse que los

parámetros del filtro son fijados de forma *implícita* por el sistema. Es el caso de una operación de edición que requiere de la utilización de varios buffers de edición, por ejemplo. En el segundo caso se trata de respuestas a operaciones de posicionamiento invocadas directamente por el usuario. Tal es el caso si el usuario ejecuta una operación de carga de línea en un editor orientado a líneas, por ejemplo.

Cuando termina la operación de edición el contenido del buffer de edición es filtrado de nuevo (pero en sentido inverso) y almacenado en memoria principal.

### 1.4.2 El proceso de visualización

En los editores interactivos se realiza una tarea constante de presentación de información al usuario mediante un dispositivo de salida tal como una pantalla o una impresora.

Por lo general dichos dispositivos de salida presentan al usuario una *vista sesgada* del texto bajo edición.

Por ejemplo, supongamos que estamos utilizando una pantalla orientada a texto capaz de mostrar 80 líneas compuestas de 25 caracteres cada una<sup>1</sup>. El *componente de display* es el encargado de proporcionar a la pantalla el contenido del *buffer de visualización*, que a su vez es mantenido por el *filtro de visualización*. Es este filtro el que gestiona qué porción del texto bajo edición es visualizado en el dispositivo.

El filtro de visualización es actualizado por operaciones explícitas por parte del usuario, contenidas en el componente de posicionamiento (mover el puntero a la siguiente línea cuando ésta es la última de la pantalla, por ejemplo) y en el componente de visualización (operaciones de scroll, por ejemplo).

Es de notar que nuestro esquema conceptual permite que el usuario invoque directamente al componente de display. Esto asume que el editor exporta rutinas semánticas para controlar la forma en que se actualiza el dispositivo de salida correspondiente con el contenido del buffer de visualización. Por ejemplo, muchos editores que funcionan en plataformas Unix proporcionan el comando **Ctrl-L** para forzar una fase de refresco de pantalla.

## 1.5 Descomposición funcional de un editor

Pese a que cualquier editor puede ser especificado en los términos conceptuales expuestos en el apartado anterior, la implementación de dichos conceptos puede variar dependiendo de muchos factores. Por ejemplo, podría plantearse la necesidad de mantener buffers en memoria separados para mantener el buffer de edición y el documento en memoria principal.

---

<sup>1</sup> Este número es mucho mayor en el caso de las pantallas gráficas, pero es irrelevante a efectos de esta explicación.

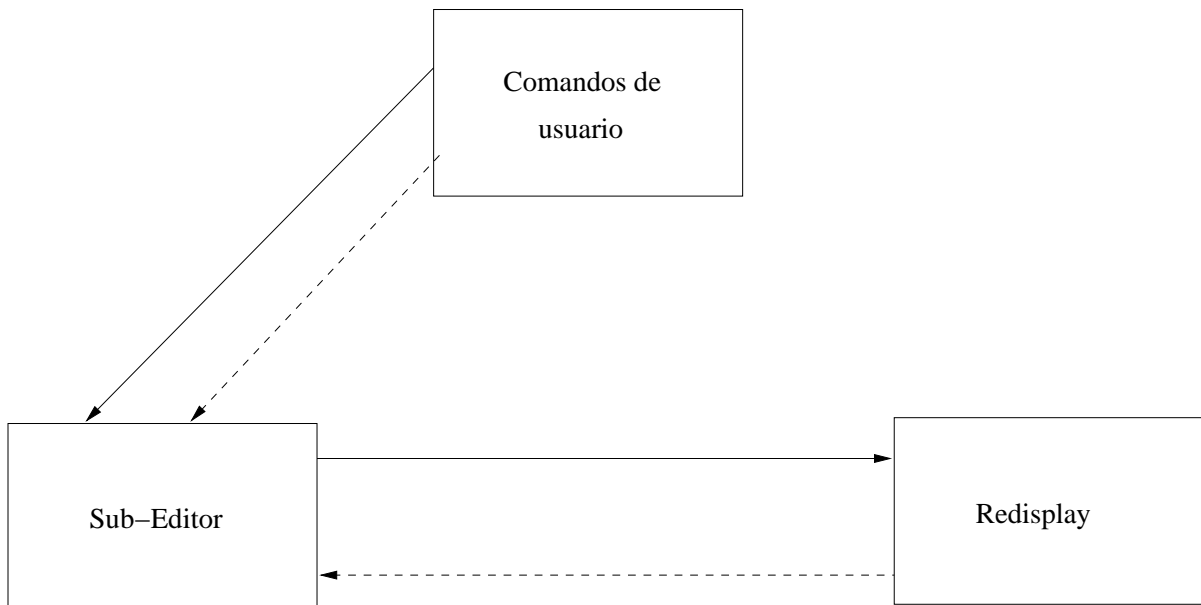


Figura 1.5: Descomposición funcional de un editor

En este apartado se presenta la descomposición **funcional** propuesta por Finseth para la construcción de editores de texto. Se trata de un esquema **Modelo-Vista-Controlador** que ha sido implementado con éxito en infinidad de editores de muy diferentes características. De hecho, puede considerarse la *descomposición de Finseth* como el “modelo de referencia” en la construcción de editores de texto. La mayoría de las alternativas consisten en soluciones *ad-hoc* poco estructuradas.

A continuación se revisa cada uno de los tres componentes funcionales del esquema y su relación.

### El sub-editor

Puede afirmarse que el sub-editor es el corazón de cualquier editor de textos implementado mediante la descomposición de Finseth. La eficiencia y la usabilidad del editor descansan sobre todo en este componente. Sin embargo, se trata de la parte menos visible por parte del usuario.

El cometido del sub-editor consiste en ocultar los detalles del almacenamiento del texto en memoria así como su estructuración interna. De este modo los otros dos componentes (comandos de usuario y refresco de pantalla) pueden abstraerse de los complejos mecanismos de almacenamiento.

La estructura de datos principal mantenida por el sub-editor es el *buffer de texto*. Un *buffer* es la unidad básica de información disponible para edición. Puede ser de cualquier tamaño, desde cero octetos hasta la máxima región de memoria soportada por la máquina.

El sub-editor debe proporcionar una interfaz para manipular tantos objetos *buffer* como se requiera, incluyendo operaciones de actualización de información

en un buffer (insertar o borrar texto) y operaciones de gestión de buffers a mas alto nivel (creación y destrucción de buffers, etc).

El sub-editor implementa las siguientes entidades conceptuales: las memorias principal y secundaria, los componentes de edición y localización, el buffer de edición y el filtro de edición.

### **El refresco de pantalla, o redisplay**

El refresco de pantalla o redisplay implementa las entidades conceptuales siguientes: filtro de visualización, buffer de visualización, componente de visualización y el componente de display.

Su cometido principal consiste en mantener actualizada una vista del contenido de uno o varios buffers (muchos editores permiten utilizar varias ventanas simultáneamente) en uno o varios dispositivos de salida, tales como pantallas o impresoras.

En general existen dos formas de interacción de este módulo con los dos restantes:

#### **Redisplay guiado por los comandos de usuario**

En este esquema (distinto al de Finseth) las acciones de refresco de pantalla se implementan como respuestas directas de los comandos de edición.

Por ejemplo, supongamos que el módulo de comandos de usuario recibe la orden de introducir un caracter en una posición determinada del buffer bajo edición. Deberá generar instrucciones tanto para el sub-editor (introducir el carácter en el buffer) como para el refresco de pantalla (hacer visible la modificación en el dispositivo de salida correspondiente).

Este esquema no es muy recomendable por los siguientes motivos:

- Genera una gran dependencia entre el sistema de redisplay y el módulo de comandos de usuario.
- Suele llevar a implementaciones ad-hoc difíciles de depurar y poco o nada escalables.
- Es muy propenso a errores e inconsistencias en la sincronización *buffer-vista*.

Por ello es preferente el siguiente esquema.

#### **Redisplay independiente**

Este esquema es el recomendado por Finseth y el mostrado en la figura Figura 1.5. El proceso de refresco de pantalla es totalmente independiente de los comandos de usuario y su relación con los demas módulos se limita a obtener información del estado actual del buffer.

Habitualmente el proceso de redisplay se activa periódicamente varias veces por segundo, y su operativa consiste en obtener la información necesaria del sub-editor acerca del estado actual del buffer para actualizar la información que pueda estar mostrándose en pantalla.

Esta arquitectura es mucho mas clara y menos propensa a fallos que la anterior, si bien requiere de algoritmos mas sofisticados y complejos.

### **Interacción con el usuario: comandos**

Como ya hemos visto en este mismo capítulo, la interacción de un editor con el usuario compone la única visión que tiene éste del sistema. Por tanto el “sabor” o impresión que cause el editor en el usuario viene determinado por las funcionalidades implementadas en este módulo.

El lenguaje de interacción y el acceso a las rutinas semánticas exportadas por las otras partes del editor son partes componentes de este módulo.

## **1.6 Origen y evolución de algunos editores**

### **1.6.1 Tape/Text Editor and Corrector: TECO**

TECO es, junto con EMACS, uno de los editores mas importantes en la historia de la edición de texto. De aparición temprana (el primer TECO fué desarrollado para el PDP-1 de Digital) representó un avance brutal en la edición.

Dan Murphy, estudiante en el MIT y posteriormente empleado de DIGITAL, hastiado de la edición orientada a líneas de los editores existentes por entonces, desarrolló la primera versión del *Tape Editor and Corrector* para funcionar en el DEC PDP-1 en 1964 o 1965. Una característica muy interesante de este primer TECO es que editaba una página de texto en memoria. Segun el propio Murphy nunca hubo versiones de TECO que editaran una sola línea.

El TECO PDP-1 utilizaba *FIO-DEC* para codificar los caracteres ,como otros programas sobre el PDP-1. El FIO-DEC era una codificación de caracteres de seis bits, incluyendo caracteres **upshift** y **downshift**. Además tenía un código de control denominado **stop code** (o código de parada) que provocaba que el lector de la cinta de papel se parara. Este carácter de control fué utilizado como la marca *fin de página* en TECO. Es de notar que este **stop code** no pertenecía al buffer bajo edición. Esto quiere decir que no era editable por el usuario, y por tanto era posible concatenar varias páginas de forma transparente. Además, cuando se sacaba por una impresora varias páginas de texto editadas con TECO se incluía este carácter, provocando la parada de la impresora entre página y página.

En este primer TECO el tamaño del buffer (de la página en edición) estaba limitada a la capacidad de la memoria del PDP-1 menos lo ocupado por el propio editor. Esto dejaba unos 6000 caracteres en el PDP-1 de 4k-palabras original. Suficiente para editar típicas páginas de código o de texto, pero definitivamente no para editar un fichero entero.

Mas tarde TECO fue portado para funcionar en una PDP-6 (máquina mucho mas capaz que su antecesora PDP-1) y modificado para utilizar ASCII en lugar de FIO-DEC. En esta nueva versión se incluyó el carácter “formfeed” como perteneciente al buffer.

### **1.6.2 QED**

El editor de texto *QED* fue escrito por Butler Lampson y Peter Deutsch en 1967. Funcionaba en el SDS 940, el sistema de tiempo compartido de Berkeley. Este editor estaba orientado a caracteres y permitía utilizar varios buffers para editar mas de un fichero de forma simultánea. Era posible copiar y mover texto de un buffer a otro e incluso ejecutar

programas escritos en el lenguaje de QED almacenados en buffers. De esta forma permitía una gran extensibilidad. Es probable que Lampson y Deutsch se basaran en el TECO de Murphy para adoptar este paradigma, aunque Dennis Ritchie afirma que fue elaborado de forma independiente por ambos editores.

Ken Thompson utilizó el QED de Berkeley antes de instalarse en los Laboratorios Bell. De hecho, lo primero que hizo al llegar a los Bell fue escribir una nueva versión de QED en ensamblador de IBM 7090 para funcionar en los sistemas de tiempo compartido CTSS. Esta nueva versión era similar a la de Berkeley exceptuando sus excepcionales funciones de búsqueda. Thompson introdujo la utilización de expresiones regulares como patrones de búsqueda. Inventó un algoritmo para compilar código máquina nativo partiendo de una expresión regular que creaba un automáta finito no determinista para efectuar la búsqueda. Hasta este momento las funciones de búsqueda de los editores de texto (incluyendo TECO y sus derivados) se basaban exclusivamente en patrones fijos.

La arquitectura de tiempo compartido CTSS de los Bell se utilizó como parte del proyecto MULTICS, y Thompson escribió otra versión de QED para dicho sistema. Esta nueva versión estaba escrita en BCPL y como novedad ya no compilaba las expresiones regulares, sino que las interpretaba directamente en forma de árboles para simular el autómata finito.

En este momento Dennis Ritchie llegó a los laboratorios Bell y escribió una nueva versión del CTSS QED de Thompson para GE-TSS (el sistema de tiempo compartido para GECOS), esta vez en lenguaje ensamblador. Es de notar que esta versión seguía compilando las expresiones regulares a lenguaje máquina y no las interpretaba directamente como Multics QED.

La implementación GECOS QED tal vez fue la mas barroca y compleja de todas las implementaciones de QED, especialmente al utilizar las expresiones regulares hasta un nivel al que Kleene probablemente nunca soñó: las expresiones regulares se ampliaron para soportar *referencias inversas*, y por tanto era posible denotar algunos lenguajes de contexto libre. Por un tiempo el GECOS QED fue empleado como lenguaje de scripting de forma similar a como se utilizan hoy en día AWK o Perl. Una gran colección de macros útiles se mantenía en un sitio accesible a todo el que quisiera utilizarla o ampliarla.

En 1974 Jay Michlin escribió otra versión mas del editor para funcionar en los sistemas IBM TSO, y fue utilizado intensivamente en los Laboratorios Bell junto con el hardware y el software de IBM.

### 1.6.3 El editor estándar de Unix: ED

El editor estándar de Unix, *ed*, fue escrito por Ken Thompson para su Unix y el PDP-7. Descendiente directo de QED, esta primera implementación era mucho mas simple. Aunque conservaba la orientación a líneas las expresiones regulares estaban horriblemente mutiladas al solo disponer del metaoperador \* (cierre transitivo-reflexivo de Kleene): sin alternativa, sin agrupamiento con paréntesis. Evidentemente ni siquiera era capaz de denotar cualquier lenguaje regular.

La simplificación también afectó al modelo general de edición del editor: no tenía soporte para varios buffers ni para ejecutar su contenido como programas *ed*.

Afortunadamente, a medida que el propio Unix iba madurando de implementación a implementación, el editor fue incorporando algunas de las poderosas capacidades de GECOS QED, como las referencias inversas en las expresiones regulares.

Finalmente la inevitable versión GNU hizo su aparición, llamada *sed*, y con ella el añadido de muchas más capacidades. Sin embargo, a diferencia de Unix, GNU no considera *ed* como su editor estándar. Ese papel lo ocupa GNU Emacs, del cual hablaremos en un apartado posterior.

### 1.6.4 EMACS

El proyecto EMACS (Editing MACroS, o macros de edición) comenzó en el verano de 1976 en el laboratorio de inteligencia artificial del MIT.

Inicialmente EMACS se escribió en TECO. Sin embargo, no se trataba del TECO que conoce la mayoría de la gente, en realidad una versión muy simplificada adaptada por Bob Clements para el sistema operativo de DEC en el PDP-6.

Como ya se ha comentado en el apartado dedicado a TECO, el programa original fue escrito por Dan Murphy en 1964 y funcionaba en el PDP-1 y fue posteriormente reimplementado en el PDP-6 por Richard Greenblatt, Jack Holloway y Tom Knight. Esta última reescritura se completó en solo una semana. El TECO original utilizaba un área de edición en el que se mostraba el texto que se estaba editando alrededor del puntero o cursor, y por tanto se trataba de un genuino editor orientado a la pantalla. El hecho de que mucha gente utilizara TECO sin disponer de una pantalla asombró y preocupó a los autores originales. Esta versión de TECO (descendiente directo del original) se utilizaba en el sistema operativo del MIT para las PDP-6 y PDP-10: ITS (Incompatible Timesharing System).

En Stanford, por otra parte, se desarrolló una familia de editores orientados a pantalla totalmente distintos a los del MIT. Sin embargo compartían el mismo ancestro: el TECO para el PDP-1 con su área de edición visual. Los editores más importantes de Stanford fueron *TVEDIT* (para el Tenex/TOPS-20, un sistema operativo de DEC) y *E* (para WAITS).

Así estaban las cosas cuando Richard Stallman visitó el laboratorio de inteligencia artificial en Stanford. Inmediatamente se sintió impresionado por las capacidades de edición en tiempo real de *E*. En el ITS TECO que utilizaban en el MIT, pese a ser orientado a pantalla (Stallman había implementado el procesador de pantalla ya en 1974), no existía esta capacidad de edición en tiempo real. El usuario tenía que introducir explícitamente comandos TECO para introducir información al buffer bajo edición.

A raíz de esta visita Stallman decidió que ITS TECO debía tener capacidades de edición en tiempo real, y cuando regresó al MIT se puso manos a la obra. El resultado fue el *Modo  $\hat{R}$*  de ITS TECO. Cuando un usuario ejecutaba el comando CTRL/R se pasaba a “modo en tiempo real” en el cual podía introducir caracteres en el buffer simplemente pulsando las teclas correspondientes a los caracteres. Luego ITS TECO dispuso de una base funcional modal donde se pasaba al modo  $\hat{R}$  para insertar caracteres en el buffer y se salía del mismo al prompt de TECO para ejecutar cualquier otra tarea.

El Modo  $\hat{R}$  introdujo además algunas innovaciones propias, constituyendo un nuevo modelo de edición por derecho propio. Por ejemplo, inauguró el concepto de *inserción* en contraposición al de *sustitución* a la hora de añadir caracteres al buffer. Esto corresponde al modelo aceptado hoy en día, en el que la inserción de un carácter en una posición dada provoca el desplazamiento hacia la derecha de todo el contenido del buffer situado a la derecha de la posición. Los editores orientados a pantalla de tiempo real existentes hasta entonces (*E* y *TVEDIT*) únicamente permitían la sustitución del carácter situado en la

posición de inserción (el mismo efecto que se obtiene en un PC cuando se activa el modo “Insert” con la tecla del mismo nombre<sup>2</sup>).

Pese a todas sus innovaciones el Modo  $\hat{R}$  seguía presentando serios problemas en la evolución hacia los editores orientados a pantalla de tiempo real. Las funciones de búsqueda solo admitían patrones compuestos de un solo carácter, y era necesario salir del modo  $\hat{R}$  para introducir comandos TECO para hacer otras muchas cosas (como leer o escribir ficheros).

Es de notar que, aparte del Modo  $\hat{R}$ , otros dos grandes paquetes de macros que añadían funcionalidad a TECO pululaban por el MIT: TECMAC y TMACS<sup>3</sup>. Mientras que las extensiones implementadas en TECMAC iban orientadas principalmente hacia la edición en tiempo real, en TMACS podía encontrarse funcionalidades muy potentes mas orientadas a la programación en TECO: soporte de comandos con nombres y variables. Cuando un hacker del MIT escribía una extensión en forma de un conjunto de macros lo añadía a alguno de los dos paquetes anteriores, o a ambos.

Stallman comprendió que el camino a seguir consistía en sustituir TECMAC, TMACS, el Modo  $\hat{R}$  y los otros paquetes de macros con un nuevo editor que englobara todas las funcionalidades de una forma coherente y (sobre todo) extensible. Ese fue el comienzo del proyecto EMACS propiamente dicho.

Se tenía muy claro el camino a seguir. Para comenzar había que superar los inconvenientes inherentes al propio lenguaje TECO, que era primitivo y carecía de muchas construcciones de control y estructuración. Esto provocaba que los programas escritos en TECO fueran terriblemente complicados y difíciles de mantener. Para solventar esto se procedió a la construcción de un sistema de bibliotecas que permitieran escribir programas TECO de una forma estructurada y otras facilidades impotantes a la hora del mantenimiento, como auto documentación del código. Posteriormente, utilizando las facilidades de la nueva biblioteca, se implementaron los comandos de edición en pantalla propiamente dichos. El juego de comandos del nuevo editor se diseñó en base a los anteriores editores basados en TECO y de modo que las operaciones mas comunes requirieran pocas pulsaciones de teclas (una o dos).

A finales de 1976 ya se disponía del primer sistema EMACS funcional. Sin embargo solo podía ejecutarse en sistemas que dispusieran del ITS TECO extendido con las nuevas funcionalidades. Dado que ITS TECO estaba escrito en ensamblador para el ITS del MIT no era posible ejecutar EMACS en ninguna otra plataforma y su uso estaba por tanto limitado al propio MIT. Mientras tanto las noticias acerca de las bondades del nuevo editor trascendían al MIT y llegaban a otros centros tecnológicos.

Michael McMahon (de SRI International) fue una de las personas que supieron del tremendo éxito de EMACS en el MIT. McMahon se encontraba profundamente disgustado por la situación de los editores de texto disponibles para el sistema operativo Twenex (o TOPS-20). Las mejores alternativas para editar texto en TOPS-20 eran TVEDIT, QED o TV (un clon del TECO de 1964). Decidió que EMACS debería funcionar en TOPS-20 y suplir así las tan deseadas capacidades de edición. Tras un gran esfuerzo consiguió hacer funcionar el MIT EMACS en TOPS-20. En 1978 unas pocas máquinas Twenex del MIT, Stanford y SRI ya disponían de su EMACS.

---

<sup>2</sup> Aunque un mejor nombre para la tecla y su efecto hubiera sido “Replace”.

<sup>3</sup> Hubo otros menos importantes: MACROS, Russ-mode, DOC.

En esa época el poder funcionar en TOPS-20 garantizaba una gran compatibilidad con miles de sistemas informáticos repartidos por todo el mundo. Por tanto Richard Stallman realizó una intensiva labor de propagación de EMACS a dichos sistemas. Se dice que consiguió hacer llegar EMACS a virtualmente todos los sistemas TOPS-20 del mundo.

Pero EMACS no solo fue portado a TOPS-20. Bernard S. Greenberg, que trabajaba en Honeywell, escribió un nuevo EMACS para el sistema Multics, que pasó a llamarse *Multics EMACS*. La gran aportación de esta versión fue la sustitución del lenguaje TECO por MacLisp<sup>4</sup>. Este lenguaje estaba mucho mejor dotado que teco en capacidades de estructuración del código y legibilidad, y por tanto la mantenibilidad de los programas resultaba mucho mas facil. Por ello Multics EMACS se benefició de gran cantidad de extensiones escritas por desarrolladores y llegó a superar incluso las funcionalidades ofrecidas por el EMACS original basado en TECO.

A principio de los años ochenta DEC decidió que TOPS-20 ya no era un sistema operativo mantenible para sus máquinas y promovió su sustitución por la combinación VAX/VMS en 1983. Muchos clientes no confiaban en la nueva solución propuesta por DEC, y su reacción fue migrar a Unix en lugar de a VMS.

Tuvieron que portarse muchas aplicaciones de TOPS-20 a Unix. Entre ellas se encontraba EMACS. Fue James Gosling quien llevó a cabo el port. Con la lección de Multics EMACS bien aprendida, Gosling escribió el nuevo editor en el lenguaje C (lenguaje de sistema de Unix). El editor consistía principalmente en un intérprete de Lisp en el que se escribió todo lo demás. TECO resultaba así totalmente abandonado en esta nueva generación de EMACS.

En 1984 Richard Stallman abandonó el MIT para crear el Proyecto GNU y la Fundación para el Software Libre. El objetivo del Proyecto GNU era la construcción de un nuevo sistema operativo del mismo nombre que fuera libre en contraposición al modelo de mercado privativo que por entonces comenzaba a acaparar el mundo del software. El nuevo sistema operativo necesitaba un editor, y dado que ya estaba decidido que fuera tipo Unix, Stallman basó las primeras versiones de GNU Emacs en la implementación de Gosling. A medida que GNU Emacs fue madurando se sustituyeron estas partes de código por otras originales debido a cuestiones legales y de licencias.

Hoy en dia las únicas implementaciones prácticas de EMACS las encontramos en GNU Emacs y XEmacs (este último fruto de una escisión en el desarrollo de GNU Emacs en su versión 18).

---

<sup>4</sup> El sabor de Lisp utilizado en MULTICS

## 2 Codificación del texto

Una “A” (u otro carácter cualquiera) es algo así como una entidad platónica; se trata de la idea de una “A”, y no la “A” en si misma.

- Michael E. Cohen: Text and Fonts in a Multi-lingual Cross-platform World.

### 2.1 Introducción

Desde un punto de vista puramente computacional la unidad básica de un buffer editable es el *carácter*. Sorprendentemente, se trata de un concepto complejo y dado a divergencias entre distintas implementaciones. Por ello es imprescindible efectuar una investigación acerca de qué es un carácter y cómo se puede implementar dicho concepto en EDKIT de la mejor forma posible.

Comenzaremos la investigación exponiendo algunas definiciones que, pese a no ser formalmente adoptadas de forma universal, ayudan a clasificar conceptos comunmente algo difusos, como *repertorio de caracteres* y *codificación*.

### 2.2 Definiciones y conceptos

#### 2.2.1 Repertorios de caracteres

Se trata de un conjunto de caracteres distintos. No se asume ningun aspecto interno de representación, ni existe un orden para los componentes. Esto implica que cualquier orden que quiera imponerse al conjunto de caracteres debe especificarse de forma separada.

Los caracteres pertenecientes al repertorio son identificados por un nombre que, generalmente, viene acompañado por una representación visual del carácter (o *glyph*). Es importante notar que estas representaciones visuales son de **referencia** y por tanto no implican una obligación a la hora de representar visualmente un carácter. De hecho, es muy posible que varios caracteres (distintos; con distinto nombre dentro del repertorio) compartan una representación visual de referencia. En estos casos los caracteres son lógicamente distintos. Por ejemplo, puede ocurrir cuando un repertorio contenga la letra latina “A mayúscula”, la letra cirílica “A mayúscula” y la letra griega “Alfa mayúscula”.

El lector puede preguntarse por qué se ha utilizado el término **repertorio** en lugar de **conjunto**, que parece denotar a la perfección el concepto descrito anteriormente: no hay definido ningun orden implícito entre los caracteres del repertorio. La razón es que, desgraciadamente, existe una sobrecarga en la utilización del término *conjunto de caracteres*, *character set* o *charset*, que puede hacer referencia tanto a un repertorio, a un código o a un codificación de caracteres. Por tanto evitaremos esos términos en esta exposición.

Veamos un ejemplo de repertorio de caracteres: el repertorio de caracteres ASCII.

```
NUL @ SOH A STX B ETX C EOT D ENQ E ACK F BEL G BS H HT I LF J VT
K FF L CR M SO N SI O DLE P DC1 Q DC2 R DC3 S DC4 T NAK U SYN V
ETB W CAN X EM Y SUB Z ESC [ FS \ GS ] RS ^ US _ SPACE ' ! a " b
# c $ d % e & f ' g ( h ) i * j + k , l - m . n / o 0 p 1 q 2 r
3 s 4 t 5 u 6 v 7 w 8 x 9 y : z ; { < | = } > ~ ? DEL
```

Obsérvese que el nombre de muchos caracteres es su representación visual de referencia. Por otra parte, existen caracteres sin representación visual posible: **NUL**, **SOH**, etc. Estos son los **caracteres de control**.

## 2.2.2 Códigos de caracteres

Una código de caracteres es una correspondencia uno a uno entre los caracteres que componen un repertorio y un conjunto de números enteros no negativos. Esto es, un código asigna una **posición de código** distinta a cada uno de los caracteres del repertorio. Es muy comun denominar simplemente **códigos** a estas posiciones de código.

El conjunto de números no negativos que conforman los códigos no tienen por qué ser consecutivos. Tanto es así que es habitual que los códigos incluyan “agujeros” en sus asignaciones de números cuyos códigos serán utilizados en el futuro o bien son utilizados en la actualidad para funciones de control.

Es de notar que los códigos pueden especificarse en distintas bases numéricas. Es común la utilización del decimal (base 10), octal (base 8) y hexadecimal (base 16). En cualquier caso debe considerarse la posición de código como el valor numérico asociado al literal correspondiente. Luego, por ejemplo, en un código de caracteres la posición de código 10 (decimal) es la misma que *0xA* (hexadecimal) o 12 (octal).

Como ejemplo se expone a continuación el código habitual utilizado en ASCII. Obsérvese que en este caso los códigos vienen especificados en tres bases de numeración distintas.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL	107	71	47	G
010	8	08	BS	110	72	48	H
011	9	09	HT	111	73	49	I
012	10	0A	LF	112	74	4A	J
013	11	0B	VT	113	75	4B	K
014	12	0C	FF	114	76	4C	L
015	13	0D	CR	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V

027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\
035	29	1D	GS	135	93	5D	]
036	30	1E	RS	136	94	5E	^
037	31	1F	US	137	95	5F	_
040	32	20	SPACE	140	96	60	'
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	\$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

### 2.2.3 Codificaciones de caracteres

Una *codificación de caracteres* es un algoritmo para representar digitalmente caracteres pertenecientes a un código. Define una correspondencia entre secuencias de posiciones de código de los caracteres y secuencias de octetos.

Para repertorios de caracteres con menos de 255 componentes es posible establecer una codificación que haga corresponder un solo octeto a cada posición de código, tal y como ocurre con el ASCII. Para repertorios mas grandes es necesario utilizar métodos de codificación mas complejos.

### 2.2.4 CCS + CES

Como se ha expuesto, los repertorios, códigos y codificaciones de caracteres son conceptos distintos aunque relacionados. A menudo es necesario utilizar un término que se refiera al conjunto **repertorio + código**. Una buena opción es el acrónimo CCS (Coded Character Set) tal y como especifica el RFC 2278.

Por otra parte es común denominar a las codificaciones de un CCS como CES o “Character Encoding Scheme”.

En definitiva, estándares como ISO 646, ISO 8859-1 o UCS definen tanto un CCS como un CES, o CCS+CES. Aunque el RFC 2278 define que a la combinación de un CCS y un CES se le debe llamar “charset” o “character set”, yo encuentro el término lo suficientemente propenso a confundirse con “repertorio de caracteres” como para evitar su utilización. En su lugar utilizaré el acrónimo CCS para referirme a las combinaciones CCS+CES.

## 2.3 La familia ISO-8859

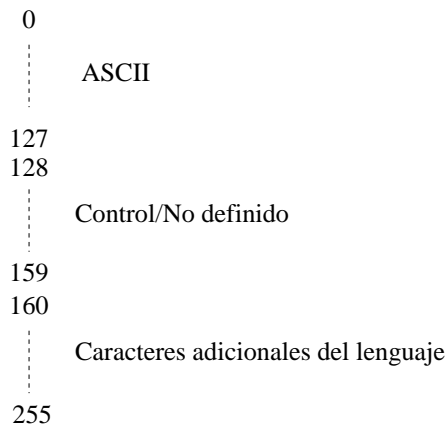


Figura 2.1: Codificación ISO-8859

Los estándares ISO 8859 definen repertorios de caracteres y codificaciones similares al del ASCII: cada posición de código es codificada en un octeto. Las características mas importantes de los CCS definidos por estos estándares son:

- Cada estándar define un repertorio de caracteres que tiene un nombre (tal y como “*Latin alphabet No. 1*”) y está ligado a determinados tipos de lenguajes o zonas de influencia. Así, **ISO 8859-1** incluye caracteres utilizados en las lenguas europeo-occidentales tal y como el inglés, el español y el alemán. Por su parte **ISO 8859-7** (“*Latin/Greek alphabet*”) incluye los caracteres utilizados en la escritura de griego moderno. Véase Tabla 2.1 donde se encuentran listados los estándares ISO 8859 definidos hasta ahora.

- Todas las codificaciones de ISO 8859 siguen la equivalencia **un carácter implica un octeto**. Esto implica que cada uno de los CCS definidos no deben contener mas de 255 caracteres. Esa es la razón por la que existen varios CCS ISO 8859: no podrían meterse todos los caracteres en un solo repertorio y seguir utilizando codificaciones de 8 bits.
- En todos los repertorios y códigos definidos por ISO 8859 el ASCII es un subconjunto. Especialmente importante es el hecho de que las posiciones de los caracteres ASCII en todos los códigos ISO 8859 es el mismo que en la codificación ISO 646 original. Como consecuencia las codificaciones de ISO 8859 funcionan bien con sistemas operativos y entornos preparados para utilizar codificaciones de 8 bits de ISO 646.
- La estructura de los códigos definidos por los estándares ISO 8859 es compartida por todos ellos (véase Figura 2.1). El espacio de los códigos está dividido en tres regiones:

**0 - 127** Región ASCII. Esta parte de los códigos contiene los mismos caracteres que en ASCII y con las mismas posiciones.

**128 - 159** Región no utilizada para significar caracteres. Está pensada para incluir caracteres de control propios de cada aplicación. Por ejemplo, supóngase que un editor de texto considera el carácter nulo (ASCII **0x00**) como editable (como es el caso de TECO). Esto implica que debe utilizarse algun otro caracter como terminador de la secuencia de texto. Podría utilizarse alguno de los caracteres de este rango para ello.

**160 - 255** Región específica de cada estándar. Es aquí donde cada lenguaje define sus caracteres propios. Por ejemplo, las letras minúsculas con acento grave, por ejemplo, están definidas en esta región dentro del CCS ISO 8859-1 (“Latin 1”). Esta es la única parte en la que los distintos códigos ISO 8859 difieren entre sí.

<b>Estándar</b>	<b>Nombre del alfabeto</b>	<b>Caracterización</b>
ISO 8859-1	Latin alphabet No. 1	“Western”, “West European”
ISO 8859-2	Latin alphabet No. 2	“Central European”, “East European”
ISO 8859-3	Latin alphabet No. 3	“South European”; “Maltese & Esperanto”
ISO 8859-4	Latin alphabet No. 4	“North European”
ISO 8859-5	Latin/Cyrillic alphabet	(para lenguajes eslavos)
ISO 8859-6	Latin/Arabic alphabet	(para el lenguaje árabe)
ISO 8859-7	Latin/Greek alphabet	(para griego moderno)
ISO 8859-8	Latin/Hebrew alphabet	(para Hebreo y Yiddish)
ISO 8859-9	Latin alphabet No. 5	“Turkish”
ISO 8859-10	Latin alphabet No. 6	“Nordic” (Sámi, Inuit, Icelandic)
ISO 8859-11	Latin/Thai alphabet	(para el lenguaje Thai)
<i>El conjunto 12 estaba pensado para Vietnamita, pero finalmente no se utilizó</i>		
ISO 8859-13	Latin alphabet No. 7	Baltic Rim
ISO 8859-14	Latin alphabet No. 8	Celta
ISO 8859-15	Latin alphabet No. 9	“euro”
ISO 8859-16	Latin alphabet No. 10	Para varios lenguajes: Albanés, Croata, Inglés, Finés, Francés, Alemán, Húngaro, Irlandés, Gaélico (con la nueva ortografía), Italiano, Latín, Polaco, Rumano y Eslovaco. En particular, contiene las letras s y t con una tilde debajo para el rumano.

Tabla 2.1: Familia ISO 8859

Los CCS definidos por los estándares ISO 8859 están muy bien soportados por los sistemas operativos y entornos de aplicaciones. Gran parte de este éxito de implantación es la posibilidad de codificar estos CCS en un solo octeto y así no requerir muchas modificaciones en el funcionamiento del entorno. Por contra, presentan la gran desventaja de no poder expresar caracteres pertenecientes a distintos códigos ISO 8859 en el mismo contexto. Esto nos impide, por ejemplo, mezclar textos en español y griego moderno: o utilizas ISO 8859-1 o ISO 8859-7.

## 2.4 Páginas de códigos en MS Windows

La empresa de software privativo Microsoft optó por diseñar repertorios, códigos y codificaciones de caracteres propios para sus sistemas operativos en lugar de adherirse a los estándares ISO disponibles de la época. Esta decisión, sin duda alguna bastante objetable en vista a la portabilidad y contribución a la estabilización de estándares abiertos, supuso el diseño de las *páginas de códigos* de MS-DOS primero, y posteriormente las *páginas de códigos* de MS Windows.

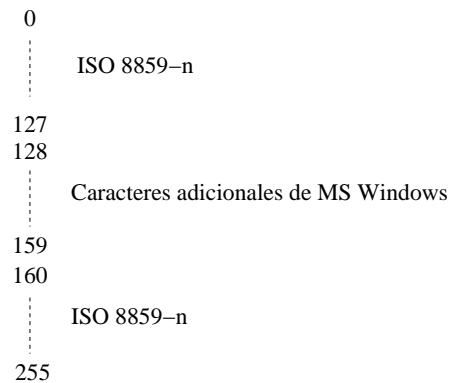


Figura 2.2: Codificación MS Windows

Los códigos de página de MS Windows son similares a los estándares ISO 8859. Así, cada código de página equivale a un grupo de lenguajes con características similares. Estructuralmente los códigos de páginas extienden los CCS ISO 8859 asignando caracteres en la región de control del código (algo explícitamente prohibido en cualquier estándar ISO 8859). Por ejemplo, el “*WinLatin 1*” o *Windows code page 1252* es equivalente al ISO 8859-1 con el añadido de algunos caracteres imprimibles en el rango **128 - 159** (como curiosidad, uno de estos caracteres “alienígenas” es el del símbolo de copyright y otro el de marca registrada. Como siempre, Microsoft muestra sus prioridades).

En resumen: los códigos de página de MS Windows son similares en estructura a los estándares ISO 8859 pero no idénticos. Utilizan el rango de control **128 - 159** para añadir caracteres extra. Esto implica que los códigos de página suelen definir repertorios mas amplios que los ISO 8859. Es importante notar que la estructura de los códigos de página no es muy coherente: hay mas diferencias entre win-1253 (WinGreek) e ISO 8859-7 que entre win-1252 (WinLatin) e ISO 8859-1.

Al repertorio de caracteres de MS Windows a veces se le llama “ANSI character set”. Esto es una seria equivocación: jamás ha sido aprobado por la ANSI. Sin embargo, es rotundamente cierto que Microsoft basó el diseño de su repertorio de caracteres en un conjunto de drafts de ANSI. Un documento publicado por Microsoft admite esto último de forma explícita.

## 2.5 ISO 10646, UCS y Unicode

A finales de los años ochenta la comunidad informática comenzó a apreciar un problema: la proliferación y uso masivo de distintos repertorios, códigos y codificaciones de caracteres de ocho bits: desde ISO 646 hasta los quince miembros de la familia ISO 8859. Como se ha expuesto en los apartados anteriores, los distintos lenguajes ISO 8859 no son compatibles entre sí. Esto implica que un sistema que utilice Latin 1 no podrá leer ficheros producidos en otro sistema que utilice ISO 8859-7 (un correo en griego desde Atenas, por ejemplo). El único CCS compatible entre todos los sistemas seguía siendo el ASCII.

Los problemas consecuencia de esta heterogeneidad en la utilización de CCSs (incompatibles) por parte de los sistemas operativos y las herramientas de sistema pueden resumirse en los siguientes puntos.

### Sistemas operativos

Los sistemas operativos comunmente utilizados, tanto POSIX como no-POSIX (MS Windows, por ejemplo) fueron desarrollados sobre hardware diseñado en los EEUU, y por tanto implementan de forma nativa soporte para el CCS ASCII. El impacto de la codificación del CCS (un carácter  $\Rightarrow$  un octeto) en el diseño de los componentes del sistema operativo es grande, e implica que la adopción de CCSs cuyas codificaciones sean multibyte es un proceso complejo y delicado.

Como ejemplo de esta dependencia fatal, podemos mencionar los sistemas de ficheros de sistemas POSIX. En POSIX los nombres de los ficheros no pueden incluir caracteres nulos (¡octetos nulos!) ni slashes (/). Esto es algo soportable mientras la codificación de los caracteres de los nombres de los ficheros sea monobyte: la única posibilidad de encontrar un octeto con todos sus bits puestos a cero equivale a encontrar el carácter ASCII **NUL** (0x00). Dado que en POSIX los caracteres **NUL** sirven como marca de terminación de cadena, se concluye que en el nombre de los ficheros de sistemas POSIX solo puede figurar un octeto 0x00 al finalizar la cadena.

Como vemos, el hecho de que los sistemas operativos implementen el concepto “Una cadena de caracteres o string finaliza con un octeto 0x00” en oposición a “Una cadena de caracteres o string finaliza con un carácter **NUL**” implica un gran impedimento para la adopción de CCSs multibyte.

En el panorama que se presenta aquí, donde variados CCSs monobyte conviven en distintas máquinas, el sistema operativo no ha tenido que cambiar su orientación a ASCII para funcionar. Prácticamente la totalidad de las codificaciones de 8 bits corresponden a códigos que son superconjuntos de ASCII. Esta es la razón de poder incluir nombres de ficheros codificados en variados CCSs (como Latin-1) sin grandes impactos en el sistema operativo: la base sigue siendo “Un carácter  $\Rightarrow$  un octeto”.

### Herramientas de sistema

Las herramientas de sistema tales como editores, compiladores, formateadores, etc, son muy sensibles a la codificación de caracteres utilizada en el sistema. Cuando en un sistema conviven varias codificaciones distintas de códigos de ocho bits el usuario debe especificar explícitamente qué codificación utilizar en cada momento. Los compiladores, por poner otro ejemplo, también son muy sensibles a la codificación de los programas que se les pasa para procesamiento.

## 2.5.1 ISO 10646: UCS

En 1993 se diseñó el primer CCS pensado para albergar todos los caracteres utilizados en el mundo: los caracteres latinos, árabe, hebreo, símbolos matemáticos y científicos, e incluso caracteres utilizados en algunos lenguajes ficticios como los élficos de Tolkien y el Klingon. La propuesta se convirtió en el estándar ISO 10646<sup>1</sup>.

---

<sup>1</sup> El número del estándar intencionadamente nos recuerda al del ASCII: ISO 646

ISO 10646 describe un gran repertorio de caracteres denominado *UCS* (Universal Character Set) así como un código que lo ordena. El repertorio se diseñó como un CCS de 31 bits, esto es, capaz de albergar  $2^{31}$  caracteres distintos: en total, 2.147.483.648. Evidentemente, esto proporciona un espacio en el código capaz de albergar cualquier carácter imaginable.

Al ser tan grande, el espacio del código se divide en 128 *grupos* cada uno conteniendo 256 *planos* de 16 bits capaces de albergar 65.534 caracteres. Por su parte, cada plano se divide en 256 *columnas* con 256 *células* en cada una. Cada célula contiene un carácter. Luego se comprueba que en UCS hay espacio para  $128 \cdot 256 \cdot 256 \cdot 256 = 2^{31}$  caracteres distintos. Véase Figura 2.3 para una representación visual del espacio.

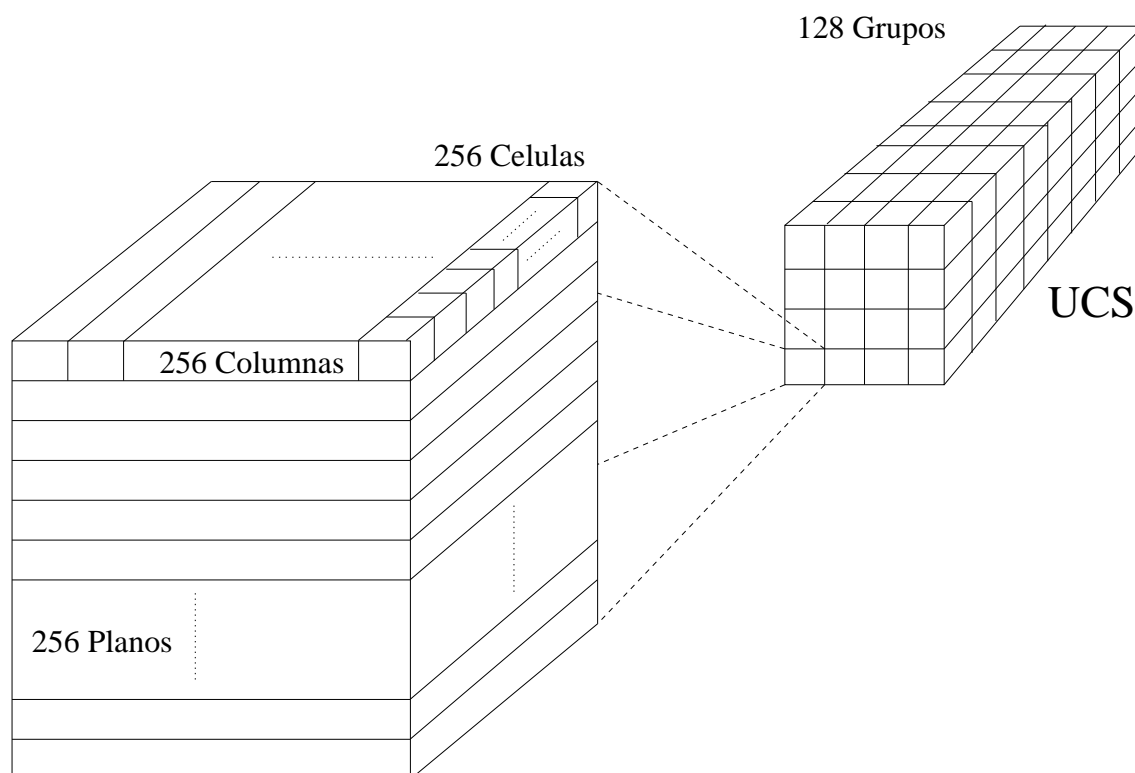


Figura 2.3: Espacio UCS

Las distintas revisiones del estándar Unicode fueron añadiendo caracteres a UCS y de este modo definiendo los planos:

- (1993) ISO 10646-1  $\Rightarrow$  *Basic Multilingual Plane* o *Plane 0*. Rango  $0x0000 - 0xFFFF$ .
- (2001) ISO 10646-2  $\Rightarrow$  Otros planos.

En realidad casi todos los caracteres utilizados en la práctica están en el *BMP* (Basic Multilingual Plane). La excepción son algunos sistemas idiogramáticos asiáticos. Es evidente que el espacio UCS es **enorme**. Es prácticamente imposible que llegue a llenarse alguna vez, ya que existirían mas de dos mil millones de caracteres.

Una de las grandes aportaciones del estándar ISO 10646 fue el cambio radical del concepto de **texto plano**:

**El texto plano no tiene por qué ser monoespaciado.**

El antiguo concepto de texto plano (en realidad, texto ASCII) incluía la idea del espaciado uniforme entre caracteres, esto es, el *ancho* que un carácter ocupa en cualquier medio es único para todo el repertorio. Como es natural muchos programas con interfaces textuales adoptaron esta convención para pintar caracteres en la pantalla.

UCS, por el contrario, permite mezclar en un solo documento muchos tipos de caracteres, desde una **a** minúscula hasta un ideograma egipcio. El ancho de ambos caracteres no tiene por qué ser el mismo.

Es de notar que el estándar no define qué ancho debe ocupar en la pantalla un carácter determinado del repertorio, sino que éste no tiene por qué ser igual al de los demás caracteres del repertorio.

**Los caracteres no deben ser identificados por glifos.**

Como vimos a la hora de exponer el repertorio de caracteres ASCII, muchos de ellos estaban representados por una representación visual como **a** o **#**. En UCS, por el contrario, cada carácter del repertorio viene identificado por un nombre descriptivo, tal y como **Letra latina ‘a’ minúscula**. De esta forma, se ahuyenta toda tentación de confundir un carácter con su representación gráfica, que además puede variar mucho dependiendo de dónde se visualice, y puede ser compartida por varios caracteres lógicamente distintos.

**Los caracteres no identifican el lenguaje en el que está escrito un texto.**

UCS es un solo repertorio de caracteres y, por consiguiente, permite introducir cualquiera de sus caracteres en un texto cualquiera. Esto implica que podemos escribir en múltiples idiomas dentro del mismo texto.

La antigua idea *CCS implica Lenguaje o zona* queda por tanto anulada con un repertorio de caracteres como UCS.

**2.5.2 Unicode**

Paralelamente al diseño del estándar ISO 10646 por parte de la ISO, un consorcio de empresas conocido como *Unicode Consortium* trabajaba en su propia propuesta de un CCS universal: el *Unicode*. Afortunadamente, ambas organizaciones pronto se dieron cuenta de la inconveniencia de tener dos CCS universales distintos, y a partir de finales de los años ochenta fueron de la mano en la elaboración de sus respectivos estándares.

De esta forma, todos los caracteres tienen los mismos nombre y las mismas posiciones en ambos estándares. Véase Tabla 2.2 donde se muestran las equivalencias entre los estándares Unicode y las revisiones a ISO 10646 a lo largo del tiempo.

<b>Año</b>	<b>Estándar Unicode</b>	<b>Revisión ISO 10646</b>
1993	Unicode 1.1	ISO 10646-1:1993
2000	Unicode 3.0	ISO 10646-1:2000
2001	Unicode 3.2	ISO 10646-2:2001
????	Unicode 4.0	ISO 10646-3:????

Tabla 2.2: Equivalencias Unicode-UCS

No obstante ser ambos estándares carácter por carácter compatibles, Unicode es más prolijo en cuanto a la descripción de la semántica propia de cada carácter, incluyendo descripciones de:

- Algoritmos de trazado de caracteres.
- Manejo de texto bidireccional (mezcla de caracteres latinos y hebreos, por ejemplo).
- Algoritmos de ordenación.
- Algoritmos de comparación.
- etc...

### 2.5.3 Codificaciones de UCS/Unicode

Los CCS de “pequeña envergadura” tales como ISO 646 y la familia ISO 8859 inducen de forma natural a una codificación muy simple: se codifica un carácter en un octeto. Evidentemente, esto requiere que no existan en el código más de 255 posiciones.

Dado que los sistemas informáticos siempre han estado orientados a manejo de secuencias de octetos, estas codificaciones presentan muchas consecuencias ventajosas desde un punto de vista técnico. Por ejemplo, puede implementarse una rutina de conteo de caracteres válida para cualquier codificación en 8 bits: simplemente se cuenta el número de octetos de la secuencia. En un esquema de codificaciones de 8 bits, además, la compatibilidad entre los distintos CCS (o sus codificaciones) es más sencilla de obtener. El soporte de los CCS codificados en 8 bits por parte de los sistemas operativos puede implementarse de forma muy abstracta. De este modo muchos de los componentes del sistema operativo no requieren conocer el CCS utilizado para codificar una secuencia de caracteres: trabajan con secuencias de octetos que codifican caracteres. Esta es la razón por la que herramientas POSIX como `wc` y `grep` pueden trabajar con casi cualquier CCS codificable en 8 bits sin ninguna modificación.

Codificar secuencias de caracteres UCS/Unicode no resulta tan sencillo. Partimos de la base de que el código UCS ya no es representable en octetos, sino que se requiere más espacio para almacenar los números enteros del código. El principal inconveniente a la hora de abordar la codificación de valores por encima de 255 en una secuencia de octetos es que existen muchas formas de hacerlo. En el caso específico de UCS, de hecho, existen varias codificaciones alternativas. Cada forma de codificación de UCS posee ciertas virtudes y sufre de ciertos inconvenientes. Como veremos la compatibilidad hacia los sistemas operativos tradicionalmente ASCII ha determinado que se busquen codificaciones específicas.

En los siguientes apartados se examinan varias de las codificaciones propuestas en orden a determinar su adecuación al objetivo del trabajo.

### 2.5.4 Codificaciones nativas: UCS-2 y UCS-4

UCS-2 es una codificación de UCS donde cada carácter se codifica mediante dos octetos. Es decir, mediante un número entero de 16 bits. Es evidente que de esta forma solo pueden codificarse hasta 65535 (es decir, un plano de 16 bits). Por ello UCS-2 solo sirve para codificar caracteres presentes en el “Basic Multilingual Plane”.

Por su parte, UCS-4 es una codificación de UCS donde cada carácter se codifica mediante un número entero de 32 bits. Claramente, UCS-4 permite codificar todo el espacio UCS que, recordemos, es de 31 bits (Véase Figura 2.3).

### 2.5.5 Formatos de transformación

Las codificaciones nativas propuestas por ISO/IEC vistas en el apartado anterior son obvias desde el punto de vista del CCS a codificar: sin dar prioridad a ninguna parte del código UCS se utiliza el ancho mas amplio necesario para codificar todos los puntos de código (o de un plano, en el caso de UCS-2). Sin utilizar caracteres surrogados se requieren 31 bits para codificar cualquier punto de código UCS. Alineado a requerimientos hardware: se requieren 32 bits para codificar los puntos de código UCS. El tipo de datos *entero con signo* sirve a la perfección.

Sin embargo estas codificaciones plantean problemas cuando se tiene en cuenta la gigantesca tarea consistente en adecuar los sistemas informáticos actuales a la utilización de UCS/Unicode. Los sistemas operativos, sistemas de ficheros, herramientas de sistema y aplicaciones de usuario dependen en gran medida de las viejas codificaciones mono-octeto. Su adecuación al manejo de UCS-2 o UCS-4 es una tarea enorme. Para suavizar al máximo de lo posible los traumas derivados de una transición semejante se diseñaron codificaciones temporales con ciertas características que los hacen fácilmente adaptables a los existentes sistemas orientados a codificaciones mono-octeto. Estas características incluyen:

1. Compatibilidad con sistemas de ficheros actuales.

Los sistemas de ficheros actuales no permiten la presencia en los nombres de los ficheros del octeto nulo **0x00** ni del octeto correspondiente al carácter ASCII **slash**. Luego dichos octetos no pueden figurar en la codificación de cadenas UCS si se desea compatibilidad con los sistemas de ficheros.

2. Compatibilidad con programas existentes.

Ningún carácter ASCII no intencionado puede figurar en una codificación multibyte, ya que sería interpretado por los programas existentes como un carácter y no como parte de la codificación de otro carácter.

3. Fácil conversión desde/a UCS.

Es razonable pensar que, tarde o temprano, una secuencia multibyte será transformada a UCS-2 o UCS-4. Esta transformación debe ser eficiente y no involucrar operaciones costosas como multiplicaciones o divisiones.

4. El primer octeto debe indicar el número de octetos que siguen en la secuencia multibyte.
5. El número de octetos utilizados para codificar los caracteres no debe ser extravagante.
6. Debe ser posible encontrar el comienzo de un carácter de forma eficiente cuando se parte de una localización arbitraria en la cadena de octetos. Esto es, el formato de transformación debe ser sincronizable.

En los siguientes apartados se examinan dos formatos de transformación propuestos por ISO 10646: *UTF-16* y *UTF-8*.

### 2.5.6 UTF-16

Esta codificación representa cada posición de código del “Basic Multilingual Plane” mediante dos octetos (0x0000 - 0xFFFF). Para representar caracteres no pertenecientes al BMP se utilizan los llamados *pares surrogados*, utilizando algunas posiciones de código especiales del BMP para tal fin.

La codificación UTF-16 es muy simple siempre y cuando la secuencia de texto contenga únicamente caracteres del BMP.

En definitiva, UTF-16 es una extensión de UCS-2 en la cual pares de determinadas palabras UCS-2 pueden utilizarse para codificar caracteres no pertenecientes al BMP (hasta 20 bits: **0x10FFFF**).

### 2.5.7 UTF-8

UTF-8 es un formato de transformación (esto es, una codificación de UCS/Unicode seguro para sistemas de ficheros y comunicación con núcleos Unix) en el que los caracteres se codifican con una amplitud variable.

Este método de codificación fué inventado por Ken Thompson en circunstancias cuanto menos curiosas. La gente de los Laboratorios Bell (en los que se incluían Ken Thompson y Rob Pike) estaban trabajando en la incorporación de soporte para Unicode en *Plan 9*<sup>2</sup>. Para ello utilizaron UTF-16 (el formato de transformación presente en ISO 10646). Sin embargo, no estaban nada satisfechos con ese formato debido a sus evidentes limitaciones y la falta de “elegancia” debida a la utilización de los pares surrogados. El lanzamiento del sistema operativo estaba próximo cuando, inesperadamente, Rob Pike recibió una llamada de unos colegas de IBM. Esta gente estaba en Austin como parte de una reunión del comité **X/Open**. Como una aportación a la reunión habían diseñado un formato de transformación UCS nuevo, denominado FSS/UTF (File System Secure UCS Transformation Format). En definitiva, querían ver si era posible que la gente de los Bell revisara la propuesta de FSS/UTF en busca de posibles mejoras, antes de presentarla al comité. Tan escarmentado estaba Rob Pike de las limitaciones de UTF-16 que comprendió inmediatamente por qué la gente de IBM estaba buscando una mejora.

Rob Pike se citó con Ken Thompson esa misma noche y, durante la cena, remendaron la propuesta de la gente de IBM para montar un buen formato de transformación. A la vuelta al laboratorio una vez acabada la cena llamaron a los colegas de IBM y les explicaron sus modificaciones a la propuesta. A la gente de IBM les encantó y preguntaron cuanto tardarían en implementarlo en los Bell. Entonces Pike y Thompson se comprometieron a tener Plan-9 funcionando con el nuevo esquema de codificación antes del lunes siguiente.

Por la noche Ken Thompson escribió librerías para empaquetar y desempaquetar la codificación mientras Rob Pike trabajaba en las librerías de C. Al día siguiente ya tenían el código funcionando y procedieron a convertir todos los ficheros de texto de Plan-9 a la nueva codificación.

Como curiosidad es de merecida mención el nombre del nuevo tipo C correspondiente a los caracteres codificados en UTF-8: **rune**.

La codificación UTF-8 no es muy complicada. Codifica caracteres UCS en el rango **[0,0x7FFFFFFF]** utilizando secuencias de octetos de longitud 1, 2, 3, 4, 5 o 6. Para todos los caracteres codificados en mas de un octeto el byte inicial determina el número de octetos total utilizados. En el resto de octetos el bit de mayor orden siempre está encendido. Luego todo octeto que no comience con **10xxxxxx** es el comienzo de un carácter UCS. Véase Tabla 2.3.

---

<sup>2</sup> Una super evolución de Unix.

	Bits	Hex. Min.	Hex. Max.	Secuencia de bits codificada
1	7	00000000	0000007F	0vvvvvvv
2	11	00000080	000007FF	110vvvvv 10vvvvvv
3	16	00000800	0000FFFF	1110vvvv 10vvvvvv 10vvvvvv
4	21	00010000	001FFFFF	11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
5	26	00200000	03FFFFFF	111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
6	31	04000000	07FFFFFF	1111110v 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv

Tabla 2.3: Codificación UTF-8

El esquema de codificación UTF-8 disfruta de las siguientes características, muy deseables:

1. Transparencia y unicidad de los caracteres ASCII.

Los caracteres ASCII de 7 bits {U+0000..U+007F} se especifican de forma transparente en UTF-8 como {=00..=7F} y todos los caracteres no ASCII se representan mediante octetos con el bit más significativo puesto a 1 (es decir, valores puramente de 8 bits) {=80..=F7}. De este modo se elimina la posibilidad de confundir un octeto como un carácter ASCII cuando éste no lo es.

2. Auto sincronización.

En UTF-8 siempre seremos capaces de reconocer un octeto “líder” **11vvvvvv** de un octeto “seguidor” **10vvvvvv**, y por tanto no permite ambigüedad en cuanto al comienzo o la extensión de un carácter multiocteto. Los octetos “líder” actúan de octetos de sincronización, y siempre es posible encontrar uno partiendo de cualquier punto de la cadena multiocteto.

3. Está bien adaptado a los procesadores actuales.

Las secuencias UTF-8 pueden leerse y escribirse de forma rápida y eficiente utilizando las operaciones de enmascaramiento de bits y desplazamiento de bits proporcionadas por los procesadores. Particularmente, no se requiere la utilización de operaciones costosas como la multiplicación o la división. Además, dado que el octeto “líder” especifica la longitud del carácter, es muy sencillo superar dicho carácter.

4. El espacio ocupado por los textos no es extravagante.

UTF-8 es una codificación muy concisa en cuanto a tamaño ocupado en memoria por las secuencias de caracteres. Los textos compuestos únicamente de caracteres ASCII siguen ocupando lo mismo que con las codificaciones de 8 bits. Los textos compuestos de caracteres latinos ocuparán aproximadamente el doble (dos octetos por carácter). Únicamente los caracteres no latinos verán triplicado su tamaño.

Sin embargo, UTF-8 también adolece de ciertos problemas:

1. Longitud variable de los caracteres.

No es posible calcular el número de caracteres que codifica una secuencia de octetos del mero número de octetos que ocupa la secuencia. Y viceversa, naturalmente. De forma similar, no es posible cargar cualquier carácter UTF-8 en un registro del procesador, ya que se desconoce a priori su tamaño.

2. Consumo de octetos extra.

UTF-8 utiliza dos octetos para codificar todos los caracteres no latinos (como griego, cirílico, árabe, etc) que tradicionalmente han sido almacenados en un octeto, y tres octetos para codificar todos los símbolos que tradicionalmente han sido codificados en dos octetos (ideogramas asiáticos, principalmente).

3. Posibilidad de secuencias ilegales.

Dada la naturaleza redundante de UTF-8 (completamente deliberada, como se ha visto) existen secuencias de octetos que no son codificaciones legales de UTF-8. Esto implica que deben implementarse algoritmos de recuperación de errores en los programas o librerías que trabajan con UTF-8.

4. Caracteres de 8 bits.

UTF-8 utiliza caracteres de 8 bits. Esto puede ser un problema cuando se transmite texto codificado en UTF-8 mediante correo electrónico (que estaba originalmente concebido para transportar caracteres codificados en 7 bits).

Este problema llevó a la invención de otro formato de transformación: UTF-7. Por otra parte, la utilización de MIME y QuotedPrintable en los correos electrónicos es suficiente para evitar este problema.

5. Incompatibilidad con ISO 8859-1 (Latin 1).

Pese que el CCS ISO 8859-1 es un subconjunto de ISO 10646 (es decir, ambos códigos contienen los mismos caracteres en las mismas posiciones) la codificación UTF-8 de UCS no es compatible con la codificación utilizada en ISO 8859-1. Recuérdese que en ISO 8859-1 se utilizaba el rango [160-255] para representar los caracteres propios de los idiomas latinos. Pues bien, en UTF-8 no es posible almacenar en un octeto valores mayores de **10xxxxxx** excepto en los octetos que encabezan los caracteres (y éstos nunca utilizan todos los bits como parte de la codificación en sí). Luego ambas codificaciones son claramente incompatibles.

Esto puede ser un problema en la transición de ISO 8859-n a UTF-8, ya que al no existir transparencia el cambio es muy radical<sup>3</sup>.

## 2.6 Aspectos de implementación

Hasta este punto hemos examinado el problema de la representación de caracteres, internacionalización y la adecuación de utilizar un CCS universal como ISO 10646 o Unicode en lugar de uno de los CCS de 8 bits disponibles como ISO 8859-1. Ahora aplicaremos estos conceptos a la construcción de un programa que requiera manejar caracteres y CCS, tal y como es el caso de EDKIT.

La aparición de los denominados *CCS amplios* (de espacios de código de longitud mayor a 255) ha supuesto un impacto en el diseño y la arquitectura de las aplicaciones relacionadas con la edición de texto, desde los mismos editores hasta los *pretty printers*. Hoy en día no se comprendería la construcción de ningún editor de texto que no soportara UCS de forma nativa.

---

<sup>3</sup> En cambio, la transición ISO 646 a ISO 8859-n fué bastante suave, ya que las codificaciones eran transparentes

### 2.6.1 Representaciones internas y externas

A la hora de plantearse soluciones al cambio de ratio caracteres-octetos es necesario hacer una distinción entre “representaciones internas” y “representaciones externas” de una secuencia de caracteres.

#### Representación interna

Una *representación interna* de un texto es la representación con la que trabaja un programa mientras almacena texto en memoria.

#### Representación externa

Las *representaciones externas* de un texto se utilizan cuando el texto se almacena o se transmite por una línea de comunicaciones. Entre los ejemplos de representaciones externas está el de los ficheros esperando en un directorio a ser leídos y parseados.

Tradicionalmente no ha habido diferencias entre ambas formas de representación. Con los CCS de 8 bits resultaba más cómodo utilizar la misma representación monobyte tanto interna como externamente.

### 2.6.2 La librería C estándar

Es claro que para representar “caracteres amplios” el tipo C `char` ya no es conveniente, ya que tiene una fuerte relación con el concepto de byte u octeto. Por esta razón el estándar *ISO C* introduce un nuevo tipo diseñado para albergar un carácter de una cadena de texto “amplia”.

El tipo ISO C en cuestión es `wchar_t`. El estándar donde se introduce el tipo `wchar_t` no especifica nada acerca de su representación interna. Solo exige que dicho tipo sea capaz de almacenar los códigos del CCS base utilizado por el sistema operativo. Luego sería legítimo definir `wchar_t` como simplemente `char`, lo cual tendría sentido por ejemplo en sistemas empotrados o limitados de otra forma. Otro requerimiento es que la representación interna de un carácter amplio sea de anchura fija.

En sistemas GNU, sin embargo, `wchar_t` siempre es de 32 bits y, por tanto, es capaz de codificar cualquier carácter UCS-4 cubriendo de este modo todo el espacio UCS. Algunos sistemas Unix definen `wchar_t` como un tipo de 16 bits y, por tanto, siguen la definición original de Unicode (16 bits) de forma muy estricta. En estos sistemas la única forma de representar caracteres UCS es utilizando UTF-16 (caracteres surrogados). Dado que UTF-16 es, al fin y al cabo, una forma de codificación de anchura variable, contradice claramente el propósito del tipo `wchar_t`.

Un programa puede utilizar `wchar_t` internamente para almacenar caracteres UCS-4 y, por ende, asegura así la capacidad de soporte tanto UCS como Unicode. Sin embargo, como vimos en apartados anteriores, esta forma de representación presenta problemas a la hora de transmitir los caracteres por canales de comunicaciones o almacenarlos en sistemas de ficheros:

- Dado que cada carácter `wchar_t` consiste en más de un octeto hay que tener en cuenta el *orden de bytes* de la máquina o sistema operativo en cuestión. Máquinas con *endian* diferentes interpretarían diferentes valores sobre los mismos datos.
- Los protocolos de comunicaciones están orientados a octetos y, por tanto, el programa tiene que enfrentarse al problema de partir las cadenas de `wchar_t` en octetos.

- El tamaño que ocupan los `wchar_t` suele ser mucho mayor que si se utilizan otras codificaciones hechas mas a la medida (como UTF-8).

Por tanto, lo razonable es que `wchar_t` se utilice como una forma interna de representación del texto y se haga uso de una representación externa mas adecuada a los sistemas operativos y protocolos de comunicaciones. Una buena opción es utilizar UTF-8 para esta representación externa, ya que es seguro respecto a los sistemas de ficheros y no presenta problemas de endian. Además, proporciona una compatibilidad total con ASCII y es muy compacto.

A efectos prácticos desde GNU glibc 2.2 el tipo `wchar_t` implementa valores ISO 10646 de 32 bits independientemente del *locale* POSIX activo. Este hecho se comunica a las aplicaciones mediante la definición de la macro `__STDC_ISO_10646__` tal y como requiere el estándar ISO C99. Todas las funciones ISO C para conversiones a/desde cadenas multibyte (`mbsrtowcs()`, `wcsrtombs()`, etc) están implementadas en glibc 2.2 o superior y pueden utilizarse para convertir entre valores UCS almacenados en `wchar_t` y cualquier codificación multibyte dependiente del *locale* POSIX como UTF-8, ISO 8859-1, etc.

## 2.7 El caso GNU Emacs

El editor *GNU Emacs* es uno de los ejemplos de aplicación sucesivamente modificada para soportar internacionalización en los caracteres. La parte de GNU Emacs que implementa la utilización de diferentes CCS (incluyendo UCS y Unicode) se denomina MULE<sup>4</sup>. Emacs no incluye soporte UCS/Unicode de forma nativa. Para Emacs el UCS no es mas que una codificación de caracteres mas, como ISO 646 y la familia ISO 8859. Debido a ello GNU Emacs utiliza una codificación interna propia donde cada carácter se representa por un número de 24 bits. Dicho número se denomina **buffer code** en contraposición a la codificación utilizada en el fichero correspondiente al buffer, **file code**. Se realiza una conversión a/desde la codificación interna utilizada por Emacs cuando se leen o se escriben ficheros, cuando se envía o se reciben datos de una terminal y cuando se comparte información con otros procesos (como un proceso *shell* arrancado desde Emacs). El formato de codificación de caracteres interno de Emacs es conocido como **emacs-mule**.

```

character: f ('f', 0146, 102, 0x66, U+0066)
  charset: ascii (ASCII (ISO646 IRV))
code point: 102
  syntax: w  which means: word
  category: a:ASCII  l:Latin
buffer code: 0x66
file code: 0x66 (encoded by coding system utf-8-unix)
display: by this font (glyph code)
  -Misc-Fixed-Medium-R-Normal--20-200-75-75-C-100-ISO8859-1 (0x66)

```

Figura 2.4: Carácter en un buffer Emacs

Véase Figura 2.4 para la descripción de un carácter en un buffer Emacs. Se trata del carácter **f**, de código UCS **U+0066**. Es de notar que en este caso el *buffer code* del carácter

---

<sup>4</sup> Multi Lingual Emacs

es el mismo que el *file code*. Esto es así porque la representación UTF-8 del carácter (el fichero está codificado en UTF-8) es idéntica a la representación ASCII en un solo octeto.

## 3 Algorítmica del texto

*Every string which has one end also has another end - Finagle's First Fundamental Finding.*

- Robert Anton Wilson: Schrödinger's Cat Trilogy, 1979.

### 3.1 Introducción

El texto es una entidad puramente *simbólica*. Su tratamiento automático, por tanto, contrasta con el más clásico (y mejor entendido) tratamiento *numérico*.

En un principio todas las computadoras procesaban únicamente problemas numéricos. No fue hasta la construcción de computadoras de propósito general cuando se comenzó a vislumbrar una forma de resolver problemas de índole puramente simbólico.

A lo largo del tiempo, a medida que se fueron cimentando los conceptos básicos de la computación, se desarrolló una rama de la algorítmica para resolver problemas relativos a cadenas de caracteres. Muchos de estos problemas se presentan en las labores cotidianas que tiene que llevar a cabo un editor de texto: ordenación de cadenas, búsqueda de patrones, hashing de subsecuencias, etc.

En los siguientes apartados expondremos alguno de estos problemas y los algoritmos que les dan solución. Dado que el campo es extremadamente amplio nos limitaremos al problema de la búsqueda de patrones fijos, que es con mucho el problema de cadenas de texto más recurrente.

### 3.2 Notación y definiciones previas

Antes de desarrollar los algoritmos de esta sección es conveniente definir y exponer la notación seguida. Hay que notar que, a falta de una notación estandarizada, se ha optado por mimetizar la encontrada en numerosos libros y artículos del ramo.

Los *símbolos* o caracteres son las piezas fundamentales para construir secuencias. Denominamos un *alfabeto* a un conjunto (posiblemente vacío) de símbolos. Una *cadena* definida sobre un alfabeto se define como una secuencia de instancias de los símbolos que pertenecen al alfabeto (entendiendo como instancia una “realización” del símbolo). Por ejemplo, tanto **abc** como **d** son cadenas definidas sobre el alfabeto **{a,b,c,d}**.

Definimos como la *longitud* de una cadena **x**, denotada como  $|\mathbf{x}|$ , como el número de símbolos presentes en **x**. Si  $|x| = m$ , entonces podemos escribir **x** como  $x = x_1x_2\dots x_m$  donde  $x_i$  representa al elemento *i*-ésimo de la cadena. La cadena vacía,  $\epsilon$ , es aquella que tiene longitud cero.

Dos cadenas pueden concatenarse escribiéndose una a continuación de la otra. La concatenación de las cadenas  $x$  e  $y$  se escribe como el producto “mudo” de ambas:  $xy$ , y es igual a la cadena  $xy = x_1x_2\dots x_my_1y_2\dots y_n$ . Una operación asociada a la concatenación es la *exponenciación de cadenas*, que se define recursivamente como  $x^i = x^{i-1}x$  dado que  $x^0 = \epsilon$ . Por ejemplo, si  $x = abc$ , entonces el “cuadrado” de la cadena vendría dado por:  $x^2 = abcabc$ .

Un *lenguaje* es un conjunto de cadenas definido sobre un alfabeto dado. Por ejemplo,  $\{\epsilon, a, aa, aaa, bbbb\}$  es un lenguaje definido sobre el alfabeto  $\{a, b\}$ . Un lenguaje muy importante (aplicable a cualquier alfabeto) es el *cierre de Kleene*. Dado un alfabeto  $C$ , el cierre

de Kleene de  $C$ , denotado por  $C^*$  se define como el lenguaje que contiene todas las palabras que puedan formarse con los símbolos del alfabeto, incluyendo la palabra vacía  $\epsilon$ . El cierre de Kleene en realidad es el cierre reflexivo-transitivo del conjunto alfabeto utilizando la concatenación. Una peculiaridad de estos lenguajes es que, exceptuando el caso en que el alfabeto es el conjunto vacío, siempre contienen infinitas palabras. En efecto, dado el alfabeto  $C = \{a, b\}$ ,  $C^*$  contiene todas las palabras posibles compuestas de  $a$  y  $b$ , siendo la cantidad de estas palabras infinita.

Pasemos ahora a considerar los símbolos que constituyen una cadena. Como hemos visto, todos ellos pertenecen a un alfabeto, que es un conjunto. Una secuencia puede considerarse como un orden total impuesto sobre un subconjunto del alfabeto: *el elemento  $x$  está situado antes del elemento  $y$  en la cadena*. Como tal orden posee una estructura, y podemos basarnos en ella para agrupar determinados símbolos de una cadena que comparten propiedades interesantes. Estos grupos son las *subcadenas* y las *subsecuencias*.

Una *subcadena* de  $x$  es una cadena formada de un grupo de símbolos contiguos de  $x$ . Por ejemplo, dada la cadena  $x = abcdefg$ , podemos definir las siguientes subcadenas:  $a$ ,  $ab$ ,  $def$ ,  $fg$ ,  $c$ , etc. Puede obtenerse cualquier subcadena de una cadena borrando cero o más elementos del comienzo y el final de la misma. Dada una cadena  $x_1x_2x_{m-i}...x_m$  definimos la subcadena  $x(i, j) = x_ix_{i+1}...x_j$  donde  $i \geq 0$  es el *extremo inferior* y  $j \leq m$  es el *extremo superior* de la cadena. Podemos definir los *prefijos* y *sufijos* de una cadena como las subcadenas para las que  $i = 0$  y  $j = m$ , respectivamente. Nótese que la cadena vacía  $\epsilon$  es tanto un prefijo como un sufijo de la cadena, del mismo modo que la cadena completa (la cadena  $abc$  es tanto prefijo como sufijo de  $abc$ , donde  $i = 0$  y  $j = 2$ ). Para expresar el concepto de *prefijo no vacío y distinto de la cadena completa* utilizamos el término *prefijo propio*. De forma similar un sufijo no vacío se denomina *sufijo propio*.

Un concepto relacionado con las subcadenas, pero mucho más general, es el de *subsecuencia*. Una subsecuencia está formada de símbolos *ordenados* de una cadena. Nótese que esta definición no requiere que los elementos contiguos de la subsecuencia lo sean también en la cadena. El único requisito es que para cualquier par de elementos de la subsecuencia  $x_i$   $x_{i+1}$  correspondan a dos elementos de la cadena  $x_j$  y  $x_k$  tal que  $j \leq k$ . Luego podemos obtener subsecuencias de una cadena borrando un número arbitrario de símbolos de la cadena, en cualquier posición.

Finalmente me gustaría resaltar el papel especial de la cadena vacía  $\epsilon$ , que es, al mismo tiempo, *subcadena*, *prefijo*, *sufijo* y *subsecuencia* dada cualquier cadena.

### 3.3 Búsqueda de patrones fijos

Una de las operaciones más utilizadas en los buffers de los editores de texto es la búsqueda de algún patrón. El editor debe analizar el contenido del buffer en busca de subsecuencias que satisfagan el patrón proporcionado por el usuario. Respecto a la composición del patrón caben dos posibilidades:

- El patrón es una cadena de texto que denota el contenido exacto, elemento por elemento, a encontrar en el texto del buffer.
- El patrón es una cadena de texto que denota el contenido exacto, elemento por elemento, a encontrar en el texto del buffer, exceptuando algunos elementos especiales que denotan otro contenido. Un ejemplo de este tipo de patrones son las *expresiones regulares*.

En este apartado nos centraremos en la primera posibilidad: el usuario especifica en el patrón una cadena fija y el objetivo consiste en encontrar una subsecuencia con el mismo contenido.

De este punto en adelante denominaremos **texto objetivo** a la cadena de texto en la cual se buscan las subsecuencias.

Podemos formular formalmente el problema de la búsqueda de cadenas de la siguiente forma:

Dada una *cadena de texto patrón*  $x$ , con  $|x| = m$ , y una *cadena de texto objetivo*  $y$ , con  $|y| = n$ , donde  $m, n < 0$  y  $m \leq n$ , si  $x$  figura como una subcadena de  $y$  determinar la posición en  $y$  de la primera ocurrencia de  $x$ . Esto es, retornar el valor más pequeño de  $i$  dado que  $y(i, i + m - 1) = x(1, m)$ .

A continuación se describen una serie de algoritmos que resuelven este problema. Es de notar que estos algoritmos pueden extenderse fácilmente para determinar *todas* las apariciones de la cadena patrón en la cadena objetivo, y no únicamente la primera. Sin embargo, a efectos de simplicidad, dichas extensiones no se incluyen en esta exposición.

### 3.3.1 Fuerza bruta

El método más intuitivo para buscar la aparición de un patrón dentro de un texto objetivo es el conocido como “fuerza bruta”. Consiste en buscar apariciones de  $x$  en posiciones sucesivas de  $y$ .

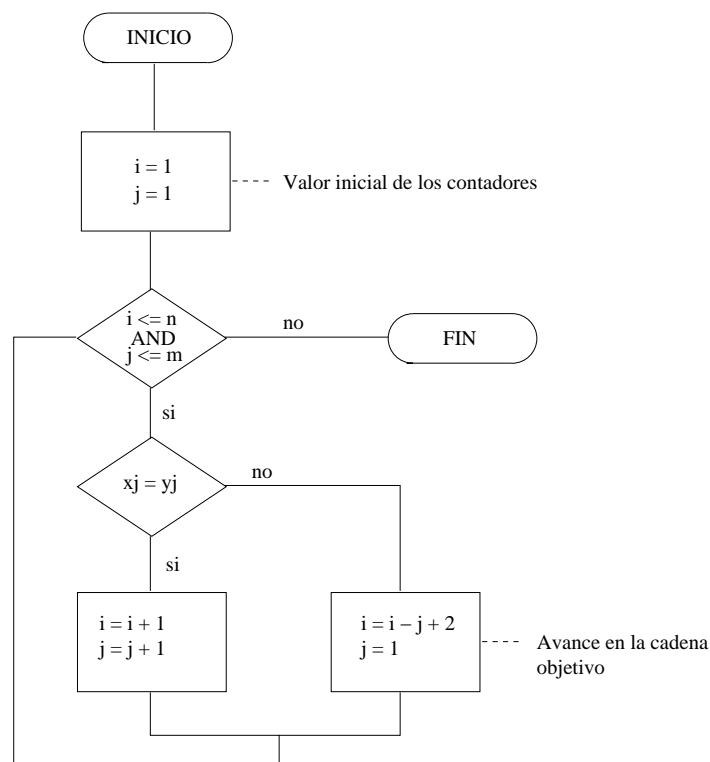


Figura 3.1: Fuerza Bruta

Como puede verse en la Figura 3.1,  $x$  es comparado con las subsecuencias  $y(i, i + m - 1)$  en las posiciones sucesivas  $i$ . El proceso se termina cuando se ha encontrado una aparición del patrón o se ha llegado al final de  $y$  ( $i > n$ ).

Este método sufre de dos grandes inconvenientes: una evidente ineficiencia y el requerimiento de tener almacenada la secuencia objetivo en memoria. En el peor de los casos este método se ejecuta en tiempo  $O(mn)$ . Sin embargo, en aplicaciones prácticas como buscar en un texto escrito en castellano, el rendimiento esperado es  $O(m + n)$ .

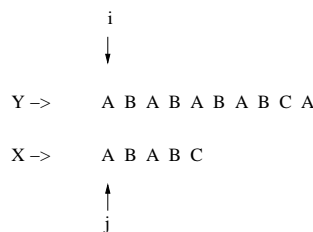
### 3.3.2 Knuth-Morris-Pratt

Un teorema sobre autómatas a pila bidireccionales y deterministas desarrollado por Cook llevó a la determinación del siguiente resultado: existe al menos un algoritmo que resuelve el problema de la búsqueda de patrones fijos en tiempo  $O(m + n)$  en el peor de los casos. Este comportamiento asintótico es lineal y mejora con mucho el peor de los casos del método de la fuerza bruta:  $O(m * n)$ .

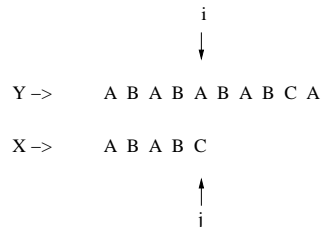
En 1970 Donald Knuth leyó el teorema de Cook y bajo su inspiración desarrolló un algoritmo de búsqueda con el deseable comportamiento asintótico de  $O(m + n)$ . Posteriormente dicho algoritmo fue modificado por Pratt para que el tiempo de ejecución fuera independiente del alfabeto de las cadenas. El algoritmo resultante también fue desarrollado de forma independiente por Morris en 1969, el cual no tenía conocimiento del teorema de Cook. La motivación de Morris era implementar el método en un editor de texto no interactivo, en el cual solo era posible un acceso secuencial a la cadena objetivo, y por tanto no podía utilizar la fuerza bruta con sus saltos atrás. Este algoritmo suele recibir el nombre de *Knuth-Morris-Pratt* o simplemente *KMP*.

El método se basa en el desplazamiento de la cadena patrón sobre la cadena objetivo en el caso de un fallo al comparar sus elementos. El número de elementos que se desplaza el patrón se extrae de información conocida.

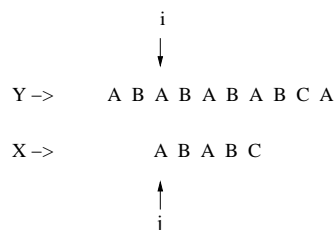
Una buena forma de comprender el algoritmo (que no es ni con mucho trivial) es mediante un ejemplo. Supongamos que estamos buscando la primera aparición del patrón  $x = ABABC$  en la cadena objetivo  $y = ABABABABCA$ . Al igual que en el método de la fuerza bruta comenzaríamos fijando dos índices  $j$  e  $i$  para el patrón y el objetivo, respectivamente. La situación inicial puede verse en la siguiente figura:



Vamos incrementando los índices  $i$  y  $j$  a medida que los elementos son iguales, esto es, mientras  $x(j) = y(i)$ . Sin embargo, al llegar al siguiente punto:

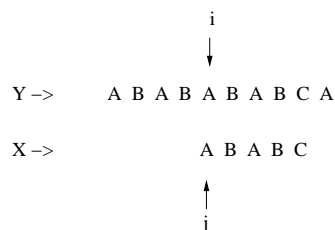


Estamos frente a un fallo en la comparación de los elementos. En el método de fuerza bruta retrocederíamos el índice  $i$  a la posición 2 en el objetivo y volveríamos a comenzar la comparación elemento a elemento. En este caso, pondríamos  $i = 2$  y  $j = 1$ . Pero consideremos la situación con cuidado. A la vista de la posición de las cadenas es evidente que podríamos desplazar el patrón hacia la derecha 2 posiciones sin perder ninguna aparición del patrón en el objetivo:



¿Como hemos determinado este desplazamiento? En el momento del fallo, cuando  $i = 5$  y  $j = 5$ , conocíamos el contenido de la subcadena  $y(1,4)$ , que no es otro que  $x(1,4)$ . Ahora bien. Si hubiéramos desplazado el patrón hacia la derecha en un elemento entonces tendríamos un fallo seguro al comparar el primer elemento, ya que  $y(2) \neq x(1)$ , o lo que es lo mismo,  $x(2) \neq x(1)$ . De esta forma podemos aprovechar nuestro conocimiento a priori del contenido del patrón para conseguir desplazamientos mayores de 1 en el caso de un fallo en la comparación de un elemento.

Siguiendo con nuestro ejemplo, encontraríamos un fallo en la comparación en  $j = 5$  e  $i = 7$ . De nuevo aprovecharíamos nuestro conocimiento del patrón para desplazar éste dos posiciones hacia la derecha:



Finalmente, tras cinco comparaciones de elementos, encontraríamos la aparición del patrón en la cadena objetivo en la posición 5.

En este ejemplo hemos podido observar las dos grandes ventajas del algoritmo KMP frente a la fuerza bruta:

- Dado que el índice de la cadena objetivo  $j$  nunca disminuye el método es aplicable a situaciones donde el acceso a la cadena objetivo es puramente secuencial.
- La utilización del conocimiento a priori del contenido del patrón nos ayuda a no intentar comparaciones “condenadas de antemano”. En el ejemplo hemos ahorrado dos comparaciones del patrón que se traducirían en 10 comparaciones de elementos.

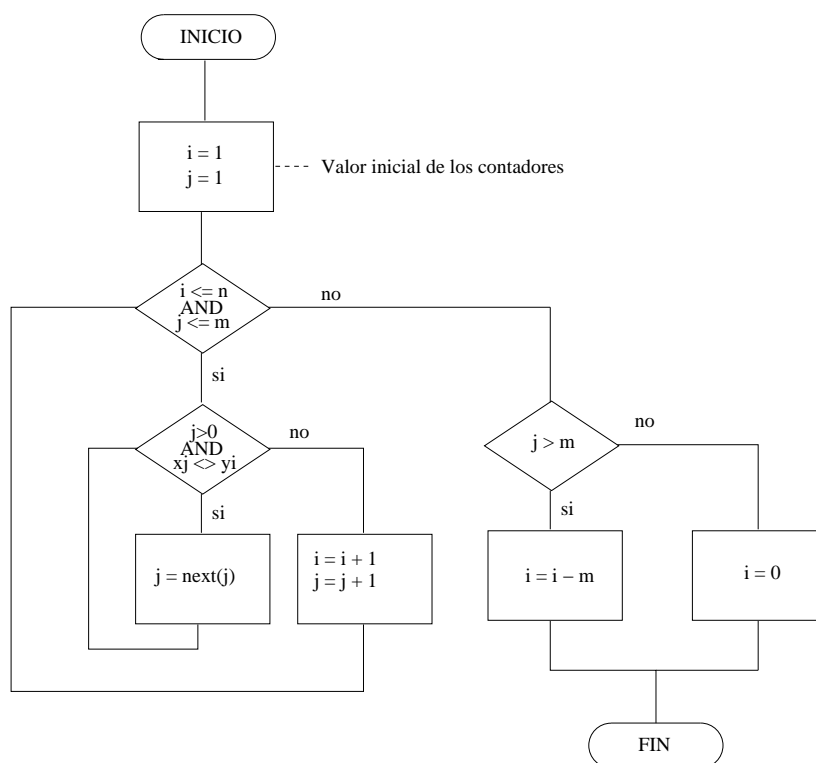


Figura 3.2: Knuth-Morris-Pratt

Veamos el método en detalle. La Figura 3.2 nos muestra el algoritmo KMP modelado en un diagrama de flujo. Al igual que en el caso de la fuerza bruta tenemos un bucle exterior que se encarga de realizar las comparaciones del patrón con la cadena objetivo. La única diferencia con el método de fuerza bruta es el tratamiento del caso de un error en la comparación de un elemento. En lugar de incrementar el índice de la cadena objetivo  $i$  en una unidad y resetear a 1 el índice del patrón  $j$ , obtenemos el nuevo valor de  $j$  (que determina el desplazamiento del patrón sobre el objetivo) de una tabla llamada `next`. Dicha tabla contiene la “información a priori acerca del patrón” de la que hablábamos en el desarrollo del ejemplo. Está indexada por los índices posibles dentro del patrón, y contiene el número de elementos a desplazar al patrón hacia la derecha en la cadena objetivo.

Ya debe ser evidente que la clave del algoritmo KMP está en la construcción de la tabla `next`. Consideremos la siguiente situación: utilizando un patrón  $x$  de tamaño  $m$  se produce un fallo de comparación de elementos  $y(i) \neq x(j)$ . Sabemos que en este momento  $y(i - j + 1, i - 1) = x(1, j - 1)$ . Es posible aprovechar esta información para descartar desplazamientos consecutivos del patrón sobre la cadena objetivo. Si, por ejemplo,  $y(i - j + 2) = x(1)$ , no tiene sentido comparar dichos elementos, ya que sabemos que la comparación fallará. Dado que  $y(i - j + 2) = x(2)$  parece plausible que *la información requerida para tomar estas decisiones está por entero contenida en el patrón*. Esta es la clave del algoritmo KMP.

En efecto, el contenido de la tabla `next` puede determinarse de un exámen del patrón como sigue: en el momento en que se produce un fallo de comparación entre  $x_j$  y  $y_i$ , sabemos que la subcadena  $x(1, j - 1)$  corresponde a los últimos  $j - 1$  símbolos de  $y$ . Si existe un *prefijo propio* de la subcadena  $x(1, j - 1)$ , de longitud máxima  $k - 1$ , igual a un sufijo de la misma subcadena, entonces el siguiente elemento del patrón a comparar con  $y_i$  se obtiene desplazando el patrón hasta que el prefijo ocupe el espacio antes ocupado por el sufijo. Luego el elemento del patrón a comparar con  $y_i$  es el que sigue de forma inmediata al prefijo en el patrón, esto es,  $x_k$ . En el caso especial en el que el fallo se produzca en el primer elemento del patrón, se le da a `next(1)` el valor de 0 para que el patrón entero se desplace una posición a la derecha sobre el texto objetivo.

Luego el valor de cada elemento de la tabla `next(i)` viene dado por el máximo valor  $k$  menor de  $j$  tal que  $x(1, k - 1)$  es un sufijo de  $x(1, j - 1)$ , es decir,  $x(1, k - 1) = x(j - k + 1, j - 1)$ .

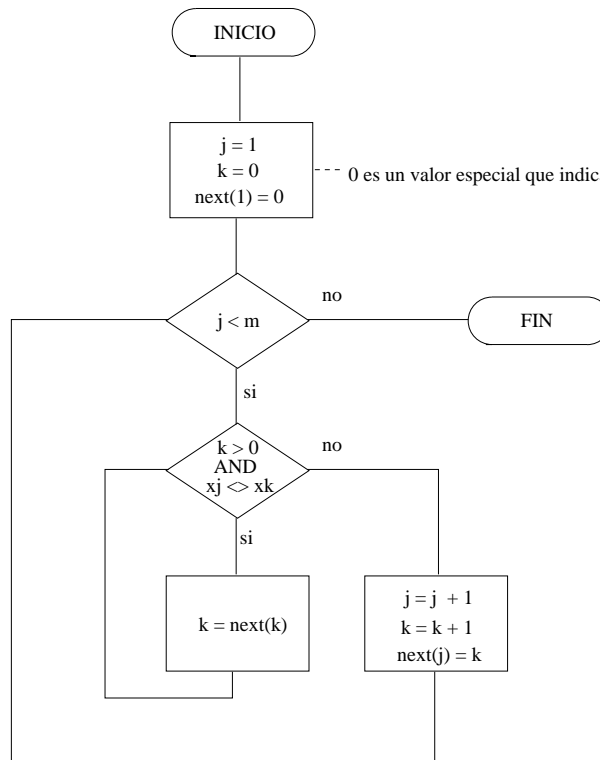


Figura 3.3: Inicialización simple de la tabla `next`

En la Figura 3.3 puede verse un algoritmo para computar los valores de la tabla *next* en base al razonamiento que acabamos de ver. El algoritmo del cálculo de la tabla funciona de la siguiente forma: para cualquier posible  $j$  se compara  $x(1, j - 1)$  consigo mismo deslizando una copia sobre la otra elemento a elemento, de izquierda a derecha. El proceso termina cuando, o bien todos los símbolos apilados coinciden entre sí, o bien no queda ninguno. En el primer caso los símbolos apilados determinan el prefijo de tamaño máximo deseado, y por tanto  $next(j)$  valdrá el tamaño de dicho prefijo mas uno.

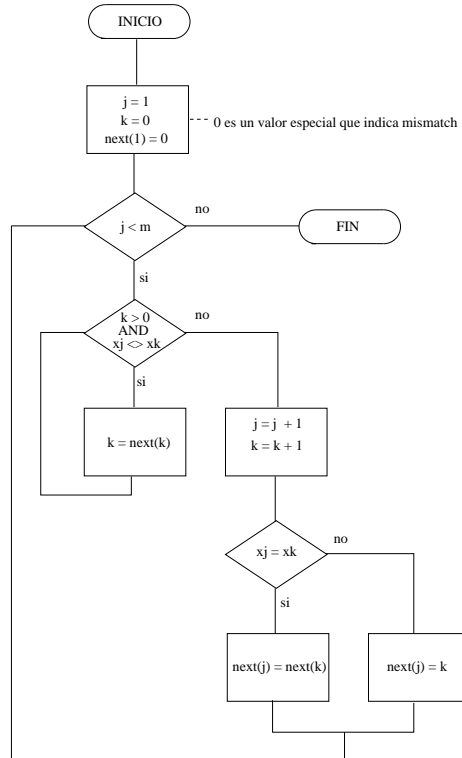


Figura 3.4: Inicialización la tabla *next*

Sin embargo este método es mejorable: todavía puede aprovecharse mas información sobre el patrón para mejorar el proceso de búsqueda. Esto nos lleva a un retoque en la construcción de la tabla *next*. Supongamos que estamos buscando la aparición del patrón **ABCDABCE**. En el caso de un fallo en la coparación de  $x_7$  e  $y_i$ , nuestro esquema nos dice que el siguiente elemento a comparar con  $y_i$  será  $x_3$ . Sin embargo es claro que de esta forma la comparación fallará sea cual sea el valor de  $y_i$ , dado que  $x_3 = x_7 \neq y_i$ . Es decir, volveríamos a comparar  $y_i$  con una **C**. Luego la modificación a la construcción de la tabla *next* es evidente:  $next(j)$  será igual al mayor  $k$  menor de  $j$  tal que  $x(1, k - 1) = x(j - k + 1, j - 1)$  y  $x_j \neq x_k$ , siendo 0 si no existe tal  $k$ . La Figura 3.4 muestra el algoritmo que se ajusta a esta definición mejorada.

El punto flaco del algoritmo KMP consiste en una baja eficiencia en problemas reales, pese a su indudable elegancia teórica. Exceptuando en los casos donde el alfabeto es muy

pequeño y por tanto el texto es muy repetitivo el algoritmo KMP no presenta en la práctica una mejora significativa respecto a la fuerza bruta.

El comportamiento asintótico del KMP es  $O(n + m)$ . Curiosamente el peor comportamiento se da cuando el patrón consiste en una sucesión de Fibonacci. En dichas cadenas los prefijos-sufijos requisitos para una buena tabla *next* brillan por su ausencia.

### 3.3.3 Boyer-Moore

Este rápido algoritmo de búsqueda de subcadenas fue descubierto en 1974 por Boyer y Moore, e independientemente por Gosper. Boyer y Moore publicaron una revisión en 1977 (que conforma lo que se conoce como el “paper original”) recogiendo sugerencias para mejorar el algoritmo de parte de B. Kuipers, Donald Knuth y R.W. Floyd.

La velocidad de este algoritmo (sobre todo considerada en términos medios, realmente sorprendente) se basa en el hecho de que se descartan porciones del texto objetivo de las que se saben que no pueden contener el patrón buscado, de una forma similar al KMP.

¿Como obtenemos la información necesaria para determinar qué porciones del texto objetivo pueden ser descartadas?. En este método el patrón se desplaza sobre el texto objetivo de izquierda a derecha (igual que en fuerza bruta y KMP), pero sus símbolos se comparan con los correspondientes del texto objetivo **de derecha a izquierda**. Luego la primera comparación se realiza entre  $x_m$  e  $y_m$  (recuérdese que  $m$  es la longitud del patrón  $x$ ). Si el símbolo  $y_m$  no figura en ninguna parte del patrón entonces es imposible encontrar una aparición del patrón en los primeros  $m$  símbolos de  $y$ . Luego podemos descartar la porción del texto objetivo  $y(1, m)$ , siendo las siguientes posiciones a comparar  $x_m$  y  $y_{2m}$ .

En general el método utiliza dos heurísticos para determinar qué porción de texto objetivo bajo el patrón puede descartarse. Estos heurísticos son los siguientes:

- El **heurístico de ocurrencia**, cuyos valores se almacenan en la tabla *skip*. Esta tabla está indexada por símbolos, y por tanto tiene tantas entradas como símbolos el alfabeto que se esté utilizando.
- El **heurístico de coincidencia**, cuyos valores se almacenan en la tabla *shift*. Este heurístico es prácticamente idéntico al utilizado en el método KMP y al igual que la tabla *next*, la tabla *shift* se indexa por índices de la cadena patrón.

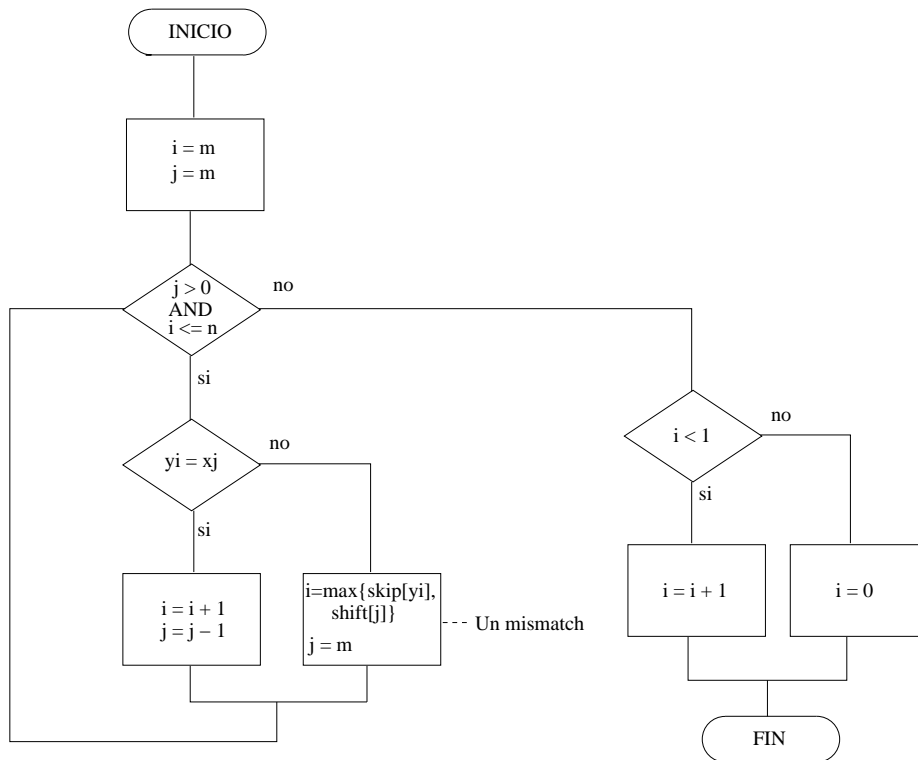


Figura 3.5: Boyer-Moore

En la Figura 3.5 tenemos el algoritmo de Boyer-Moore expresado en un diagrama de flujo. Vemos como en el caso de un fallo de comparación se desplaza el patrón sobre la cadena objetivo una cantidad de posiciones dada por la expresión:

$$\max(\text{skip}(y_i), \text{shift}(j))$$

¿Qué decir acerca de la eficiencia de este algoritmo? El Boyer-Moore es uno de los algoritmos de búsqueda de patrones fijos mas rápidos por término medio. Dado un alfabeto grande y un patrón pequeño el número de comparaciones de símbolos esperados por término medio es  $n/m$ , mientras que en el peor de los casos el comportamiento asintótico es  $O(m+n)$ .

La construcción de ambas tablas contribuyen a la complejidad del algoritmo con un  $O(m)$  de parte de la tabla *shift* y un  $O(m + |C|)$  de parte de la tabla *skip* (donde  $C$  es el alfabeto utilizado).

### 3.3.4 Adecuación a codificaciones multi-octeto

Los algoritmos de búsqueda de cadena vistos en los apartados anteriores se desarrollaron en un contexto donde predominaban las codificaciones mono-octeto. Luego es necesario examinar el comportamiento de los algoritmos cuando es utilizada alguna codificación multi-octeto.

Es evidente que cuanto mayor sea la compatibilidad con ASCII de la codificación multi-octeto, mayor será la probabilidad de que puedan utilizarse estos algoritmos.

Una opción simple consiste en trabajar con representaciones de tamaño fijo de las cadenas. De esta forma no habría que cambiar nada en los algoritmos, ya que por lo general solo dependen de la disponibilidad de alguna función de comparación. Sin embargo esto requiere un preprocesamiento del texto objetivo siempre que esté codificado en multi-octeto.

Hay que tener en cuenta, además, que casi todos los algoritmos de búsqueda de cadenas (aparte del de fuerza bruta) se basan en nociones estadísticas del contenido del texto objetivo para garantizar su eficiencia media y no caer a menudo en el peor de los casos. Es posible que una codificación multi-octeto haga que algunas de estas nociones ya no sean aplicables. Por ejemplo, piénsese en el algoritmo KMP. Hemos visto que se comporta muy eficientemente en los casos en que el alfabeto del patrón es pequeño y el texto objetivo grande.

Por otra parte la utilización de codificaciones multi-octeto desmorona la capacidad del algoritmo KMP de no requerir almacenar el texto objetivo ya analizado.

## 4 Buffers de texto

*El primer editor que utilizó la técnica del buffer gap fue TECO, que además se cuenta entre los primeros editores nunca implementados. Se escribió a comienzos de los años sesenta. Explique por qué casi todo el mundo se pasó los siguientes quince años reinventando editores de líneas difíciles de utilizar y tremendamente limitados (De mediana dificultad, pero si da con una respuesta me encantaría escucharla)*

- Craig A. Finseth: The Craft of Text Editing, (Questions to Probe Your Understanding, capítulo 6)

### 4.1 Introducción

Si tuvieramos que identificar el componente *central* de un editor de texto seguramente apuntaríamos a los *buffes de texto* que contienen la información que se está editando.

Desde un punto de vista de usuario el editor no es mas que los mecanismos funcionales y de interfaz necesarios para manipular el texto. Dado que el texto se almacena y organiza en buffers, resulta razonable que para el usuario el buffer juegue un papel central.

Desde el punto de vista de las implementaciones, por su parte, el buffer sigue siendo el corazón del editor. Recordemos que en la descomposición funcional de Finseth el componente que se encargaba de la gestión de los buffers bajo edición era el *sub-editor*. Los otros dos componentes, *redisplay* e *interacción con el usuario*, accedían al sub-editor para llevar a cabo todas sus tareas. Incluso en las implementaciones que no siguen la descomposición de Finseth se observa este papel central del buffer. De hecho, en estas implementaciones la dependencia con la implementación del buffer suele ser mucho mayor, ya que la descomposición de Finseth sigue un esquema *modelo-vista-controlador* en el que cierta independencia con el modelo (el buffer) de los otros dos componentes está garantizada.

En la literatura y los diseños existentes no se hace una distinción clara entre el concepto de la secuencia editable que almacena el contenido del buffer y la posible estructuración sobre la secuencia, en forma de punteros, marcas u otras estructuras mas complejas. Esto ha desembocado en implementaciones poco claras y en las que la gestión de la secuencia editable afecta tremendamente en la cantidad de abstracciones sobre la misma que se presentan al resto del editor.

Un ejemplo lo podemos encontrar en el editor *Lara*. Este editor incluía ciertas capacidades de estructuración del buffer en párrafos, líneas y palabras. Desgraciadamente su implementación se basaba en el propio mecanismo de almacenamiento de la secuencia: las tablas de piezas. Como resultado sería implensable sustituir este mecanismo por otro mas eficiente o que se comportara mejor en sistemas limitados (como los empotrados). Cualquier sustitución del mecanismo de almacenamiento de la secuencia, que en principio no tiene nada que ver con la estructuración el contenido en párrafos y líneas, supondría una total reimplimentación de dicha estructuración. Ciertamente no se trata de un modelo muy flexible a efectos de modificaciones y mejoras.

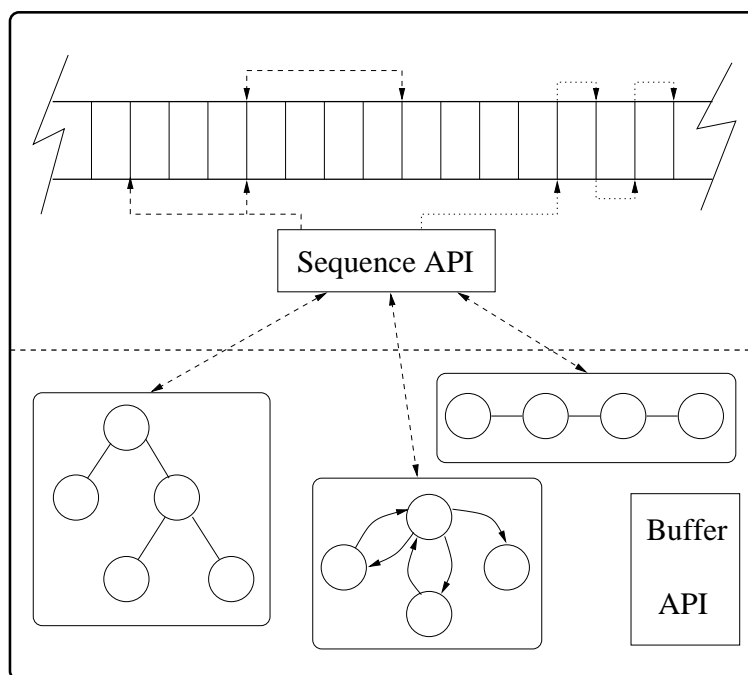


Figura 4.1: Estructura compuesta para buffers

Estos problemas incitan a introducir en este libro una estructuración dual del componente buffer traducida en dos niveles de arquitectura (véase Figura 4.1.) bien diferenciados:

#### Estructuración interna del buffer

Cubre la gestión de la secuencia editable que almacena los contenidos textuales del buffer.

Los problemas a resolver en este nivel tienen que ver sobre todo con cuestiones de almacenamiento en memoria. En efecto, la eficiencia tanto en espacio como en ejecución es tremendamente importante en un correcto mantenimiento del buffer.

A lo largo de los años se han propuesto y experimentado en la práctica diversos mecanismos de gestión de secuencias editables.

#### Estructuración externa del buffer

Este nivel implementa las abstracciones que componen la funcionalidad del buffer ofrecida a los demás componentes del editor.

Su función principal es dotar de estructura a la secuencia editable en mayor o menor medida. Desde la abstracción de *localización* dentro del buffer hasta árboles y grafos de tramos.

Una clara separación entre la gestión interna de la secuencia editable y las abstracciones construidas sobre la misma que conforman el buffer conlleva ciertas ventajas tanto en el diseño como en la implementación.

Dado que la estructuración externa del buffer se comunica con la secuencia editable via una interfaz bien definida, y por tanto la encapsulación de la implementación de la secuencia es total, dicha implementación resulta reemplazable sin necesidad de modificaciones en la estructuración externa del buffer.

Como se verá en los apartados siguientes los distintos mecanismos existentes para almacenar secuencias presentan ventajas e inconvenientes, algunos de ellos determinados por el área de utilización concreta del editor (qué se va a editar) o el sistema en que va a funcionar (sistemas limitados de espacio o procesador). De ahí que la capacidad de reemplazar fácilmente la gestión del almacenamiento de las secuencias permita un diseño muy flexible y portable.

En los apartados siguientes se estudian ambos niveles de estructuración.

## 4.2 Estructuración interna

### 4.2.1 Secuencias editables

El contenido de un buffer de un editor de texto consiste en una secuencia de elementos típicamente caracteres codificados mediante algun CCS.

Existen muchas estructuras de datos capaces de almacenar secuencias. Un enfoque para estudiar dichas estructuras es el adoptado por Crowley, que clasifica las secuencias como un conjunto ordenado que respeta ciertas características. La Figura 4.2 muestra una jerarquía de tipos abstractos de datos correspondientes a conjuntos ordenados. Nótese que en los tipos abstractos de datos se presuponen tres operaciones primitivas:

#### **Inserciones**

Las inserciones añaden elementos al conjunto ordenado. El cómo se organiza el elemento dentro de la estructura del conjunto ordenado es determinado por el tipo concreto de conjunto.

#### **Borrados**

Los borrados eliminan elementos del conjunto ordenado. De forma similar a las inserciones el efecto del borrado en la estructura del conjunto varía en función del tipo del mismo.

#### **Lookups**

Los *lookups* permiten recuperar la información asociada a un elemento presente en el conjunto ordenado. El cómo el usuario hace referencia a dicho elemento depende del tipo concreto de conjunto.

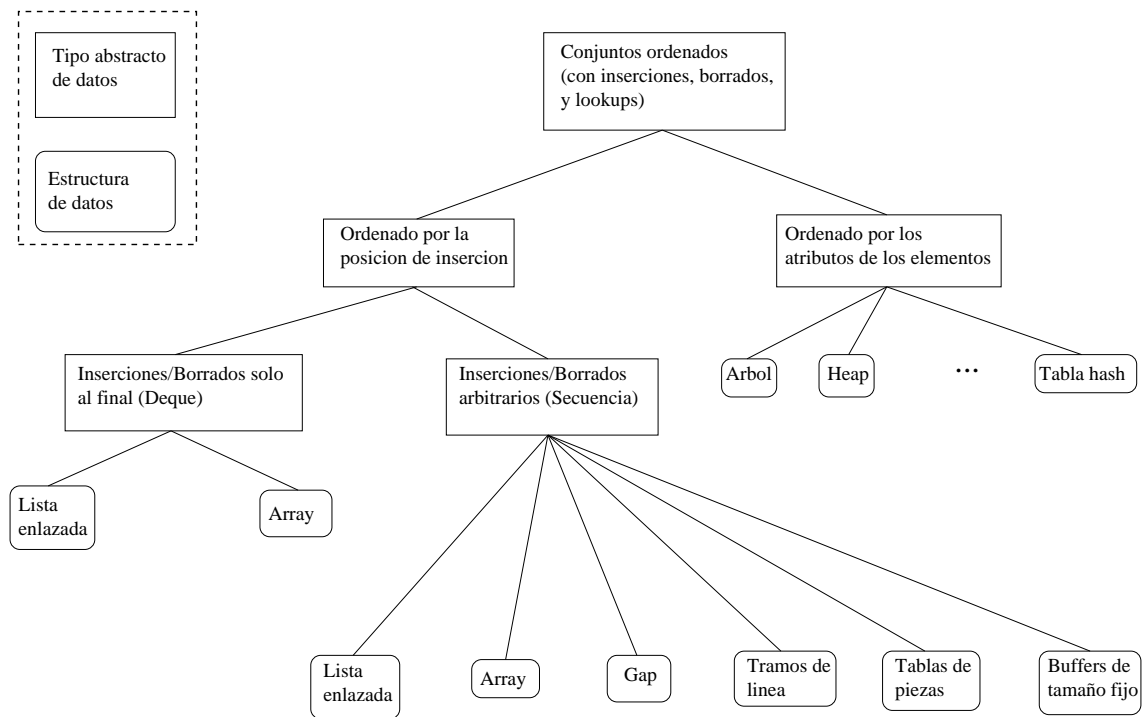


Figura 4.2: Jerarquía de Crowley para conjuntos ordenados

En algunos conjuntos el orden viene determinado por algún atributo intrínseco en los elementos del conjunto. En estos casos las inserciones, borrados y lookups vienen guiados por estos atributos. Es el caso de los árboles binarios de búsqueda y las tablas hash, por ejemplo. El orden establecido entre los elementos de un árbol de búsqueda viene determinado por el valor del elemento. Similarmente, las tablas hash ordenan sus elementos en función del valor retornado por la función hash correspondiente. Es de notar que en estos casos el usuario no especifica de forma explícita el orden de inserción, borrado o lookup.

La otra posibilidad consiste en que el orden dentro del conjunto venga determinado por la posición (expresada de forma explícita) de las operaciones de edición. Por ejemplo, en el caso en que las inserciones y borrados estén limitadas a los extremos del orden tenemos una estructura *deque*. Si es posible insertar o borrar en cualquier posición del conjunto entonces tenemos una **secuencia editable**.

Es de notar una característica muy importante de las secuencias editables presentes en los editores de texto: las operaciones de edición se encuentran muy localizadas en torno a un punto concreto de la secuencia. Este hecho, llamado *principio de la localización de la edición*, resulta determinante en la construcción de las estructuras de datos para secuencias.

En los siguientes apartados se examinan varias estructuras de datos para implementar secuencias editables. Algunas han sido puestas en práctica y, como veremos, todas presentan tanto ventajas como inconvenientes.

### 4.2.2 Vectores

La forma más simple de estructurar una secuencia editable en memoria es mediante la utilización de algún tipo de vector o array. Cada elemento del vector contiene un ítem de la secuencia editable.

El contenido de la secuencia se almacena en el vector en la forma de un solo tramo. Lo habitual es hacer coincidir el comienzo de la secuencia con el comienzo del vector, de modo que la posición 0 de la secuencia se almacena en la posición 0 del vector, y así para cualquier posición  $i$ .

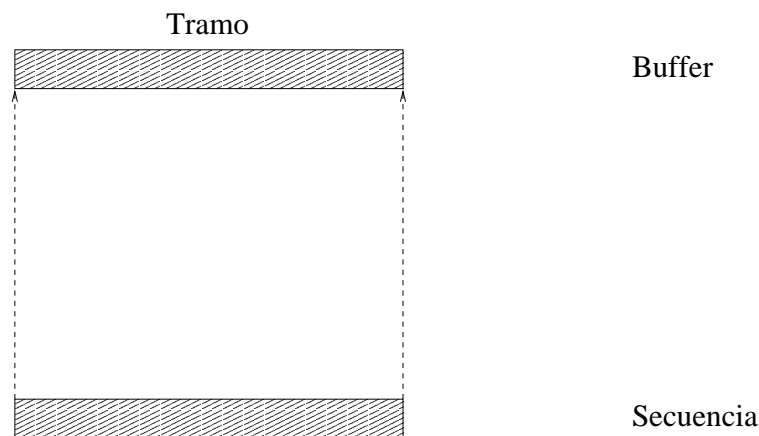


Figura 4.3: Secuencia en un vector

Podemos distinguir dos estrategias distintas en la utilización de esta estructura de datos. La primera consiste en alojar en memoria el espacio justo para albergar la secuencia (véase Figura 4.3). Si bien esta opción es óptima en cuanto a secuencias estáticas o de solo lectura (no susceptibles a cambios) el panorama cambia en cuanto examinamos el impacto de su implementación en las operaciones básicas de edición. El *borrado* se implementa alojando un nuevo vector en memoria del tamaño deseado (menor que el anterior) y a continuación copiando el vector original sin los contenidos borrados. La *inserción*, por su parte, requiere el alojamiento de un nuevo vector del tamaño deseado (mayor que el anterior), y nuevamente la copia del contenido del vector original con el añadido de los nuevos contenidos insertados. Finalmente, el *lookup* resulta muy eficiente: basta un acceso directo al contenido del vector en memoria. Dado que las posiciones de los elementos en memoria coinciden con las posiciones de su almacenamiento en el vector, no se requieren cálculos extra.

Aunque podemos pensar en algunas optimizaciones (como utilizar llamadas al sistema de realojo de memoria en lugar de desalojar y después alojar) en general este esquema es claramente deficiente. Por un lado, requiere de la intervención del gestor de memoria cada vez que se produce una inserción o borrado, lo cual implica sobrecarga. Por el otro, puede producir una alta fragmentación en la memoria del sistema. Debido a todos estos inconvenientes este esquema de almacenamiento de secuencias no se utiliza en la implementación de editores de texto, sino en aplicaciones que pueden garantizar la inmutabilidad de las secuencias, o bien un número de ediciones extremadamente bajo.

La segunda opción consiste en alojar un vector capaz de contener la secuencia original mas un espacio extra al final (véase Figura 4.4). Este espacio extra permite disminuir considerablemente los realojos de memoria durante las operaciones de edición con el precio de tener que mantener cierta información externa acerca del almacenamiento: la longitud del vector realmente ocupado por la secuencia.

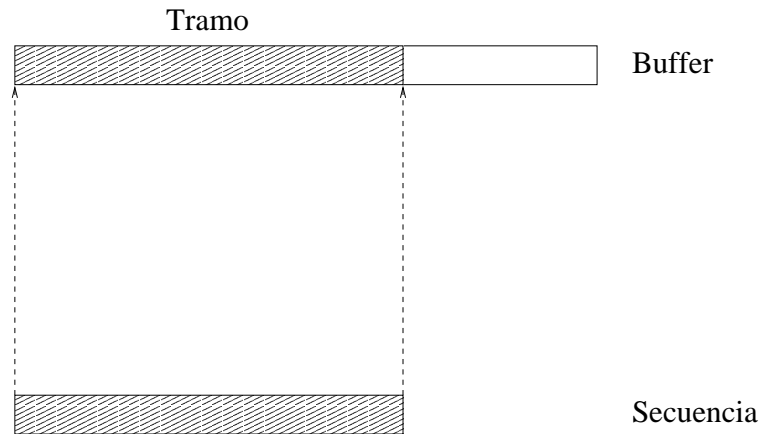


Figura 4.4: Secuencia en un vector con espacio al final

En este esquema con espacio extra al final las operaciones básicas de edición requieren de algo más de lógica, y en concreto de movimiento de elementos dentro del mismo vector. Afortunadamente estas operaciones de movimiento de bloques de elementos pueden implementarse de forma eficiente. Un *borrado* ya no implica un realojo de memoria: simplemente se desplaza el bloque de elementos situados a la derecha para llenar el hueco ocupado por el contenido borrado. Un *lookup* sigue siendo tan eficiente como en el caso del vector de tamaño fijo, si bien ahora es necesaria una comprobación extra: existen posiciones en el vector que no contienen elementos de la secuencia. Finalmente, una *inserción* puede o no requerir de un realojo de memoria. Si queda espacio en el vector para almacenar el nuevo contenido éste simplemente se inserta en la posición deseada, con el consabido desplazamiento de los elementos situados a la derecha de la posición, hacia la derecha. En caso contrario se debe alojar más memoria para el vector.

En estos casos lo habitual es alojar la memoria extra en bloques múltiples del espacio ocupado por un elemento de la secuencia. Finseth menciona 16 octetos como el incremento de tamaño del vector más típico (considerando que un elemento ocupa un octeto, como es el caso de los caracteres codificados en ISO 646). El alojar la memoria en bloque disminuye notablemente la fragmentación.

Este método de estructuración de secuencias en memoria, con su variante de espacio extra al final, fue muy utilizado en la implementación de editores de texto orientados a líneas, en los que la secuencia consistía en no más de 80 o como mucho 100 elementos (la anchura de una línea). En el paso a los editores de texto a pantalla completa (bien orientados a páginas, bien a documentos completos) sus deficiencias comenzaron a hacerse notables: cuanto más grande es la secuencia más ineficientes son los movimientos en bloque de los elementos del vector producidos por las inserciones y los borrados.

No obstante hay que tener en cuenta que estadísticamente es mucho más común editar una línea al final (añadir más contenido o borrarlo) que modificar la línea en alguna parte intermedia o el principio. Esto último corresponde generalmente a las actividades de corrección. Dado que con este esquema la inserción (cuando queda espacio) o el borrado no requieren de movimiento de bloques de elementos dentro del vector, en promedio se trata de un método eficiente en cuanto a edición de líneas se refiere.

### 4.2.3 Buffer gap

Este método consiste en estructurar la secuencia en un buffer dividido en dos tramos (véase Figura 4.5). Entre ambos tramos se sitúa un espacio que no contiene elementos válidos de la secuencia. A este espacio se le denomina *gap* o “agujero”.

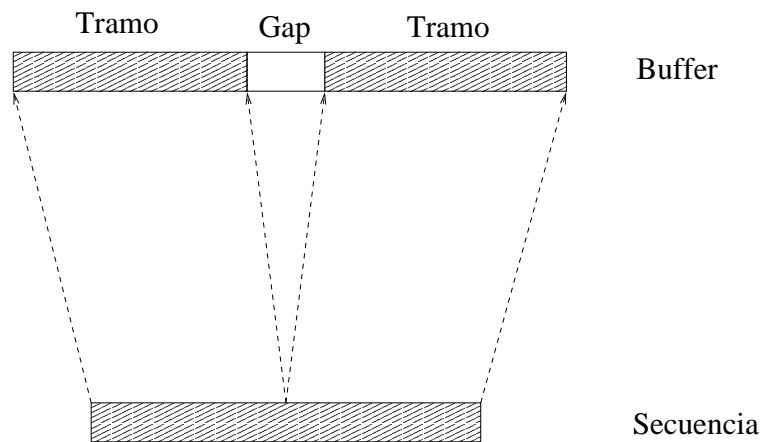


Figura 4.5: Secuencia en buffer gap

El gap no tiene por qué estar en el medio de la secuencia y puede moverse mediante el desplazamiento de bloques de elementos. Dos descriptors (o punteros) sirven para marcar dentro del buffer el comienzo y el final del gap. De esta forma las operaciones básicas de inserción y borrado consisten en:

- Inserción** El gap se mueve de forma que su comienzo coincida con la posición en la secuencia donde se va a realizar la inserción. La operación entonces consiste en copiar el nuevo contenido y avanzar el descriptor del comienzo del gap.
- Borrado** El gap se mueve de forma que su comienzo coincida con la posición en la secuencia donde se va a realizar el borrado. La operación entonces consiste en retroceder el descriptor del comienzo del gap.

Lo que convierte este esquema en uno de los más eficientes por término medio es la existencia del principio de *localidad de la edición*. Este principio empírico (XREF: estadística de Crowley) muestra que un gran porcentaje de las ediciones (inserciones o borrados) se realizan en posiciones consecutivas de la secuencia de texto. Cuando esto ocurre no es necesario mover el gap para situarlo en el punto de edición y por tanto las operaciones consisten en copias en la secuencia y desplazamiento de punteros. No se requiere de desplazamientos de bloques de elementos dentro de la secuencia.

Eventualmente es posible que el gap se llene. Entonces es necesario alojar mas memoria y desplazar el contenido de la secuencia para posicionarlo en su sitio. Esta es la única situación en la que se requiere alojar memoria cuando se utiliza este método.

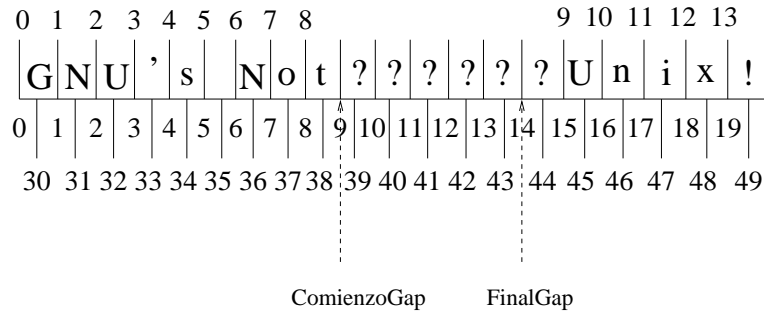


Figura 4.6: Coordenadas en buffer gap

Es posible utilizar tres sistemas de coordenadas distintas a la hora de acceder al contenido de la secuencia (véase Figura 4.6). Esto introduce alguna complejidad extra, ya que las operaciones de inserción, borrado y lookup necesitarán estar preparadas para distinguir entre los tres sistemas.

**Sistema de coordenadas de usuario**

Dibujados encima del buffer, estas coordenadas varían entre 0 (justo antes del primer elemento de la secuencia) y el tamaño de la secuencia. En este sistema el gap es “invisible”. El posicionar las coordenadas entre los caracteres y no en ellos mismos facilita la aritmética y evita la consideración de casos especiales.

**Sistema de coordenadas de gap**

Dibujados justo debajo del buffer, estas coordenadas varían entre 0 hasta el final del buffer. Esto implica que este espacio de coordenadas comprende al gap. Las operaciones internas de gestión del gap utilizan este sistema de coordenadas. Como en el caso de las coordenadas de usuario se numeran las posiciones entre los elementos del buffer, y no ellos mismos.

**Sistema de coordenadas de almacenamiento**

Se trata de la última línea de números de la figura. Es la forma en que las posiciones de memoria son accedidas por el sistema. Siendo  $X$  la dirección de memoria del comienzo del buffer basta utilizar  $X + n$  para hacer referencia al  $n$ -ésimo elemento del buffer (siendo  $n$  el tamaño que ocupa un elemento).

La conversión entre coordenadas de usuario y coordenadas de gap es muy sencilla y puede implementarse de forma eficiente. Si una localización en el sistema de coordenadas de usuario está situada antes del comienzo del gap entonces su correspondiente coordenada de gap coincide. Por otra parte, si la localización de usuario está situada después del comienzo del gap<sup>1</sup> la correspondiente coordenada de gap se calcula como  $(FinalGap - ComienzoGap) + LocalizacionUsuario$ .

---

<sup>1</sup> ¡no del final del gap!

El dividir el espacio de coordenadas de la secuencia en estos tres niveles permite que las facilidades de manejo de la secuencia (como búsqueda, ordenación, etc) utilizan el sistema de coordenadas de usuario para referirse a las posiciones de la secuencia. De esta forma dichas librerías pueden abstraerse del método de estructuración concreto de la secuencia.

Queda una pregunta por hacerse: ¿es tolerable la sobrecarga asociada con el movimiento del gap?. Hay que tener en cuenta que hay situaciones en las que debe moverse **todo** el contenido de la secuencia (cuando se hace una inserción al comienzo y luego otra al final, por ejemplo). Finseth responde a esta pregunta en *The Craft*. Utilizando 1/10 segundos como umbral de percepción de tiempo (tal y como sugieren innumerables estudios) se considera que el editor está funcionando en una máquina dedicada. Si se asumen 250 nanosegundos por ciclo de procesador y una memoria de 16 bits, dado que se requieren 10 ciclos para mover dos palabras en memoria se deduce que se podrán mover hasta 80000 octetos sin que el usuario se percate. Finseth concluye que dado 1) el principio de localidad de la edición y 2) que la mayoría de los ficheros en edición son mucho menores que 80000 octetos, estos casos extremos no representan un problema. Yo creo que la conclusión de Finseth sigue siendo válida en la actualidad, dado que aunque si bien el tamaño de los ficheros ha aumentado desde entonces, también se ha disminuido el tiempo de acceso a memoria y de ciclo de instrucción.

Es interesante notar que el tamaño del gap no incide en la ineficiencia del movimiento del contenido de la secuencia. Esto implica que es posible aumentar el tamaño del gap cuanto se quiera sin con ello aumentar la ineficiencia. En sistemas de memoria virtual paginada esto es de la máxima importancia, ya que es posible reservar grandes rangos de memoria virtual para el espacio del gap sin desperdiciar espacio en memoria utilizado. En esta situación lo óptimo es maximizar en lo posible el tamaño del gap y de ese modo reducir la posibilidad de un realojo de memoria.

El primer editor de texto en utilizar este método fué TECO y es ampliamente utilizado en editores de texto mas modernos, como *vi* y sus evoluciones, y prácticamente todas las variantes de EMACS<sup>2</sup> (como *GNU Emacs*).

Las ventajas que Finseth concede a este método son las siguientes:

- Simple y de fácil implementación.
- De fácil depuración.
- Dado que el texto de cada uno de los dos tramos se almacena de forma continua en el buffer es generalmente eficiente transferir la información desde un fichero: basta una o dos llamadas al sistema.
- Se adecuía perfectamente a las estaciones de trabajo modernas y sus grandes espacios de memoria virtual.

Por su parte, Crowley también avala la simplicidad y la (sorprendente) eficiencia de este método. En general el buffer gap es considerado como el método mas adecuado para cualquier editor de propósito general a no ser que incida alguna circunstancia muy particular (que veremos al examinar otros métodos).

---

<sup>2</sup> Exceptuando las implementadas en entornos Lisp, que utilizan otro método expuesto mas adelante

#### 4.2.4 Listas enlazadas

Este esquema de estructuración utiliza una lista enlazada por punteros para almacenar los tramos, con la peculiaridad de que cada tramo almacena un solo elemento de la secuencia (véase Figura 4.7).

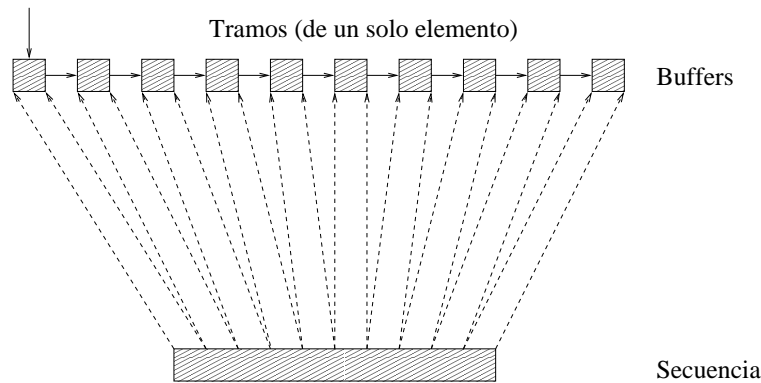


Figura 4.7: Secuencia en lista enlazada

Las operaciones de *inserción* y *borrado* pueden implementarse muy eficientemente en este esquema mediante la mera manipulación de los punteros de la lista. Sin embargo la operación de *lookup* pasa a ser un proceso secuencial de complejidad  $O(n)$ , consistente en el recorrido de la lista hasta obtener la referencia del nodo deseado.

Otras desventajas del método son la gran fragmentación externa de memoria que genera y el espacio invertido en el almacenamiento de los propios descriptores o ficheros.

Debido a lo anterior este método nunca se utiliza para el almacenamiento de secuencias de texto, sino en la estructuración de secuencias de elementos mas complejos, como pueden ser buffers gestionados con buffer gap.

#### 4.2.5 Líneas en tramos

En ocasiones es muy deseable una gran localización de las líneas dentro del contenido de una secuencia. Esto es, el contenido de la secuencia de texto se encuentra estructurado en líneas. En estos casos puede ser útil estructurar la secuencia como una lista de líneas en lugar de como un array monodimensional de elementos.

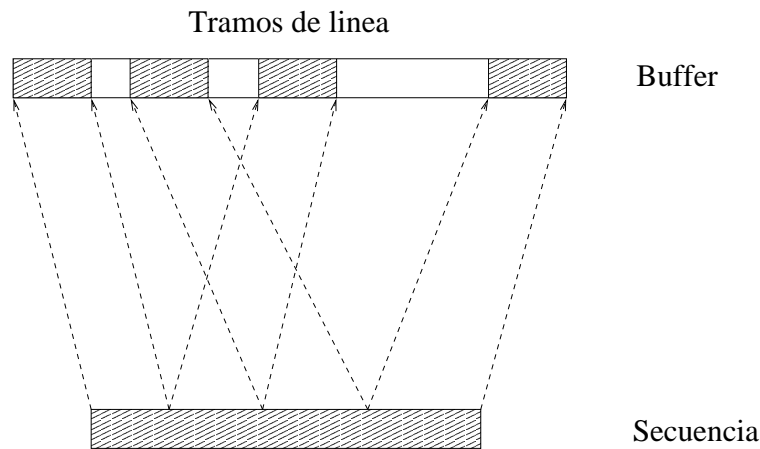


Figura 4.8: Secuencia con líneas en tramos

Cada línea se estructura en un tramo en memoria, y por tanto requiere de un descriptor por línea. A menudo se utiliza un buffer largo para almacenar todas las líneas (véase Figura 4.8). Las nuevas líneas son entonces insertadas al final del buffer, con el consiguiente realojo de memoria.

En este esquema la gestión de líneas resulta muy eficiente. El borrado de líneas consiste en la eliminación de los descriptores, mientras que la inserción consiste en la definición de nuevos descriptores. En cuanto a la gestión de los caracteres de las líneas, ésta requiere de movimientos de bloques de caracteres (como en el caso del método del vector). Dado que no se prevén líneas extremadamente grandes estas ediciones a nivel de carácter no representa un problema en cuanto a eficiencia.

Entre los editores que utilizan este método para estructurar secuencias se incluyen *Ved* (utilizando una lista enlazada para almacenar los descriptores de las líneas) y *Godot*, *Gina* y *ed* (que utilizan un array para almacenar los descriptores).

Una variante muy popular de este método es la utilización de una lista enlazada para almacenar los tramos en lugar de un buffer como en el caso anterior. Comenzó a utilizarse el método de las líneas enlazadas (tal es el nombre) en los editores tipo emacs implementados en entornos Lisp. En este método cada línea se almacena en un nodo de una lista enlazada por punteros. La gestión interna de almacenamiento de cada tramo puede variar, siendo comunes la técnica del vector con espacio al final y el buffer gap. Lo habitual es que la lista de líneas sea doblemente enlazada.

Las operaciones básicas de edición en el método de las líneas enlazadas se implementan de forma inmediata. Las líneas nuevas se insertan en el lugar adecuado de la lista<sup>3</sup>.

Como se ha mencionado este método resulta interesante sobre todo en implementaciones en entornos con buenas capacidades de manejo de líneas (tal y como es el caso de los entornos Lisp). La primera implementación del *DoberSEE Programming Environment* utilizaba este método de almacenamiento para las secuencias de texto.

<sup>3</sup> No se utilizan caracteres para denotar la nueva línea

### 4.2.6 Buffers de tamaño fijo

En ocasiones es necesario o deseable limitar el tamaño de los buffers alojados en memoria. Algunos de los motivos pueden ser:

- Capacidades limitadas del sistema, como es el caso en arquitecturas empotradas.
- Adaptación al sistema de memoria virtual paginada del sistema.
- Capacidad de editar ficheros muy grandes aun en entornos con limitaciones de memoria.

El primer punto hace referencia a arquitecturas muy limitadas como puede ser un dispositivo de mano, una agenda electrónica o un dispositivo Galileo<sup>4</sup>. En estos entornos la cantidad de memoria disponible puede ser pequeña. Por otra parte, el segundo punto hace referencia a los sistemas de memoria virtual paginada, donde es habitual dividir la memoria en buffers del mismo tamaño de la página (típicamente cuatro kilo octetos en arquitecturas intel). Esto propicia una gestión muy eficiente de los buffers por parte del sistema operativo. Finalmente, este método permite editar ficheros muy grandes en sistemas con memoria física limitada sin que el trashing provocado por el swapping se convierta en un problema. Crowley aduce que puede eliminar la dependencia con el gestor de memoria virtual fijando el tamaño de los buffers a un múltiplo del tamaño de bloque de disco.

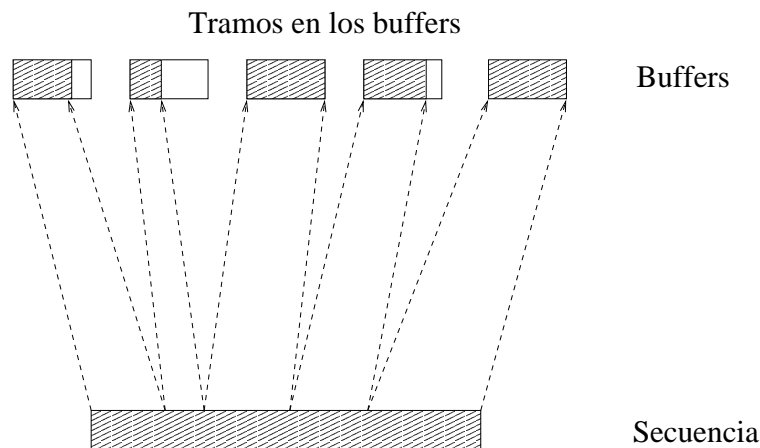


Figura 4.9: Secuencia en buffers de tamaño fijo

La estructura es muy simple. Se alojan tantos buffers de tamaño fijo (determinado en base a criterios como los anteriores) y se utilizan descriptores para almacenar partes de la secuencia en ellos (véase Figura 4.9). En la gestión interna de cada buffer se puede utilizar cualquier esquema de estructuración (habitualmente el buffer gap). Esto implica que en ocasiones hay que mover información entre los buffers y en ocasiones juntar varios buffers en uno solo (cuando el contenido de uno de ellos sea muy pequeño en comparación con su tamaño). Esto evita la proliferación de muchos buffers con poco contenido que conllevaría la aparición de varios problemas:

- Fragmentación interna (espacio malgastado en los buffers).

<sup>4</sup> Sistema europeo de posicionamiento global

- La lista de descriptores se hace mas larga cuanto mas buffers se utilicen para almacenar la secuencia.
- La probabilidad de que una operación compuesta de edición esté confinada a un solo buffer se ve reducida.

Finseth hace un análisis de la utilización del buffer gap para estructurar el contenido de cada buffer de tamaño fijo. Identifica dos grandes ventajas:

- Dado que cada buffer es razonablemente pequeño el gap nunca debe moverse muy lejos y por tanto los bloques de texto que deben desplazarse en el buffer no es muy grande.
- Dado que los buffers son de tamaño fijo la gestión de memoria es simple.

Estas dos cualidades son muy deseables en sistemas de recursos limitados. Un buen ejemplo de aplicación lo encontramos en el editor *Mince*, preparado para funcionar en sistemas CP/M con 48 Kilo octetos de memoria y pequeños disquetes. Este editor implementaba todo un sistema de memoria virtual paginada para gestionar los buffers de tamaño fijo.

Este método es también utilizado en los editores *Gina* y *sam*.

#### 4.2.7 Tablas de piezas

Este método es considerablemente mas complejo que los presentados anteriormente y puede considerarse como el intento mas serio de presentar una alternativa al método del buffer gap.

Para almacenar la secuencia editable se utilizan los siguientes componentes (véase Figura 4.10):

##### Un buffer primario

Este buffer es de **solo lectura** y de **tamaño fijo**. Contiene la secuencia tal y como se ha podido leer de un fichero.

##### Un buffer secundario

Este buffer es de **solo añadidos** (*append-only*) y puede crecer arbitrariamente. El nuevo contenido producido por las operaciones de edición se introducen en este buffer.

##### Una tabla de descriptores de piezas

Como veremos el contenido de la secuencia se va partiendo en subsecuencias denominadas **piezas**. Esta tabla contiene descriptores de las piezas, que pueden estar almacenadas en cualquiera de los buffers anteriores.

Esta es la **tabla de piezas** que da nombre al método.

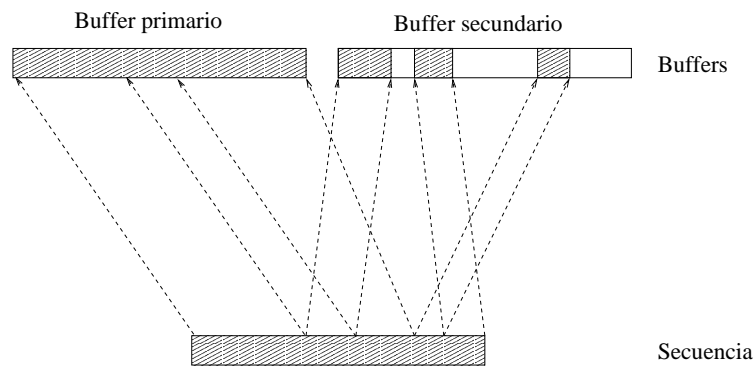


Figura 4.10: Secuencia en tabla de piezas

Cada uno de los descriptores de la tabla de piezas define un tramo en el buffer primario o secundario. Luego un descriptor debe contener tres campos de información (véase Figura 4.11):

- Qué buffer contiene la pieza (un valor booleano para seleccionar el buffer primario o bien secundario).
- El offset dentro de dicho buffer que identifica el comienzo del tramo (un entero no negativo).
- La longitud del tramo (un entero positivo<sup>5</sup>).

Buffer	Offset	Length
b0	off0	len0
⋮		
bn	offn	lenn

Figura 4.11: Estructura de la tabla de piezas

Inicialmente existe una única pieza que abarca todo el buffer primario. Las modificaciones realizadas en la tabla de piezas y ambos buffers resultado de las operaciones básicas de edición se describen a continuación.

---

<sup>5</sup> Las piezas de tamaño cero son eliminadas.

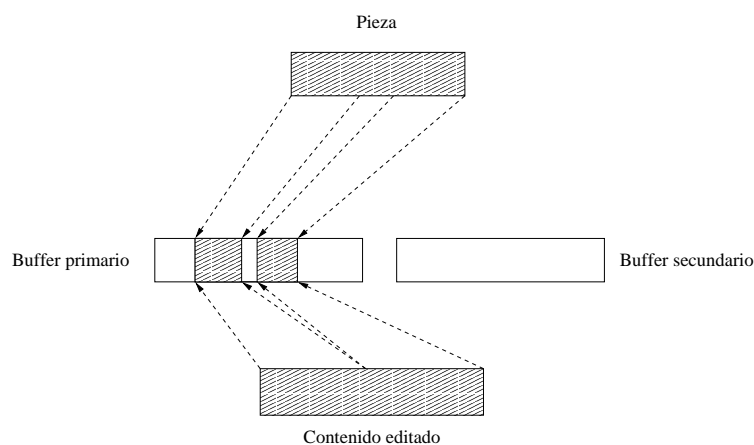


Figura 4.12: Operación de borrado en tabla de piezas

### Borrado

Un borrado en el interior de una pieza provoca que ésta se parta en dos. Una de las nuevas piezas abarca los elementos de la pieza original hasta la posición del borrado y la otra abarca los elementos desde el final de la región borrada hasta el fin de la pieza original.

Si el elemento borrado está situado en el comienzo o el final de la pieza simplemente ajustamos el offset o la longitud, según proceda.

### Inserción

Una inserción en el interior de una pieza provoca que ésta se parta en tres. La primera pieza abarca los elementos de la pieza original desde el comienzo hasta el punto donde se ha insertado el elemento. El elemento insertado se almacena en la segunda pieza al final del buffer secundario. Finalmente, la tercera pieza abarca los elementos de la pieza original desde el punto de inserción hasta el final de la misma.

De nuevo aparecen casos especiales si la inserción se hace al comienzo o al final de una pieza.

A efectos de optimización, si varios elementos son insertados en fila se combinan en una sola pieza en el buffer secundario.

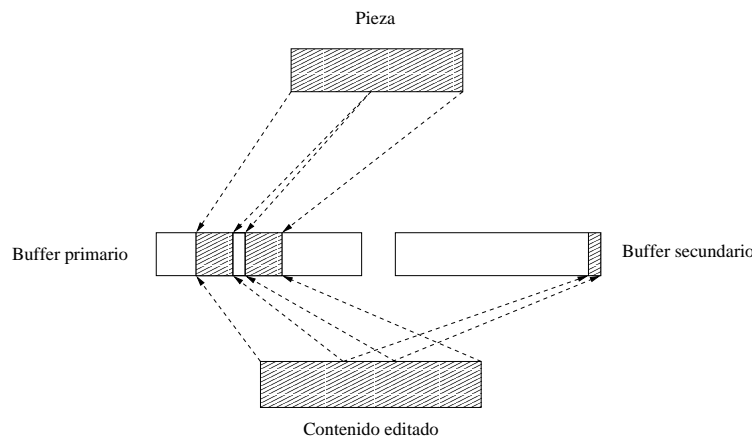


Figura 4.13: Operación de inserción en tabla de piezas

Crowley identifica las siguientes ventajas de este método de estructuración de secuencias:

- El buffer primario es estático y por tanto puede almacenarse en memoria como un buffer de solo lectura. En el caso de leer la secuencia de un fichero puede mapearse en memoria virtual en modo solo lectura, y por tanto ser compartido por varios buffers.
- Dado que el buffer secundario es de solo añadidos la parte ya escrita nunca cambia.
- Los elementos nunca se mueven una vez insertados en cualquiera de los dos buffers.
- La implementación de la operación **undo** (*deshacer*) se convierte en algo inmediato. Dado que los contenidos nunca se borran es cuestión de mantener los punteros adecuados.
- No es necesaria ninguna fase de preprocesado del fichero que contiene la secuencia a editar. Una sola llamada al sistema basta para almacenar la secuencia en el buffer primario.
- La cantidad de memoria utilizada para almacenar la secuencia editable *es función del número de ediciones y no del tamaño de la secuencia*. Esto implica que la edición en ficheros muy largos es muy eficiente en término de almacenamiento.

El método de las tablas de piezas ha sido utilizado en los editores **Bravo**, **Lara** (extendiendo el sistema Bravo para soportar semántica), **Point** y **Pastiche**.

## 4.2.8 Evaluación de los métodos

### 4.2.8.1 Comparativa de Finseth

En su tesis *The Craft of Text Editing* Finseth realiza una comparativa entre los varios sistemas de almacenamiento de secuencias que propone (vectores sin espacio extra, vectores con espacio al final, buffer gap, líneas enlazadas con espacio al final y buffers de tamaño fijo). En concreto no contempla el método de las tablas de piezas (posterior a la última edición de *The Craft*).

El estudio consiste en un exámen de la adecuación de los métodos a criterios como el almacenamiento utilizado, la capacidad de recuperación de errores, la eficiencia en la edición, etc. A diferencia de Crowley no utiliza estadísticas para apoyar sus afirmaciones.

### Espacio de almacenamiento

Se considera el espacio de almacenamiento que requiere cada uno de los métodos para albergar un buffer. Se asumen caracteres de 8 bits. El buffer a almacenar consiste en 150 líneas con 60 caracteres en cada una (un total de 9059 caracteres contando los fin-de-línea).

Una implementación **buffer gap** requeriría una cabecera de tamaño fijo (8 octetos) además de un octeto por cada carácter. Esto incluye los caracteres fin-de-línea. En total, 9068 octetos.

Una implementación de **líneas enlazadas** requeriría una cabecera de tamaño fijo para la lista (8 octetos) mas una cabecera de tamaño fijo para cada línea (12 octetos) mas un octeto por cada carácter. A esto hay que sumarle un promedio de fragmentación interna en cada línea de 8 octetos (utilizando el método de espacio al final). Hay que tener en cuenta que en este método no se almacenan los caracteres fin de línea. En total,  $8 + 150 * 12 + 9000 + 8 * 150 = 12008$  octetos.

Una implementación de **buffers de tamaño fijo** con gestión interna de buffer en buffer gap requeriría una cabecera fija para la lista de buffers (8 octetos) mas una cabecera de tamaño fijo para cada buffer (12 octetos) mas los buffers en si mismos (4 kilo octetos cada uno, para ajustarse al tamaño de página<sup>6</sup>). En total,  $8 + 12 * 5 + 4096 * 5 = 20548$  octetos.

Método	Espacio de almacenamiento requerido (en octetos)
Buffer gap	9068
Líneas enlazadas	12008
Buffers de tamaño fijo	20548

Tabla 4.1: Comparación Finseth en almacenamiento requerido

Las conclusiones que pueden extraerse del estudio son claras (véase Tabla 4.1). El método de las líneas enlazadas paga un alto precio en requisitos de almacenamiento debido a la sobrecarga de información por cada línea (que además depende del contenido en concreto del buffer). El método de los buffers fijos sufre de fragmentación interna al ajustar el tamaño de los buffers al tamaño de página (que no suele ser pequeño). No obstante una correcta gestión de la mezcla de buffers podría controlar la fragmentación no permitiendo la infrutilización de los buffers.

El método que claramente presenta una utilización mas eficiente de la memoria es el buffer gap.

### Recuperación de errores

Esta comparación asume que la ejecución de un editor de texto ha sido interrumpida por un error fatal. El sistema operativo ha salvado en disco una imagen del espacio de memoria utilizado por el programa en el momento del

<sup>6</sup> Finseth utiliza aquí el tamaño de página de 2 kb, mas común en aquella época.

error<sup>7</sup>. Se trata de analizar la adecuación de los esquemas de almacenamiento a la recuperación del contenido de los buffers del editor utilizando un depurador sobre la imagen de memoria en disco.

En una implementación **buffer gap** la recuperación de la información resulta relativamente sencilla y segura. Una vez localizado el comienzo y el final del buffer (buscando una secuencia muy larga de caracteres ASCII imprimibles) el usuario puede borrar el espacio ocupado por el gap de forma manual, o bien automáticamente si el programa mantiene marcas especiales al comienzo y al final del gap. Esta facilidad proviene del hecho de que prácticamente todo el contenido de la secuencia se encuentra almacenada secuencialmente en memoria.

En una implementación de **líneas enlazadas** la cosa se complica bastante. La recuperación básicamente consiste en tratar de encontrar cabeceras de líneas de forma secuencial. Una vez se encuentra una ya se tiene acceso a las demás líneas del buffer mediante el uso de los punteros de las cabeceras (línea anterior y siguiente línea). Esta técnica se basa por una parte en que el editor borre la memoria liberada (rellenada con ceros, por ejemplo) y por otra en que el formato de las cabeceras sea fácilmente reconocible (mediante un número mágico, por ejemplo). En caso contrario la localización de las cabeceras puede complicarse.

En una implementación de **buffers de tamaño fijo** la recuperación resulta mas compleja que en el caso del buffer gap pero mucho mas simple que en el caso de las líneas enlazadas. La localización de un buffer se realiza igual que en el primer caso, y la obtención del resto se hace mediante seguimiento de punteros, como en el segundo (asumiendo que los descriptores correspondientes a los buffers se almacenen en una lista enlazada por punteros).

<b>Método</b>	<b>Dificultad de recuperación</b>
Buffer gap	Baja
Buffers de tamaño fijo	Moderada
Líneas enlazadas	Alta

Tabla 4.2: Comparación Finseth en recuperación de errores

Luego el sistema de almacenamiento mejor dotado para facilitar la recuperación de la información en una situación de error es el buffer gap, siguiéndole los buffers de tamaño fijo y las líneas enlazadas en orden creciente de dificultad (véase Tabla 4.2).

### **Eficiencia en edición**

La siguiente tabla muestra el esfuerzo requerido por cada uno de los métodos a la hora de insertar un carácter o una línea en una posición arbitraria de la secuencia almacenada.

---

<sup>7</sup> Una imagen core.

	<b>Insertar carácter</b>	<b>Insertar línea</b>	<b>Max. movimiento</b>
<b>Buffer gap</b>	mover gap actualizar puntero	igual que carácter	buffer
<b>Líneas enlazadas</b>	mover contenido línea actualizar puntero	alojar nueva línea introducir en lista	
<b>Buffers de tamaño fijo</b>	si lleno, partir página mover gap actualizar puntero	igual que carácter	buffer

Tabla 4.3: Comparación Finseth en la eficiencia de edición

Como era de esperar el buffer gap resulta el esquema mas eficiente aunque a veces genere pausas relativamente grandes (cuando gran cantidad del buffer debe moverse). El esquema de líneas enlazadas introduce mucha sobrecarga en la gestión de la lista. Por último, el esquema de buffers fijos (gestionados internamente con buffer gap) elimina las pausas ocasionales de movimiento de grandes cantidades del buffer pagando el precio de una gestión mas elaborada.

#### **Eficiencia en E/S de Buffer $\Leftrightarrow$ Fichero**

Es razonable esperar que en algun momento un editor requiera utilizar las facilidades de E/S del sistema operativo para leer el contenido de un fichero (a efecto de almacenarlo en una secuencia editable) o bien escribir el contenido de una secuencia en un fichero. Cada uno de los métodos a estudio presentan ventajas e inconvenientes a este efecto.

Desde el punto de vista de la E/S el método del **buffer gap** es extremadamente eficiente<sup>8</sup>. La lectura del contenido de un fichero y su almacenamiento en un buffer requiere de las siguientes operaciones:

- Determinar el tamaño del fichero.
- Alojarse la suficiente memoria para almacenar el contenido del fichero, mas espacio extra para el gap.
- Leer el contenido del fichero.

Si consideramos que inicialmente el gap está al comienzo o al final del buffer entonces basta una sola llamada al sistema para leer el contenido del fichero y almacenarlo en el buffer. En algunos sistemas este esquema incluso puede mejorarse haciendo uso de las facilidades de la memoria virtual. En efecto, muchos sistemas operativos permiten mapear ficheros en el espacio de memoria virtual. Las páginas correspondientes al rango en el que se mapea el fichero se fijan en modo de *copy on write*.

En cuanto a la escritura del contenido del buffer en un fichero generalmente se requiere de dos llamadas al sistema: una para almacenar el contenido del buffer

<sup>8</sup> De hecho a veces este aspecto es mencionado como una de sus mayores virtudes.

hasta el gap y otra para almacenar el contenido del buffer desde el gap hasta el final. Podría pensarse en optimizar esta operación moviendo el gap al comienzo o al final del buffer y así requerir de únicamente una llamada al sistema. Pero esto último rompería el principio de localidad en el buffer penalizando cualquier edición realizada después de la escritura.

Por su parte el método de las **líneas enlazadas** dispone del obvio pero extremadamente pobre algoritmo que se describe a continuación:

1. Abrir fichero para leer (llamada al sistema).
2. Mientras queden líneas en el fichero
  1. Leer línea (llamada al sistema).
  2. Alojarse línea (llamada al sistema).
  3. Incluir la línea en la lista enlazada.
  4. Cerrar fichero (llamada al sistema).

Este algoritmo supone utilizar una llamada al sistema de E/S por cada línea que contenga el fichero. Una mejora podría consistir en leer todo el contenido del fichero en un buffer temporal y a partir del mismo construir las líneas.

Por último, el método de los **buffers de tamaño fijo** podría implementarse de modo que la E/S fuera tan eficiente como en el caso del buffer gap. Esto requeriría el leer el fichero en un solo buffer en memoria y utilizar los descriptores para dividirlo de forma lógica en los buffers de tamaño fijo. Debe tenerse en cuenta que esto último implica que la primera escritura en el buffer siempre provocaría una partición del buffer en cuestión.

### Eficiencia en búsqueda

Dada una secuencia editable estructurada en  $n$  piezas y un algoritmo de búsqueda de cadenas, el procedimiento de búsqueda adaptada a la secuencia es el siguiente:

- Por cada pieza ( $n$  veces)...
  - Aplicar algoritmo de búsqueda.
  - Componer resultado y ajustar búsqueda.

El algoritmo de búsqueda es arbitrario y puede estar más o menos adaptado al esquema interno de almacenamiento de cada pieza. Aparte de la eficiencia propia de cada algoritmo de búsqueda (fuerza bruta dará índices pobres de rendimiento, mientras que *Boyer-Moore-Horspool* puede ser muy eficiente) es claro que el mayor impacto de la estructuración de la secuencia incide en el número de piezas en las que aplicar el algoritmo. Un gran número de piezas implica un gran número de iteraciones del bucle, y por tanto introduce un factor multiplicativo en la eficiencia asintótica del procedimiento de búsqueda.

La presencia de muchas piezas plantea otro problema: es necesaria la introducción de un procedimiento de *composición de resultado* que a partir del resultado de las búsquedas en cada pieza componga un resultado de búsqueda referente a toda la secuencia. Por ejemplo, supongamos que una cadena buscada está partida entre dos piezas distintas. En este caso habrá que ajustar la búsqueda y componer el resultado.

De todo lo anterior puede inferirse que cuanto menos piezas utilice un método de almacenamiento mayor será la eficiencia del procedimiento de búsqueda.

El método del **buffer gap** requiere dos iteraciones del bucle: una para el exámen de la primera porción del buffer (desde el comienzo hasta el gap) y otra para el exámen de la segunda porción (desde el final del gap hasta el final del buffer). Podría parecer adecuado mover el gap al comienzo o final del buffer antes de cualquier operación de búsqueda, y de este modo ahorrar una iteración. Finseth advierte que esto no funcionaría debido a que las operaciones de búsqueda son *muy* comunes. Por ejemplo, la operación **avanzar una palabra** puede requerir hasta dos búsquedas. Eso representaría dos movimientos de gap con el trasiego que eso conlleva.

Por su parte el método de **líneas enlazadas** presenta el inconveniente de que fragmenta la secuencia en muchas piezas: una por cada línea presente en la secuencia. Por ello requiere de muchas iteraciones. Otro problema asociado a este método es que los caracteres de fin de línea no se encuentran en los buffers propiamente dichos, sino que su presencia es implícita en cada pieza conteniendo una línea. De este modo se complica la búsqueda de patrones que contienen el carácter fin de línea.

Finalmente el esquema de **buffers fijos** está en un punto intermedio: genera mas piezas que el buffer gap pero menos que las líneas enlazadas.

#### **Adecuación a sistemas con memoria virtual paginada**

En este apartado se analiza el comportamiento de los métodos de almacenamiento en sistemas de memoria virtual paginada. En concreto se analiza la situación donde la memoria está llena y por tanto se produce paginación.

Por norma general el método del **buffer gap** trabaja bien en este tipo de situaciones. Su forma de almacenamiento compacto permite almacenar grandes porciones del buffer en pocas páginas. Su organización secuencial conlleva a una buena localidad de referencia y por tanto la mayor parte de los accesos se hacen a páginas cercanas. Esto disminuye la ineficiencia procedente de movimientos de traslación de la cabeza lectora del disco.

No obstante el mayor problema que presenta el buffer gap consiste, como era de esperar, en las situaciones donde se debe mover el gap largas distancias. Cuando la memoria física está llena esto implica que *todas* las páginas que componen el buffer deben moverse a disco y generalmente leerse del disco despues.

Por lo general el problema anterior no es lo suficientemente significativo como para desmerecer las ventajas, y por tanto el buffer gap es considerado un buen método de estructuración de secuencias en entornos de memoria virtual paginada.

El método de las **líneas enlazadas** presenta muchas desventajas y ninguna ventaja real en este tipo de situaciones. La dispersión de los distintos buffers que almacenan las líneas (proviniente de la aleatoriedad de los alojamientos) se traduce en una utilización muy poco eficiente del sistema de memoria virtual. Dado que los sistemas de páginas basan su eficiencia en el principio de localización (sobre todo las cachés de páginas) la dispersión de los buffers genera muchas faltas de páginas y por tanto mucho trasiego en el disco.

El método de los **buffers fijos** puede adaptarse muy bien a estas situaciones. Cuando el tamaño fijo de los buffers es un múltiplo del tamaño de página del sistema (método que Finseth denomina **Buffer gap paginado**) una inserción o borrado involucra como mucho a dos páginas. Además las distancias de movimiento de los gaps disminuyen considerablemente al estar limitados a cada uno de los buffers.

### Conclusiones

Las conclusiones de Finseth son claras y rotundas:

- Debe utilizarse el método del **buffer gap** siempre sea posible.
- Solo debe utilizarse el método de las **líneas enlazadas** si la plataforma de implementación trabaja especialmente bien con listas. Este sería el caso si se implementara en un entorno Lisp o Tcl, por ejemplo.
- Solo debe utilizarse el método de los **buffers fijos** si las provisiones de memoria son precarias. Evidentemente el tamaño de los buffers debe estar ajustado a un múltiplo del tamaño de la página.

### 4.2.8.2 Comparativa de Crowley

En su artículo *Data Structures For Text Sequences* Charles Crowley realiza una comparación experimental de los métodos de estructuración de secuencias que expone, incluyendo los vectores de tamaño fijo, listas enlazadas, líneas enlazadas, buffer gap, buffers de tamaño fijo y tabla de piezas.

Para ello implementó un simulador de operaciones de edición cuyo funcionamiento dependía de un número de parámetros en los que basaba la generación estadística de las operaciones (inserciones y borrados). Los parámetros utilizados en su estudio fueron:

- Longitud inicial de la secuencia: 8000 caracteres.
- Tamaño de bloque de 1024 caracteres.
- Si se simula el método de los buffer fijos el tamaño de éstos siempre se mantienen utilizados al menos en un 50% (fragmentación interna).
- La localización del 98% de las operaciones de edición se distribuye alrededor del último punto de inserción como una normal de desviación estándar de 25.
- La localización del 2% de las operaciones de edición se distribuye uniformemente sobre toda la secuencia.
- Después de cada edición los 25 caracteres anteriores y posteriores son accedidos.
- Cada 250 ediciones se accede a todos los elementos de la secuencia de forma secuencial.

Los métodos de estructuración de secuencias editables comparados son los siguientes:

<b>null</b>	Este método no realiza ningún trabajo. Sirve para medir el tiempo invertido en las invocaciones a los procedimientos que conforman el api.
<b>Arr</b>	El método array.
<b>List</b>	El método de la lista enlazada por punteros.
<b>Gap</b>	El método del buffer gap.
<b>FsbA</b>	El método de buffers de tamaño fijo utilizando el método array para gestionar el contenido interno de cada buffer.

<b>FsbAOpt</b>	El método de buffers de tamaño fijo utilizando el método array para gestionar el contenido interno de cada buffer además de algunas optimizaciones para la operación de lookup.
<b>FsbG</b>	El método de buffers de tamaño fijo utilizando el método del buffer gap para gestionar el contenido interno de cada buffer.
<b>Piece</b>	El método de la tabla de piezas.
<b>PieceOpt</b>	El método de la tabla de piezas con algunas optimizaciones para la operación de lookup.

Los resultados completos de las simulaciones (y las inferencias estadísticas realizadas) no se incluyen en este libro<sup>9</sup>. Sin embargo comentaremos las conclusiones prácticas a las que llega Crowley.

En referencia a la operación de edición **lookup** es claro que el método **Arr** es el mas rápido (casi tanto como el método **null**). El método **FsbA** es mucho mas lento y el tiempo que invierte es parecido al de los métodos **List**, **FsbG** y **Piece**. Por su parte el método optimizado **FsbA** es casi tan rápido como **Gap**. Finalmente el método **PieceOpt** es casi tan rápido como **FsbA**. La conclusión de Crowley es que, dado que la operación de lookup es muy comun, todos los métodos deben optimizarla utilizando técnicas de *caching* exceptuando a **Gap** y **Arr**.

En cuanto a la operación **insertar** el método **List** se revela como el mas rápido. Los métodos **FsbG**, **Gap** y **Piece** consumen aproximadamente el doble de tiempo. Los grandes perdedores son **FsbA** con un orden de magnitud mas lento que los otros métodos y **Arr** con dos órdenes de magnitud.

Uno de los aspectos mas interesantes del estudio realizado por Crowley es la reacción en términos de eficiencia de los diversos métodos a medida que el principio de localidad de la edición va disminuyendo. En términos del experimento esto implica un aumento de la desviación típica en la distribución normal de las ediciones. En concreto se examina el impacto en la operación de **insertar**. Únicamente los métodos **Gap** y **FsbG** parecen afectados por la disminución de la localización, y esto solo en aumentos de la desviación típica mucho mas grandes que lo que cabría esperar en una edición normal.

Las conclusiones que extrae Crowley de las estadísticas se resumen a continuación.

- El método **Arr** disfruta del lookup mas rápido pero sufre de tiempos intolerables en inserciones y borrados.  
No es un método práctico.
- El método **Gap** es casi tan rápido en lookup como **Arr** y bastante eficiente en las inserciones y los borrados.
- El método **List** permite las inserciones y borrados mas rápidos pero el tiempo invertido en los lookups es terrible. Además consume mucha memoria debido a los descriptores de los nodos de la lista.
- El método **FsbA** es lento para inserciones y borrados. Sin embargo el lookup puede hacerse relativamente rápido cacheando las peticiones.

---

<sup>9</sup> Lea el lector el paper de Crowley “Data Structures for Text Sequences”

- El método **FsbG** mejora radicalmente los tiempos de inserción y borrado respecto a **FsbA**. Sin embargo el tiempo de lookup es un poco mas alto. Las posibles optimizaciones son mas complejas de implementar.
- El método **Piece** tiene muy buenos tiempos de inserción y borrado (solo un poco mas bajos que **List**). Sin embargo el tiempo de lookup es muy lento aun con optimizaciones simples.

Luego la optimización del lookup es crucial para hacer práctica la utilización de las tablas de piezas.

La Tabla 4.4 resume las conclusiones de Crowley respecto a las características de los métodos de estructuración de secuencias. Se observará que coinciden en gran medida con las de Finseth expuestas en el apartado anterior.

	<b>Array</b>	<b>Gap</b>	<b>Lista enlazada</b>
<b>Tiempo</b>	Lento	Rápido	Rápido
<b>Espacio</b>	Bajo	Bajo	Muy alto
<b>Facilidad de programación</b>	Fácil	Fácil	Fácil
<b>Tamaño del código</b>	Bajo (39 líneas)	Bajo (59 líneas)	Medio (79 líneas)
	<b>FSB-Array</b>	<b>FSB-Gap</b>	<b>Piece</b>
<b>Tiempo</b>	Muy rápido	Rápido (con caching)	Muy rápido (con caching)
<b>Espacio</b>	Bajo	Bajo	Bajo
<b>Facilidad de programación</b>	Difícil	Difícil	Medio
<b>Tamaño del código</b>	Mediano a alto (218 líneas)	Alto (301 líneas)	Mediano (162 líneas)

Tabla 4.4: Comparación de Crowley

### 4.3 Estructuración externa

Una vez vistas las posibilidades existentes para la estructuración interna del buffer (esto es, para la gestión de la secuencia editable que compone los contenidos del buffer) ya estamos en situación de investigar abstracciones útiles para dotar de estructura dicha secuencia y así ofrecer un componente buffer rico en semántica y, lo que es mas importante, versátil en orden a satisfacer las necesidades de otros componentes del editor.

Es de notar que casi todas las abstracciones expuestas en este apartado, en cuanto asociadas a buffers de texto en editores, no se han contemplado previamente en la literatura como tales. Mientras que Finseth contempla las localizaciones, el puntero, las marcas y los modos, en GNU Emacs podemos encontrar los conceptos de Widening y de Narrowing. Desgraciadamente ninguna de dichas fuentes identifican estos conceptos como abstracciones de estructuración del buffer, sino como *facilidades* que involucran buffers de texto directa o indirectamente.

A la hora de determinar qué abstracciones son útiles para estructurar un buffer es necesario examinar en qué modo utilizan los componentes buffers los editores de texto existentes.

Mientras que todos los algoritmos de refresco de pantalla (o redisplay) estructuran el contenido de los buffers de una forma u otra a efectos de realizar un refresco lo mas eficiente posible, los editores de estructuras dependen de una estructuración muy intensa en orden a mantener los árboles de sintáxis y otra información de estado.

Todas estas abstracciones deben implementarse mediante la interfaz ofrecida por la secuencia editable del buffer. A partir de este momento nos referiremos como *usuario* a cualquier entidad que acceda al buffer.

### 4.3.1 Localizaciones

Una *localización* es el mecanismo mediante el cual el usuario hace referencia a una posición dentro del buffer y por ende al elemento almacenado en dicha posición.

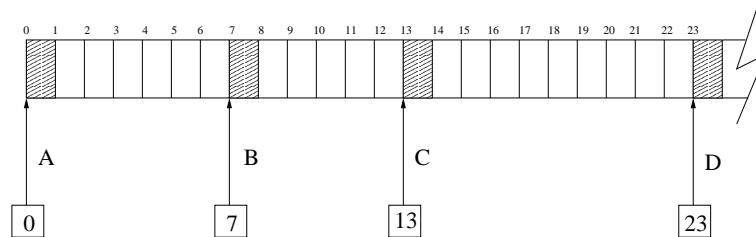


Figura 4.14: Localizaciones en una secuencia

Lo habitual es caracterizar cada posición del buffer con un número entero, a efectos de posibilitar operaciones aritméticas. Normalmente el orden de las localizaciones comienza por cero.

Una característica importante de las localizaciones es que no hacen referencia a las posiciones de la secuencia, sino al espacio entre ellas. De esta forma la localización 0 hace referencia al espacio presente antes del primer elemento, y en general cualquier localización  $i$  hace referencia al espacio entre los elementos  $i - 1$  e  $i$ .

Pero, ¿a cual de los elementos ( $i - 1$  o  $i$ ) hace referencia la localización  $i$ ? Es evidente que hay que asignar un *peso* a las localizaciones. Si el peso es a izquierdas la localización  $i$  hará referencia al elemento  $i - 1$  de la secuencia, mientras que si el peso es a derechas se tratará del elemento  $i$ .

La elección del peso resulta bastante arbitraria. A efectos de esta exposición siempre se asumirá peso a derechas, que es la mas comun en las implementaciones y la literatura (véase Figura 4.14).

Es importante resaltar que el sistema de localizaciones no tiene nada que ver con cualquier sistema o sistemas de coordenadas utilizado en la implementación interna de la secuencia editable. Las localizaciones siempre presentan un sistema de coordenadas uniforme sobre el contenido del buffer.

### 4.3.2 Punteros y marcas

A menudo resulta útil recordar o almacenar ciertas localizaciones dentro de un buffer bajo edición. Para ello se utilizan las *marcas*. El usuario puede definir una marca en cualquier localización del buffer y, posteriormente, moverla cambiando dicha localización. A menudo las

marcas incluyen mas información aparte de la localización, como su tipo u otra información de estado.

En la literatura suelen distinguirse entre dos tipos de marcas:

### Marcas regulares

Estas marcas se asocian a elementos concretos mas y no a localizaciones fijas. Esto implica que, una vez fijada una marca en la localización  $i$  cualquier operación de borrado o inserción hará que dicha localización se actualice.

De esta forma podemos marcar el comienzo de cierto contenido (una palabra, una línea, etc). Estas marcas son dependientes de la edición.

### Marcas estacionarias, o sticky

Por el contrario las marcas estacionarias jamás cambian su localización si no es explícitamente.

Un tipo especial de marca es el *puntero*. Generalmente solo hay uno por cada vista del buffer. El puntero identifica la localización del buffer donde se realizará la siguiente operación de edición. A menudo una representación visual del puntero (denominada *cursor*) se situa sobre el elemento afectado por el mismo. Como en el caso de las marcas puede interesar asociar al puntero otra información aparte de la localización actual: modo inserción o desplazamiento, etc.

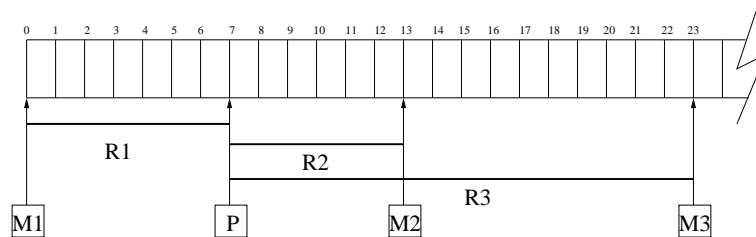


Figura 4.15: Regiones definidas por marcas

La combinación formada por el puntero y las marcas a menudo se utiliza para determinar *regiones* dentro del buffer (véase Figura 4.15). Cada marca define una región compuesta por los elementos del buffer situados entre ésta y el puntero. En los editores que soportan marcas las operaciones de edición básicas orientadas a elementos se completan con operaciones dirigidas a regiones (borrar región, copiar región, etc).

### 4.3.3 Tramos

En un correo electrónico dirigido al grupo de noticias *comp.editors* Jonathan Payne introdujo un nuevo algoritmo de refresco de pantalla o redisplay. Dicho algoritmo se basaba en la existencia de unas estructuras sobre el buffer de texto que denominó *buffer spans*.

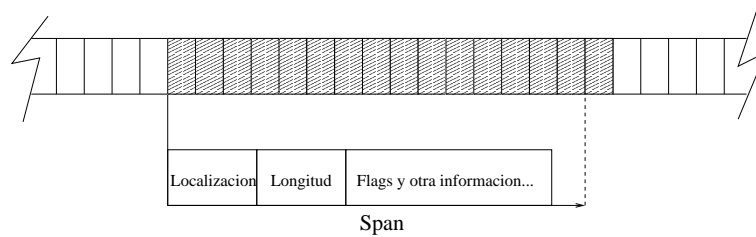


Figura 4.16: Estructura de un tramo

Los spans (que en este libro se denominan *tramos*) definen regiones dentro del buffer de una forma más inmediata que mediante los punteros y las marcas. Cada tramo consiste en:

- Una **localización** que determina el comienzo del tramo dentro del contenido del buffer.
- Una longitud que determina el número de elementos que abarca el tramo desde su comienzo.
- Flags de estado y otra información que el usuario quiera asociar con el tramo.

Como veremos más adelante, mientras que las regiones determinadas con el puntero y las marcas están pensadas para ser utilizadas por el usuario a la hora de guiar las operaciones de edición, los tramos son más adecuados a la hora de estructurar el contenido del buffer de una forma más opaca hacia el usuario.

Añadiendo información de enlace a los tramos podemos estructurar regiones del buffer en listas, árboles y grafos arbitrarios. Esto se examina en los apartados siguientes.

#### 4.3.4 Listas de tramos

Supongamos que estamos diseñando un algoritmo de refresco en pantalla que realiza el redisplay línea a línea. Para ello es necesaria una forma relativamente directa de acceder a la siguiente información del estado del buffer:

- El contenido de una línea arbitraria.
- Cuántos elementos contiene en cualquier momento una línea arbitraria.
- Determinar si una línea arbitraria ha cambiado de posición (después de la inserción o borrado de otra línea, por ejemplo).

Si el sistema de buffers que utilizamos para la implementación soporta tramos podríamos definir un tramo por cada línea que abarque su contenido. De esta forma nos basta examinar la localización de comienzo del tramo para determinar si la línea ha cambiado de sitio, y su longitud para ver si el tamaño del contenido ha cambiado. En cuanto al contenido, una invocación al procedimiento del buffer que retorna el contenido del tramo será suficiente.

Sin embargo sería necesaria alguna forma de enlace entre los tramos que definen las líneas del buffer. Para ello el sistema de tramos podría implementar *listas de tramos* como puede verse en la Figura 4.17.

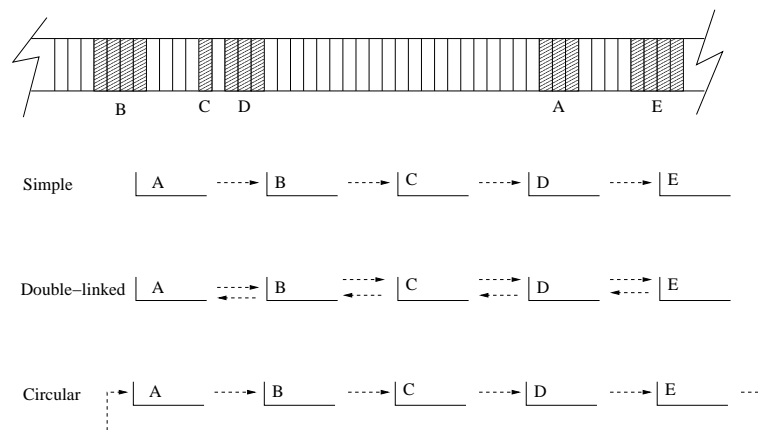


Figura 4.17: Lista de tramos

Un sistema flexible debería implementar listas simples, doblemente enlazadas, circulares o una combinación de lo anterior. Del mismo modo debería habilitarse un sistema de acceso directo por índice a los tramos que componen las listas.

### 4.3.5 Árboles de tramos

De la misma forma que la estructuración de regiones del buffer en listas puede resultar útil para ciertas labores, la posibilidad de definir árboles de tramos en el mismo buffer resulta tremendamente adecuada para otras.

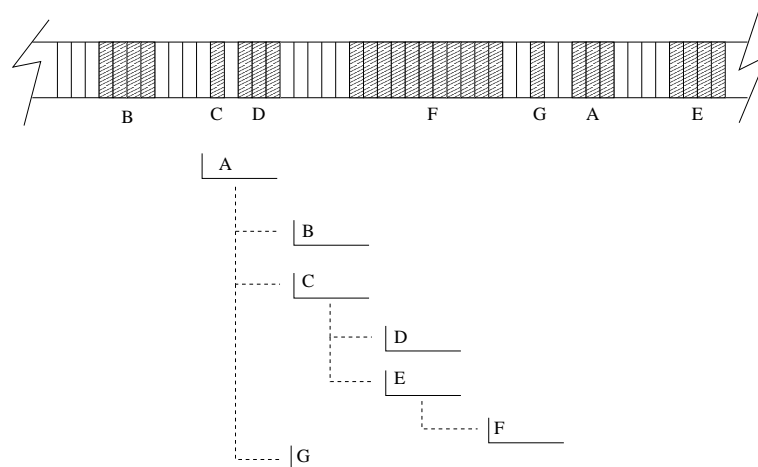


Figura 4.18: Árbol de tramos

Pensemos por ejemplo en los editores de estructura. Los editores de estructura generalmente están diseñados para gestionar contenido que se ajusta a alguna especificación formal como es el caso de una gramática. Los lenguajes de programación ocupan el lugar estrella

entre este contenido *estructurado*, pero también podría pensarse en ficheros de configuración o de datos.

Una de las tareas que deben llevar a cabo los editores de estructuras es el mantenimiento de un *árbol de sintáxis* que refleje la adecuación del contenido del buffer a la gramática del lenguaje correspondiente. Dado que la construcción del árbol debe hacerse de forma incremental y en tiempo real, a medida que el usuario va introduciendo caracteres en el buffer, la posibilidad de utilizar árboles de tramos se convierte en una gran ventaja. Ya no es necesario mantener una estructura aparte. El propio buffer mantiene las regiones que pertenecen a tal o cual regla sintáctica. Por una parte el análisis léxico de cada una de las piezas sintácticas se facilita bastante (acceso en el tramo correspondiente) y por la otra la recuperación de información correspondiente a una pieza arbitraria resulta inmediata.

### 4.3.6 Grafos de tramos

No existe ninguna razón para limitar las capacidades de enlace de los tramos a listas y árboles. La extensión para soportar grafos simples o dirigidos puede resultar muy útil a la hora de implementar otras facilidades del editor de texto.

Siguiendo con el ejemplo de los editores de estructuras, supongamos que deseamos introducir cierto soporte de semántica en el editor. Para ello se requiere ir *adornando* el árbol de sintáxis que se va construyendo progresivamente, sintetizando o derivando valores de atributos asociados a los nodos del árbol. Una de las formas más comunes de realizar este adornado consiste en la construcción de un grafo cuyas aristas simbolizan el flujo de información entre los nodos. Un grafo definido sobre los mismos tramos que componen el árbol bastaría para ello.

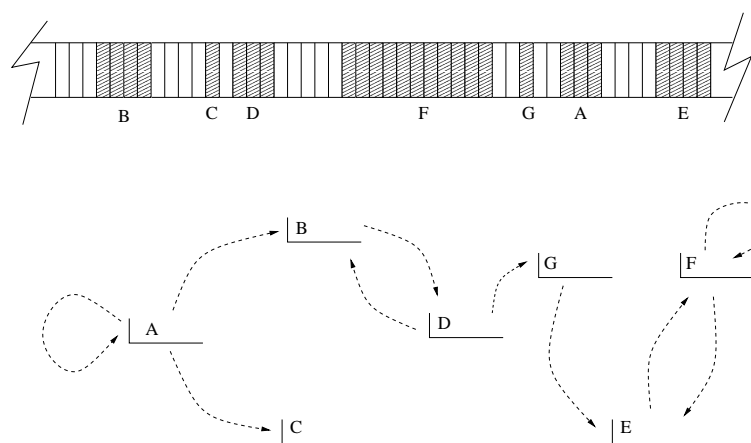


Figura 4.19: Grafo de tramos

En esta nueva forma de estructuración de tramos sería deseable disponer de algún mecanismo para asociar información a las aristas del grafo.

Las posibilidades son infinitas: podríamos definir (y probar) autómatas mientras editamos su contenido, implementar el sistema de *undo* o deshacer, seguir referencias cruzadas dentro de un documento, etc.

### 4.3.7 Widening y Narrowing

Con la aparición de los editores tipo Emacs comenzó el paradigma del *editor entorno*. Una de las capacidades de Emacs más apreciadas es la de proporcionar interfaces homogéneas a otras aplicaciones presentes en el sistema. En efecto, en GNU Emacs es posible leer el correo electrónico, navegar por una estructura de ficheros y directorios, jugar al *pong*, etc.

En lugar de introducir un mecanismo aparte para construir las interfaces de usuario de estas aplicaciones internas, en los editores tipo Emacs se utilizan los propios buffers de texto (y las ventanas mediante las cuales se interactúa con los mismos) como entorno para la aplicación. Es evidente que es necesario introducir algunas capacidades nuevas para ello, como definir regiones de solo lectura, asociar eventos a regiones de texto (para simular botones) etc.

Una de estas capacidades es el *widening y narrowing*. Mediante esta técnica es posible *estrechar* el buffer de modo que la región efectiva de edición sea una subsecuencia del contenido total del buffer. Una vez ha sido estrechado, todas las operaciones sobre el buffer (exceptuando la que rompe el estrechamiento) asumen que la parte *visible* del buffer es el total: las localizaciones se ajustan al nuevo tamaño y la nueva localización, etc. En definitiva se trata de un *buffer virtual* contenido en el buffer original.

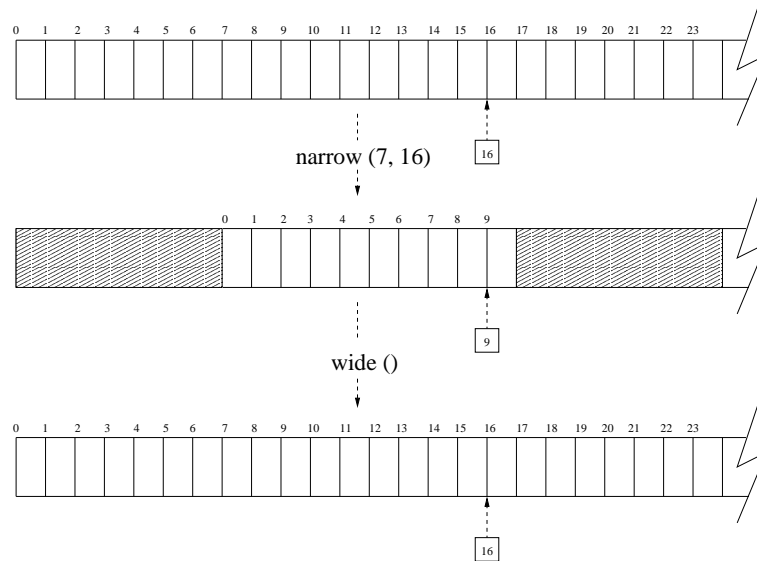


Figura 4.20: Widening y Narrowing

Las primitivas utilizadas para llevar a cabo el estrechamiento son **narrow** (para estrechar) y **wide** para romper el estrechamiento. Dado que solo se permite un estrechamiento por buffer en un momento dado la semántica de *wide* resulta muy simple: romper el estrechamiento activo. Por su parte existen varias alternativas para la semántica de *narrow*:

- El estrechamiento se realiza sobre regiones definidas por el puntero y alguna marca. En este caso la primitiva sería `narrow-to-region` o similar, y aceptaría como parámetro la marca que define la región correspondiente.

- El estrechamiento se realiza sobre dos localizaciones arbitrarias del buffer.

En este caso la primitiva sería **narrow** y aceptaría como parámetro dos localizaciones pertenecientes al buffer.

Nótese que es relativamente sencillo implementar **narrow-to-region** mediante **narrow**, e incluso da la posibilidad de incluir un **narrow-to-span** a efectos de ortogonalidad en el diseño.

### 4.3.8 Vistas

Muchos editores de texto ofrecen varias vistas sobre el mismo buffer. Es el caso de los editores tipo Emacs, donde las ventanas se comportan como *agujeros*. Mediante una ventana puede verse y editarse interactivamente una región de un buffer. Utilizando scrolling se accede a otra región del buffer para su edición o lectura. Nada impide que se visualice el mismo buffer en distintas ventanas. Esto es útil, por ejemplo, cuando se está editando una parte de un fichero mientras se tiene a la vista otra parte distinta que contenga algo de interés.

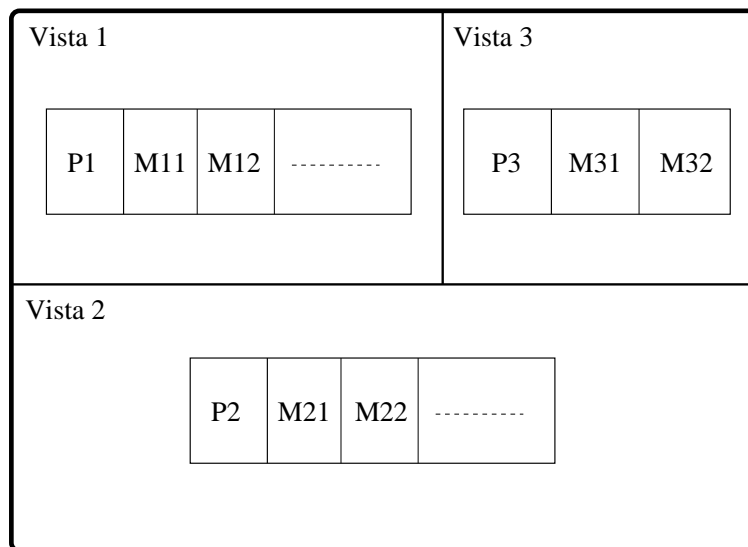


Figura 4.21: Dos vistas de un buffer

Pues bien. En dichos editores lo habitual es asociar un puntero y un juego de marcas a cada una de las ventanas que muestran un mismo buffer. Piénsese que en el caso del puntero es obligado, ya que si no no podrían mostrarse dos regiones disjuntas del buffer en las dos ventanas: ¿dónde estaría el cursor, entonces?. La Figura 4.21 muestra tres ventanas accediendo a regiones distintas de un buffer en lo que sería un editor tipo Emacs. Al contenido mostrado por cada una de las ventanas lo denominamos *vista* del buffer.

Si queremos diseñar un editor que soporte vistas simultáneas de un mismo buffer como las descritas anterioremente podemos optar por varias estrategias distintas en lo que a la implementación se refiere.

Tal vez la solución mas obvia consista en utilizar tantas marcas como vistas existan en la pantalla para que actuen al modo de punteros. En este esquema el puntero del buffer serviría de *copia efectiva*. Esta solución es muy pobre: ineficiente y problemática. Dado que el sistema de buffers está preparado para tener un solo puntero las operaciones de edición siempre se realizan en la localización indicada por dicho puntero. Esto implica que con este esquema de utilización de marcas auxiliares tendríamos que ir salvando los datos del puntero en cada marca a medida que cambiamos de una ventana a otra.

Otra solución consistiría en implementar el soporte de las vistas en el mismo sistema de buffers. De este modo el implementador no debe preocuparse de mantener el puntero adecuado en cada una de las vistas. Ahora bien, ¿qué información debe ser específica a cada vista del buffer?. Desde luego no parece ser el caso de la secuencia editable (el contenido que se modifica en una de las vistas es modificado en todas ellas) ni de los tramos (siempre orientados al contenido). Respecto al widening y al narrowing, tampoco parece apropiado hacerlos específicos de cada vista, ya que no nos permitiría fijar varias vistas en distintas partes de una interfaz construida con este mecanismo.

Por otra parte ya hemos mencionado que en el caso del puntero resulta obligatoria una copia por cada vista, dado que suele arrastrar consigo al cursor.

Por último nos queda considerar el caso de las marcas. Ya se ha comentado que las marcas son el mecanismo mas apropiado para delimitar las operaciones de edición realizadas por el usuario. Por su parte, y debido al principio de la localización de las ediciones, dichas operaciones se llevarán a cabo relativamente cerca del puntero. Luego si queremos que el usuario pueda editar regiones de texto (delimitadas por las marcas y el puntero) en todas las vistas del buffer deberemos incluir a las marcas como parte de la información específica de cada vista.

Ademas del puntero y las marcas parece razonable reservar cierto espacio en cada descriptor de vista para los datos que el implementador crea necesario incluir (como un puntero a la estructura del componente de redisplay que represente a la ventana donde se muestra la vista, por ejemplo).

## 4.4 El caso GNU Emacs

GNU Emacs emplea una implementación de sus buffers altamente especializada y eficiente. Utiliza la técnica del buffer gap en su gestión de las secuencias internas.

En GNU Emacs (a partir de este momento referido simplemente como Emacs) la implementación de los buffers de texto se divide en dos partes principales. La partición de la información en dos partes de la información que compone un buffer se realizó en vistas a la implementación de un concepto no visto hasta ahora: el buffer indirecto. Este concepto es propio de GNU Emacs y es totalmente distinto a las *vistas* estudiadas en la Sección 4.3 [Estructuración externa], página 78.

En el fichero ‘`buffer.h`’ podemos encontrar dos estructuras de datos: `struct buffer` y `struct buffer_text`. Veamos cada una de ellas junto con su contenido.

```
struct buffer_text
```

Es en esta estructura donde se almacena el contenido textual del buffer, además de otra información compartida por todos los buffers indirectos. En concreto se trata de información relativa al refresco de pantalla y el juego de marcas del buffer.

Los campos del registro dedicados al mantenimiento de la estructura interna de la secuencia editable son los siguientes:

```
unsigned char *beg;
    Se trata de la dirección en memoria del comienzo del texto del
    buffer.

EMACS_INT gpt;
    Posición del comienzo del gap dentro del buffer.

EMACS_INT z;
    Posición del final del buffer.

EMACS_INT gpt_byte;
    Posición (en octetos) del comienzo del gap dentro del buffer.

EMACS_INT z_byte;
    Posición (en octetos) del final del buffer.

EMACS_INT gap_size;
    Tamaño del gap.
```

Los siguientes campos sirven para realizar tareas de refresco de pantalla, consistiendo en contadores y flags de estado de modificación.

```
int modiff;
    Contador del número de eventos de modificación para este buffer.
    Se incrementa en cada evento que implique modificación del con-
    tenido del buffer.

int save_modiff;
    Valor previo de modiff en el momento del último fichero visitado.

int overlay_modiff;
    Contador del número de modificaciones en los overlays definidos en
    el buffer.

EMACS_INT beg_unchanged;
    Valor mínimo de GPT - BEG desde que finalizó la última pasada de
    redisplay.

EMACS_INT end_unchanged;
    Valor mínimo de Z - GPT desde que finalizó la última pasada de
    redisplay.

int unchanged_modified;
    Igual que modiff pero referido al número de modificaciones desde
    que terminó la última pasada de redisplay. Si es igual a modiff en-
    tonces beg_unchanged y end_unchanged no contienen información
    útil.

int overlay_unchanged_modified;
    Igual que overlay_modiff pero referido al número de modifica-
    ciones en los overlays desde que terminó la última pasada de re-
    display. Si es igual a overlay_modiff entonces beg_unchanged y
    end_unchanged no contienen información útil.
```

Finalmente se mantiene información relativa a las propiedades del texto de este buffer, junto con las marcas definidas en el mismo.

```
INTERVAL intervals;
```

Propiedades del texto contenido en el buffer.

```
struct Lisp_Marker *markers;
```

Lista de marcas que hacen referencia a localizaciones dentro del buffer.

```
struct buffer
```

Esta estructura representa a un buffer de texto en el contexto de la máquina Lisp que incorpora GNU Emacs.

```
EMACS_INT size;
```

```
struct buffer *next;
```

```
struct buffer *base_buffer;
```

```
struct buffer_text own_text;
```

```
struct buffer_text *text;
```

```
EMACS_INT pt;
```

```
EMACS_INT pt_byte;
```

```
EMACS_INT begv;
```

```
EMACS_INT begv_byte;
```

```
EMACS_INT zv;
```

```
EMACS_INT zv_byte;
```

La variable global `current_buffer` alberga el buffer activo en cada momento. Casi todas las operaciones sobre buffers trabajan con `current_buffer` a no ser que se especifique un buffer en concreto mediante un argumento.

## 5 EDKIT: Editor Kit

### 5.1 Motivación

A lo largo de los capítulos anteriores hemos tenido la oportunidad de examinar muchos aspectos del diseño e implementación de un editor de texto. Una de las conclusiones más evidentes es que no se trata de una pieza sencilla de software, ni mucho menos: en la construcción de un editor se ven involucradas muchas tecnologías, incluyendo estructuras de datos, algorítmica, ergonomía computacional, etc.

El aspecto positivo es que dichas tecnologías están en general bien estudiadas y aplicadas con éxito en infinidad de diseños e implantaciones. La construcción de un editor de texto “al uso” no representa un desafío de investigación en la mayoría de los casos, pudiendo llevarse a cabo sin sobresaltos derivados de problemas sin solución inmediata. Efectivamente, prácticamente la totalidad de las preguntas derivadas del proceso de construcción de un editor disponen de respuestas, muchas de ellas expuestas en este documento.

Con esto no se quiere dar a entender que la construcción de un editor no sea costosa. Por el contrario es una tarea que involucra mucho trabajo, especialmente en la fase de implementación. Parte de este trabajo proviene del hecho de la gran dispersión (un tanto caótica) documental que existe en el ámbito de los editores. Desgraciadamente no existe un “corpus documental” sobre diseño e implementación de editores de texto. Este trabajo pretende cubrir este hueco, al menos en parte.

Ahora bien, la implementación de editores de texto siempre se ha considerado una tarea cuanto menos “poco común”. Al igual que sucede con muchos programas de sistema (como compiladores, enlazadores o sistemas de ficheros) los editores de texto son considerados como herramientas indispensables, pero al mismo tiempo poco innovadoras. Este pensamiento es heredado de las décadas de los ochenta y noventa, en las que el esfuerzo principal en el ámbito de la tecnología de los editores de texto estaba dirigido a los “grandes editores” como Emacs o WordStar. En pocos proyectos de desarrollo se hacía necesaria la implementación de un nuevo editor, pudiendo utilizarse uno de los editores existentes para las tareas que requirieran de capacidades no triviales de edición.

Este panorama está cambiando en la actualidad. La generalización (y aceptación por parte de la industria) de “entornos de aplicaciones” cada vez más abstractos y portables, como Java o Tcl, está provocando que muchos proyectos de desarrollo requieran de la implementación de editores de texto (más o menos ambiciosos) que proporcionen un nivel alto de integración en sus productos. Muy a menudo estos editores tienen un campo de aplicación muy concreto:

- Editores de código en entornos integrados de desarrollo (IDEs).
- Editores de documentación en entornos orientados a la publicación.
- Editores de documentación especializada, como por ejemplo requisitos en una herramienta CASE.
- etc...

Cada vez más, estos editores “de propósito específico” suponen un impacto tremendo en el desarrollo del entorno del que forman parte. Como ya hemos destacado, la implementación desde cero de un editor es una tarea grande y compleja, que consume tiempo y recursos.

Aquí es donde entra en juego el concepto de EDKIT, o *kit* para la construcción de editores. Se parte de la premisa de que los editores de propósito específico se desarrollan por la única razón de que necesitan incorporar funcionalidades muy específicas o poco frecuentes, en cualquier caso no ofrecidas por los grandes editores convencionales. Por ejemplo, consideremos una herramienta CASE en la que debe ser posible editar tanto documentación estructurada (capítulos, párrafos, secciones, ...) como programas escritos en algún lenguaje de programación.

Se plantean aquí varios problemas. Para empezar debe tomarse una decisión sobre la identidad de los editores: ¿Deben implementarse dos editores diferenciados, o bien uno solo con la flexibilidad necesaria como para editar dos tipos de documentos de características tan dispares?.

La segunda opción puede resultar absurdamente costosa en tiempo y esfuerzo, pudiendo concluir en la elaboración (no necesaria) de todo un editor de propósito general. Este no es el objetivo del proyecto, que consiste en proporcionar un entorno CASE, compuesto por mucho más que por editores de documentación o programas.

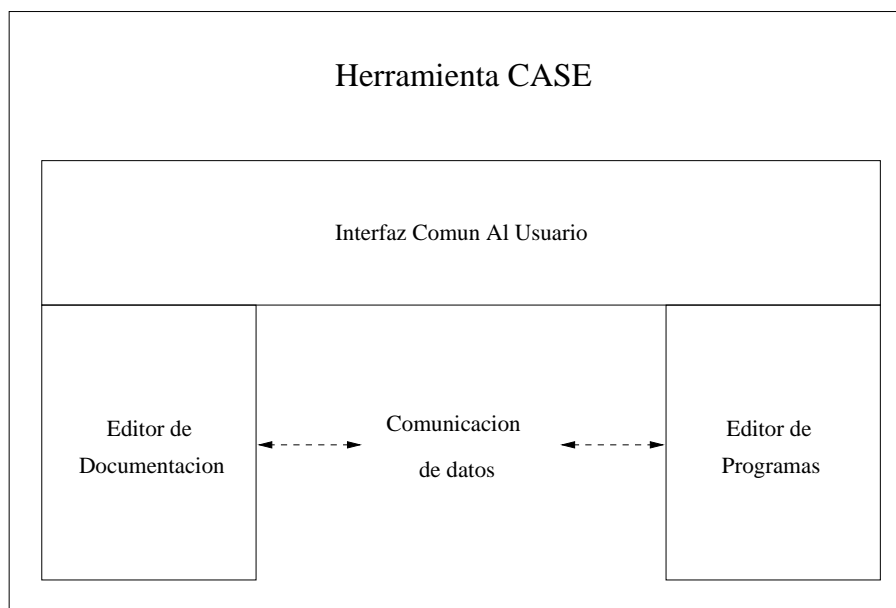


Figura 5.1: Dos editores en un entorno CASE

Esto nos lleva a la primera opción: implementar dos editores distintos. Esto llevaría a la situación de la Figura 5.1. Ambos editores deberían implementar su propio sistema de buffers, codificación de caracteres, soporte de estructuras, redisplay, etc. Los editores deben estar también adaptados a una sola interfaz hacia el usuario, que debería presentar distintos modos de operación (con distintas funcionalidades) dependiendo de si se está editando un documento o un programa. Es muy posible, además, que los editores requieran compartir cierta información: asociar parte de la documentación a una función de un programa, citar parte del código fuente en un documento, llevar parte de la documentación a un fichero

fuente en forma comentada, etc. La implementación de este canal de comunicación deberá ser muy cuidadosa, y requerirá de un esfuerzo adicional.

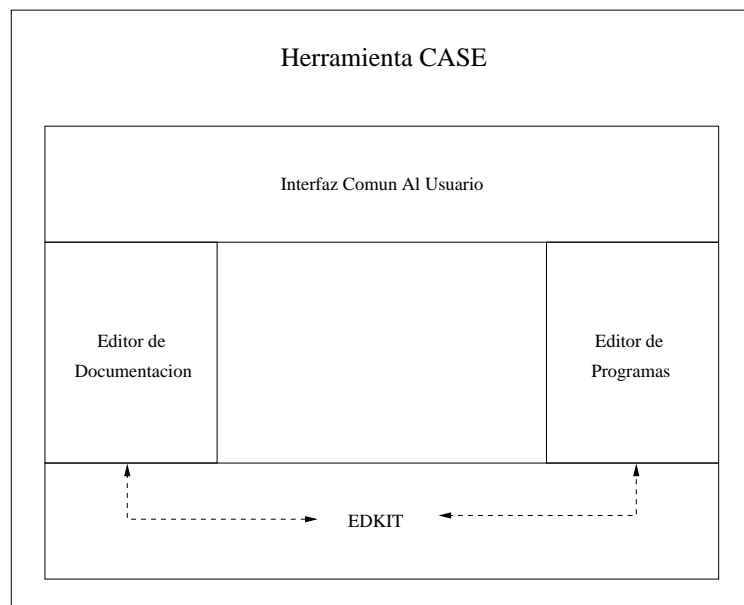


Figura 5.2: Dos editores en un entorno CASE, con EDKIT

Supongamos ahora que el proyecto de desarrollo dispone de una librería que proporciona la implementación de conceptos generales que *comparten ambos editores*, como buffers, sistemas de codificación de caracteres, facilidades de estructuración, capacidades de búsqueda, etc. El entorno CASE podría disponerse entonces como en la Figura 5.2. Los editores de propósito específico solo implementarían las funcionalidades específicas (como las abstracciones de documento y capítulo para la documentación, o navegación por funciones para el código fuente), siendo la librería la encargada de proporcionar las abstracciones básicas y comunes. Otra consecuencia de este esquema proviene del concepto de utilización común de una librería: el canal de comunicación entre los editores puede estar implementado por la misma librería, haciendo transparente la compartición de objetos como buffers, patrones de búsqueda, o cualquiera proporcionado por la propia librería.

La utilización de una librería de estas características (que en el marco de este trabajo denominamos EDKIT) conlleva las dos ventajas siguientes:

#### **Disminución de la complejidad de implementación**

En efecto, hasta ahora el implementador de un editor de propósito específico debía hacerse cargo de las funcionalidades que pueden denominarse “comunes”, presentes en casi cualquier editor de texto. Estas funcionalidades incluyen los objetos buffers y su manipulación.

Esto permite al implementador concentrarse en los aspectos específicos (dotar de estructura al contenido de los buffers, manipulación de primitivas de lenguajes a nivel de editor, etc) y no invertir gran parte de su tiempo y esfuerzo en el

diseño e implementación de los buffers (que como hemos visto en los capítulos anteriores, no es trivial).

### Transparencia en la comunicación de datos entre varios editores

Un entorno que proporcione varios editores de propósito específico (como el entorno CASE de nuestro ejemplo) debe proporcionar vías de comunicación entre los mismos.

La utilización de EDKIT hace gran parte de esa comunicación transparente, ya que la librería proporciona los mecanismos necesarios para la compartición de las abstracciones implementadas por la librería (como los buffers).

## 5.2 Editores cliente

Como ya se ha mencionado en este mismo capítulo es posible construir editores (tanto de propósito general como específico) haciendo uso de EDKIT como una librería de implantación de funcionalidades comunes a editores de texto.

En este esquema aparece el concepto de *editor cliente* o *editor objetivo*, que implementa las características específicas no cubiertas por EDKIT (o sí implementadas pero no deseadas) y hace uso de la librería. La Figura 5.3 muestra de forma esquemática la estructura de un editor implementado como un editor cliente de EDKIT. El editor cliente accede a las funcionalidades de EDKIT mediante una interfaz de programación.

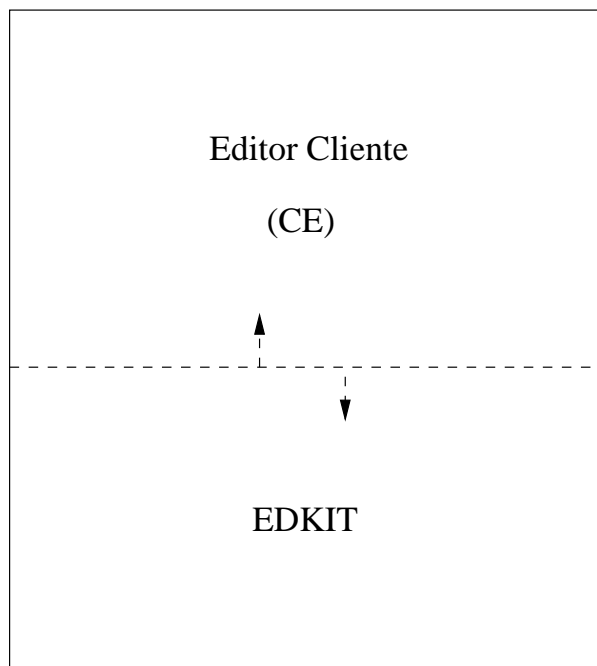


Figura 5.3: EDKIT y un editor cliente

Al ser una librería, EDKIT permite que varios editores cliente convivan y hagan uso de sus funcionalidades. En el apartado titulado “Motivación” ya se expuso un ejemplo de la

utilidad de este esquema. La Figura 5.4 muestra dos editores cliente haciendo uso de una sola instancia de EDKIT.

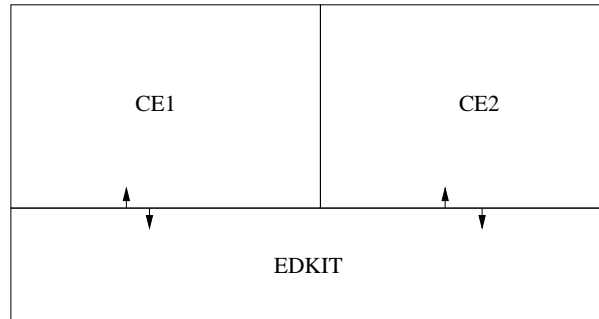


Figura 5.4: EDKIT y dos editores cliente

Cabe preguntarse por el encuadre funcional de la librería en el diseño de los editores cliente. Recordemos la descomposición funcional de un editor expuesta en el capítulo 1 (una de muchas posibles). Los tres módulos funcionales del editor (sub-editor, comandos de usuario y redisplay) pueden utilizar funcionalidades exportadas por EDKIT para su implementación. Es de notar que uno de los objetivos de EDKIT es dotar al editor cliente de completa libertad a la hora de utilizar sus funcionalidades: un editor cliente pudiera utilizar la implementación EDKIT de buffers e ignorar cualquier otra parte de la librería. Así, el encuadre funcional presentado en la Figura 5.5 debe entenderse como orientativo, ya que muy bien pudiera utilizarse la librería únicamente en el sub-editor u otro de los módulos funcionales.

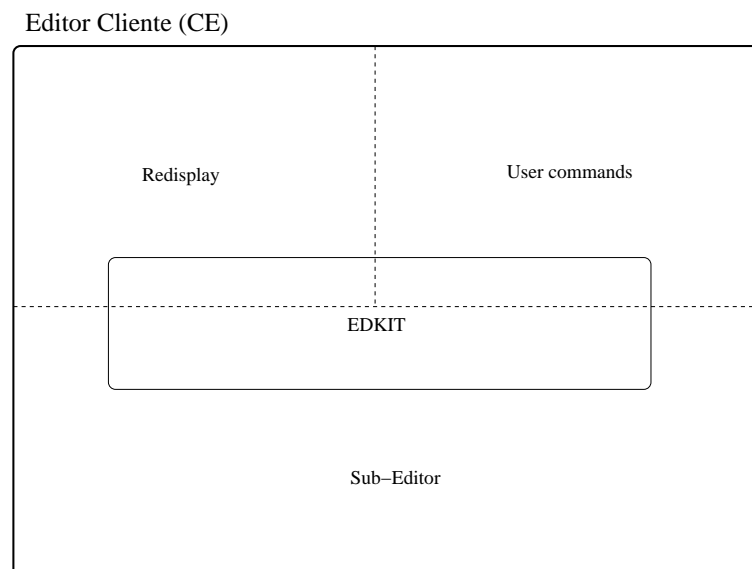


Figura 5.5: Aportación funcional del EDKIT

### 5.3 Arquitectura

Un *kit* es alegóricamente similar a una caja de herramientas: el usuario tiene acceso a una colección de herramientas (facilidades) que en principio cubren sus necesidades. Sin embargo el usuario escogerá de entre dicha colección las herramientas que necesita, en base a ciertos criterios como la naturaleza del trabajo a realizar o la experiencia del usuario en dichos trabajos.

EDKIT pretende jugar el papel de dicha caja de herramientas para los implementadores de editores de texto. Para ello proporciona implementaciones de abstracciones tal vez útiles al implementador. El implementador, por su parte, debe disponer de libertad para utilizar cero o mas de dichas funcionalidades.

De esto se deduce que la arquitectura de la librería debe ser **flexible** y **escalable**, dotando al mismo tiempo a las funcionalidades exportadas de la mayor **independencia** posible.

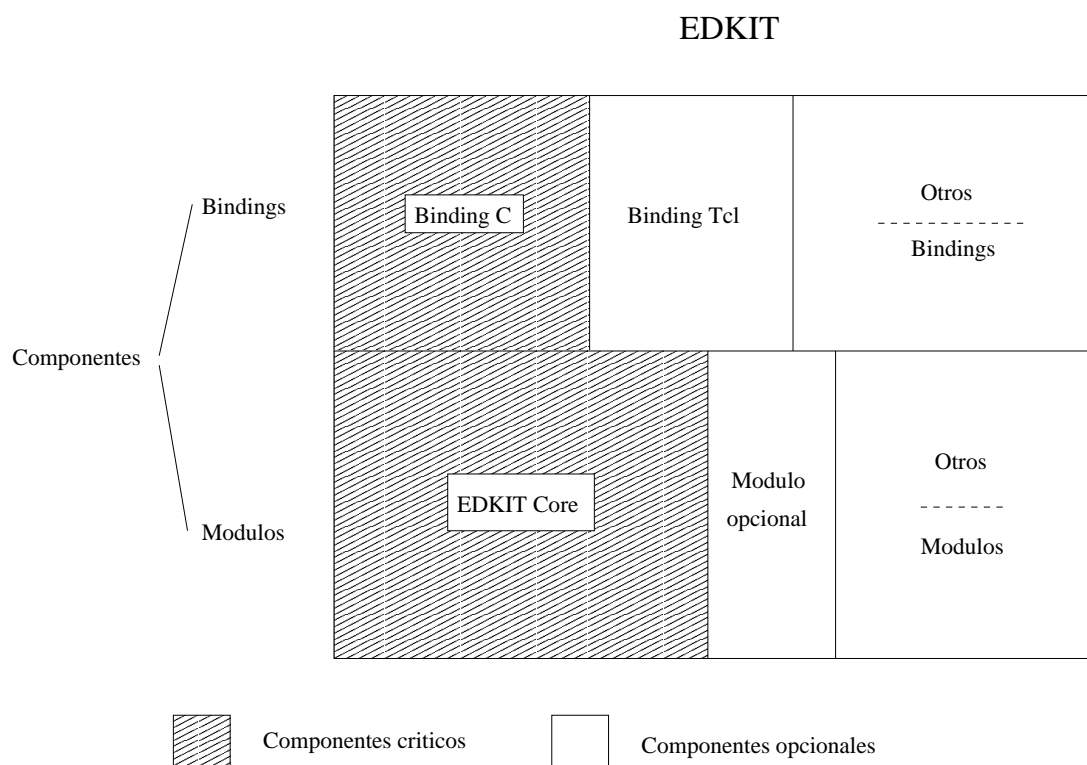


Figura 5.6: Arquitectura del kit

La figura Figura 5.6 muestra una arquitectura que cumple estos requisitos. La librería está dividida en una serie de componentes de dos tipos: módulos y bindings.

#### Módulos

Las funcionalidades exportadas por la librería EDKIT son implementadas por diversos módulos. La división de la librería en módulos responde a los requisitos de flexibilidad y escalabilidad. De la misma forma, las funcionalidades

implementadas en distintos módulos son en principio independientes entre sí, no existiendo ninguna dependencia entre ellas.

Sin embargo el módulo **core** es una excepción a esta regla. Se trata del módulo EDKIT principal. Es necesario enlazar con la librería que implementa el core para utilizar cualquier otro módulo, ya que, entre otras cosas, implementa el soporte para la gestión de los demás módulos. Es por eso que en la Figura 5.6 se ha distinguido este módulo como un *módulo crítico* (esto es, que debe utilizarse para hacer uso del kit).

## Bindings

Los bindings son el medio por el cual los editores clientes interactúan con EDKIT. Cada binding proporciona una interfaz adecuada a distintos lenguajes de programación y entornos de aplicaciones.

Es de destacar que estos bindings no deben limitarse a hacer una traducción “procedimiento por procedimiento” y “tipo de dato por tipo de dato” de las funcionalidades implementadas por los módulos. Por el contrario, deben adaptar esas funcionalidades a la idiosincrasia propia de cada lenguaje o entorno. Por ejemplo, el binding a un lenguaje orientado a objetos debe proporcionar una jerarquía de clases apropiada que represente de forma natural los objetos soportados por la librería, como los buffers. Un binding que implementara una simple y poco conveniente clase estática no explotaría al máximo la utilidad de la librería.

De forma similar al módulo core, el binding de C es considerado como “crítico”, ya que a diferencia de los demás es implementado directamente por los módulos: no hay capa de adecuación.

En resumen, se ha sugerido una arquitectura en dos capas para cumplir los requisitos del kit de flexibilidad, escalabilidad e independencia entre módulos.

## 5.4 Funcionalidades

Los apartados siguientes resumen las funcionalidades exportadas por el módulo core de EDKIT al editor cliente. Al estar implementadas en un solo módulo, estas funcionalidades forman un todo y deben utilizarse de forma conjunta. Por ello el módulo core debe implementar únicamente las funcionalidades necesarias para dotar de un mínimo de utilidad al kit. Cualquier otra funcionalidad debe implementarse en otro módulo.

### 5.4.1 Codificación de caracteres

Ya vimos en el capítulo 2 el gran impacto que tienen los CSS (sistemas de codificación de caracteres) en los editores de texto. El kit debe proporcionar una base común para que el manejo de caracteres por parte de los demás módulos de la librería (y del editor cliente) sea coherente y eficiente.

Para ello pueden formularse las siguientes decisiones de diseño:

- EDKIT debe implementar soporte nativo al CCS universal ISO 8859 (UCS) y Unicode. Esto involucra a las funcionalidades de búsqueda y comparación de texto.
- EDKIT debe utilizar UTF-8 como representación interna del texto, ya que es una forma compacta y completa.

- EDKIT debe soportar varias codificaciones externas y proporcionar al editor cliente los mecanismos adecuados para realizar las transformaciones pertinentes.

En definitiva, el módulo core debe proporcionar, en su interfaz de programación, los procedimientos adecuados para almacenar cadenas de texto codificadas en las codificaciones externas e interna y su manipulación.

### 5.4.2 Buffers

Habrá quedado patente a lo largo del libro que los buffers de texto son una parte crucial de cualquier editor de texto, independientemente de su aplicación. Como un kit para implementar editores, esta importancia permanece en EDKIT.

El *buffer EDKIT* (o *edk buffer*) es la abstracción fundamental exportada por la librería. Tanto es así que prácticamente la totalidad del resto de funcionalidades manipulan buffers de texto o los amplían con mas semántica.

Es claro que la abstracción buffer debe ser flexible de modo que pueda adaptarse tanto a editores cliente con requerimientos especiales como a entornos de ejecución muy diversos.

Los requisitos de los editores cliente pueden expresarse en términos de la **estructuración externa** del buffer. Mientras que un editor cliente convencional no requerirá de mucha mas estructura adicional que una secuencia de caracteres (o, como mucho, una lista de líneas) un editor cliente de estructura puede beneficiarse de estructuraciones externas mas elaboradas como listas de tramos, árboles de tramos o grafos de tramos.

Luego, a efectos de responder a la mayor cantidad posible de requerimientos de editores clientes, la abstracción buffer exportada por EDKIT debe proporcionar las siguientes facilidades de estructuración externa:

- Localizaciones
- Punteros y marcas
- Tramos
- Listas de tramos
- Árboles de tramos
- Grafos de tramos
- Tanto Widening como Narrowing
- Posibilidad de definir Vistas

Por otra parte, los editores cliente tambien determinan requisitos derivados de la plataforma de ejecución donde son utilizados. Los requerimientos de memoria de un sistema empujado (como un computador de mano) son radicalmente distintos a los de una estación de trabajo (PC) convencional. Estos requisitos pueden expresarse en función de los diversos esquemas de estructuración interna de buffers expuestos en este libro.

Luego, a efecto de satisfacer los variados requerimientos impuestos por las plataformas de ejecución de los potenciales editores cliente, la abstracción buffer exportada por EDKIT debe implementar las siguientes facilidades de estructuración interna:

- Vectores
- Buffer Gap
- Listas enlazadas

- Líneas en tramos
- Buffers de tamaño fijo (en concreto, buffer gap paginado)
- Tablas de piezas

### 5.4.3 Facilidades básicas de búsqueda

Buscar un patrón en un texto de un buffer es una operación muy común y se da en casi cualquier proceso de edición interactivo (buscar un nombre, buscar el principio de la palabra que se está escribiendo, etc).

Caben distinguir dos tipos principales de patrones:

- Patrones fijos
- Expresiones regulares

El módulo core de EDKIT debe proporcionar al menos búsqueda de patrones fijos en los objetos buffer. Las búsquedas basadas en patrones más complejos, como expresiones regulares, pueden implementarse en otros módulos.

Los algoritmos de búsqueda de cadenas que debe implementar el módulo core de EDKIT son los siguientes:

- Fuerza bruta
- Knuth-Morris-Pratt
- Boyer-Moore (en su variante desarrollada por Horspool)

### 5.4.4 Tablas de caracteres

Las “tablas de caracteres” son tablas dispersas indexadas por códigos de caracteres. Son útiles para implementar cualquier mecanismo que requiera indexado por caracteres (muy comunes en editores de texto) como mapas de teclas, almacenamiento de atributos de caracteres, implementación de tablas de comandos, etc.

CAR	CDR
Codigo Caracter	Puntero
Codigo Caracter	Puntero
-----	-----
Codigo Caracter	Puntero

Figura 5.7: Estructura lógica de una tabla de caracteres

La estructura lógica de una tabla de caracteres puede verse en la figura Figura 5.7. La entrada *CAR* contiene códigos de caracteres y sirve para indexar la tabla. La entrada *CDR* contiene un puntero a dos posibles objetos:

- Otra tabla de caracteres. Esto permite el anidamiento de tablas, y por tanto de la elaboración de estructuras tan complejas como se quiera.
- Datos arbitrarios definidos por el usuario. Esto permite asociar a cada carácter datos estáticos, dinámicos o incluso funciones o rutinas.

Las características principales de una tabla de caracteres son:

- Es una tabla **completa**, en el sentido de que siempre contiene una entrada para cualquier código de carácter válido.
- Es una implementación **eficiente**, dado que se utilizan estructuras *dispersas* que no requieren el almacenamiento en memoria de todo el espacio de caracteres.

Inicialmente cada CDR contiene `CT_VOID`, un nombre simbólico que indica que la entrada no contiene ningún dato asociado.

A efectos de clarificación se incluye a continuación la estructura de datos correspondiente a un CDR:

```
typedef struct _edkit_ct_cdr_t
{
    short type;

    union
    {
        struct _edkit_ct_cdr_t *chain;
        void *data;
    }

} *edkit_ct_cdr_t;
```

Y los posibles valores de `type`, que definen el estado de la entrada en la tabla de caracteres:

- |                         |   |
|-------------------------|---|
| <code>CT_VOID</code>    | No se ha definido una cadena o datos.   |
| <code>CT_DATA</code>    | Se trata de una entrada de datos, disponibles en <code>data</code> .  |
| <code>CT_CHAIN</code>   | Se trata de una entrada de cadena, y la tabla de caracteres encadenada se puede encontrar en <code>chain</code> . |
| <code>CT_INVALID</code> | Se trata de una entrada desactivada.  |
| <code>CT_FIRST</code>   | Se trata de la primera entrada del rango permitido de entradas.   |
| <code>CT_LAST</code>    | Se trata de la última entrada del rango permitido de entradas.  |

La posibilidad de encadenar varias tablas de caracteres nos permite construir jerarquías, como se muestra en la figura Figura 5.8.

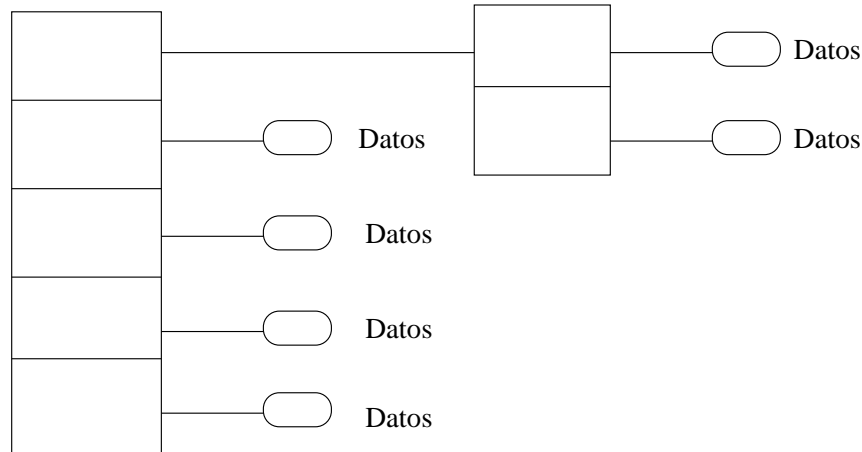


Figura 5.8: Una jerarquía de tablas de caracteres

Es posible recorrer árboles contruidos con tablas de caracteres utilizando las funciones `edkit_ct_set` y `edkit_ct_get`, detectando las terminaciones de las ramas con la constante `CT_VOID`.

## 5.5 Interfaz de programación

En este apartado se expone una interfaz de programación para EDKIT. Esta interfaz se corresponde con la implementada por el binding de C.

Nótese que la interfaz expuesta aquí solo cubre las abstracciones EDKIT mas importantes.

### 5.5.1 Interfaz para secuencias de texto

`edkit_sequence_t edkit_sequence_create (void);` [Función]

Crea y retorna una secuencia de texto vacía.

`void edkit_sequence_destroy (sequence);` [Función]

Destruye la memoria asociada con *sequence*.

`edkit_bool_t edkit_sequence_empty_p (sequence);` [Función]

Determina si *sequence* está vacía.

`edkit_sequence_offset_t edkit_sequence_get_length (sequence);` [Función]

Retorna la longitud de la secuencia (es decir, el número de caracteres que contiene).

`edkit_status_t edkit_sequence_insert (sequence, position, string);` [Función]

Inserta *string* como una subsecuencia en la posición *position*.

Cualquier intento de sobrepasar el final de la secuencia da como resultado el código `FAIL` y no se garantiza la integridad de los datos contenidos en *sequence*.

`edkit_status_t edkit_sequence_delete (sequence, position, offset);` [Función]  
 Borra la subsecuencia (`position-[position+offset]`).

`edkit_status_t edkit_sequence_get_subseq (sequence, strptr, position, offset);` [Función]  
 Retorna la subsecuencia (`position-[position+offset]`) en `strptr` (un puntero a cadena).

`edkit_status_t edkit_sequence_copy (sequence, from_position, offset, to_position);` [Función]  
 Copia la subsecuencia (`from_position-[from_position+offset]`) a la posición `to_position`.  
 Cualquier intento de sobrepasar el final de la secuencia da como resultado el código FAIL y no se garantiza la integridad de los datos contenidos en `sequence`.

`edkit_status_t edkit_sequence_move (sequence, from_position, offset, to_position);` [Función]  
 Mueve la subsecuencia (`from_position-(from_position+offset)`) a la posición `to_position`, destruyendo el contenido de cualquier subsecuencia previa (`to_position-[to_position+offset]`).  
 Cualquier intento de sobrepasar el final de la secuencia da como resultado el código FAIL y no se garantiza la integridad de los datos contenidos en `sequence`.

### 5.5.2 Inicialización y finalización

`edkit_status_t edkit_init (void);` [Función]  
 It is the EdKit initializer. It must be called upon editor invocation, before further use of the EdKit library. No other EdKit procedure except for `edkit_fini` should be called unless `edkit_init` returns a successful status. After this call, one empty buffer exist into the EdKit, called `'*scratch*`'.

`edkit_status_t edkit_fini (void);` [Función]  
 When called, this routine terminate all the EdKit internal state.

### 5.5.3 Gestión de buffers

Cada buffer EDKIT existente en el sistema es identificado por una cadena de texto. Este nombre debe ser especificado en la creación del buffer, y el editor cliente es el responsable de recordar qué nombre asignó a qué buffer.

`edkit_status_t edkit_buffer_create (char *buffer_name);` [Función]

`edkit_status_t edkit_buffer_clear (char *buffer_name);` [Función]

`edkit_status_t edkit_buffer_delete (char *buffer_name);` [Función]

`char* edkit_buffer_set_next (void);` [Función]

`edkit_status_t edkit_buffer_set_name (char *buffer_name);` [Función]

`char* edkit_buffer_get_name (void);` [Función]

### 5.5.4 Gestión de localizaciones

`int edkit_compare_locations (edkit_location_t loc1, edkit_location_t loc2);` [Función]

`int edkit_location_to_count (edkit_location_t loc);` [Función]

`edkit_location_t edkit_count_to_location (int count);` [Función]

### 5.5.5 Gestión del puntero

`edkit_status_t edkit_point_set (edkit_location_t loc);` [Función]

`edkit_status_t edkit_point_move (int count);` [Función]

`edkit_location_t edkit_point_get (void);` [Función]

`int edkit_point_get_line (void);` [Función]

`edkit_location_t edkit_buffer_start (void);` [Función]

`edkit_location_t edkit_buffer_end (void);` [Función]

### 5.5.6 Gestión de marcas

`edkit_status_t edkit_mark_create (int mark, edkit_mark_type_t type);` [Función]

`void edkit_mark_delete (int mark);` [Función]

`edkit_status_t edkit_mark_to_point (int mark);` [Función]

`edkit_status_t edkit_point_to_mark (int mark);` [Función]

`edkit_location_t edkit_mark_get (int mark);` [Función]

`edkit_status_t edkit_mark_set (int mark, edkit_location_t loc);` [Función]

`edkit_bool_t edkit_is_point_at_mark (int mark);` [Función]

`edkit_bool_t edkit_is_point_before_mark (int mark);` [Función]

`edkit_bool_t edkit_is_point_after_mark (int mark);` [Función]

`edkit_status_t edkit_swap_point_and_mark (int mark);` [Función]

### 5.5.7 Gestión del contenido del buffer

`char edkit_get_char (void);` [Función]

`void edkit_get_string (char *string, int count);` [Función]

`int edkit_get_num_chars (void);` [Función]

`int edkit_get_num_lines (void);` [Función]

`int edkit_get_column (void);` [Función]

`void edkit_set_column (int column);` [Función]

`void edkit_insert_char (char c);` [Función]

<code>void edkit_insert_string (char *string);</code>	[Función]
<code>void edkit_replace_char (char c);</code>	[Función]
<code>void edkit_replace_string (char *string);</code>	[Función]
<code>edkit_status_t edkit_delete (int count);</code>	[Función]
<code>edkit_status_t edkit_delete_region (int mark);</code>	[Función]
<code>edkit_status_t edkit_copy_region (char *buffer_name, int mark);</code>	[Función]

### 5.5.8 Búsqueda de patrones fijos

<code>edkit_status_t edkit_search_forward (char *string);</code>	[Función]
<code>edkit_status_t edkit_search_backward (char *string);</code>	[Función]
<code>edkit_bool_t edkit_is_a_match (char *string);</code>	[Función]
<code>edkit_status_t edkit_find_first_in_forward (char *string);</code>	[Función]
<code>edkit_status_t edkit_find_first_in_backward (char *string);</code>	[Función]
<code>edkit_status_t edkit_find_first_not_in_forward (char *string);</code>	[Función]
<code>edkit_status_t edkit_find_first_not_in_backward (char *string);</code>	[Función]

### 5.5.9 Gestión de tablas de caracteres

#### Obteniendo y almacenando información

<code>void edkit_ct_set (ct, ichar, cdr_type, cdr_access)</code>	[Función]
<code>void* edkit_ct_get (ct, ichar)</code>	[Función]

#### Creación y destrucción de tablas

<code>edkit_ct_t edkit_ct_create (from-char, to-char)</code>	[Función]
<code>void edkit_ct_destroy (edkit_ct_t ct)</code>	[Función]

#### Narrowing

<code>void edkit_ct_allow (ct, from-char, to-char)</code>	[Función]
<code>void edkit_ct_disallow (ct, from-char, to-char)</code>	[Función]

#### Predicados

<code>bool_t edkit_ct_void_p (ct, ichar)</code>	[Función]
---	-----------

Instead of:

```

    if (edkit_ct_get (CT, ICHAR) == CT_VOID)
    {
        /* do something */
    }

```

you can use `edkit_ct_void_p` and write:

```
    if (edkit_ct_void_p (CT, ICHAR))
    {
        /* do something */
    }
```

Similarly, there is the `edkit_ct_inval_p` predicate.

```
bool_t edkit_ct_inval_p (ct, ichar)
```

[Función]

## 6 Conclusiones y líneas futuras

La conclusión principal de este trabajo consiste en el siguiente planteamiento: el área de la edición de texto (y editores de texto) se enfrenta a un nuevo desafío, que es la necesidad de construcción de editores de aplicación específica.

Por tanto, el “status-quo” presente en el área durante los últimos quince años ya no es sostenible: es necesario un replanteamiento en los esquemas y métodos aplicados hasta ahora en el diseño e implementación de los editores. Planteamientos bien asentados, como la conveniencia de reutilizar, adaptar o integrar los grandes editores de propósito general ya no tienen por qué ser válidos.

De hecho, (EDKIT) no es mas que una respuesta (que puede ser mas o menos acertada) a esta situación.

Las líneas futuras pasan por seguir desarrollando EDKIT como arquitectura de referencia. Se planteará en foros de referencia tanto el problema como la solución propuesta en este trabajo. Del mismo modo se habilitará un proyecto de desarrollo para implementar la arquitectura EDKIT.

En cuanto al estudio expuesto en este libro, se ampliará para cubrir mas aspectos del área de la edición, como ergonomía del usuario y evaluación funcional de editores de texto.

# Apéndice A GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

#### 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled “Acknowledgments” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant

Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgments”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this

License will not have their licenses terminated so long as such parties remain in full compliance.

#### 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

### A.0.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C) year your name.  
Permission is granted to copy, distribute and/or modify this document  
under the terms of the GNU Free Documentation License, Version 1.1  
or any later version published by the Free Software Foundation;  
with the Invariant Sections being list their titles, with the  
Front-Cover Texts being list, and with the Back-Cover Texts being list.  
A copy of the license is included in the section entitled ‘‘GNU  
Free Documentation License’’.
```

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being *list*”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

## Índice de funciones

edkit_buffer_clear.....	100	edkit_insert_char.....	101
edkit_buffer_create.....	100	edkit_insert_string.....	102
edkit_buffer_delete.....	100	edkit_is_a_match.....	102
edkit_buffer_end.....	101	edkit_is_point_after_mark.....	101
edkit_buffer_get_name.....	100	edkit_is_point_at_mark.....	101
edkit_buffer_set_name.....	100	edkit_is_point_before_mark.....	101
edkit_buffer_set_next.....	100	edkit_location_to_count.....	101
edkit_buffer_start.....	101	edkit_mark_create.....	101
edkit_compare_locations.....	101	edkit_mark_delete.....	101
edkit_copy_region.....	102	edkit_mark_get.....	101
edkit_count_to_location.....	101	edkit_mark_set.....	101
edkit_ct_allow.....	102	edkit_mark_to_point.....	101
edkit_ct_create.....	102	edkit_point_get.....	101
edkit_ct_destroy.....	102	edkit_point_get_line.....	101
edkit_ct_disallow.....	102	edkit_point_move.....	101
edkit_ct_get.....	102	edkit_point_set.....	101
edkit_ct_inval_p.....	103	edkit_point_to_mark.....	101
edkit_ct_set.....	102	edkit_replace_char.....	102
edkit_ct_void_p.....	102	edkit_replace_string.....	102
edkit_delete.....	102	edkit_search_backward.....	102
edkit_delete_region.....	102	edkit_search_forward.....	102
edkit_find_first_in_backward.....	102	edkit_sequence_copy.....	100
edkit_find_first_in_forward.....	102	edkit_sequence_create.....	99
edkit_find_first_not_in_backward.....	102	edkit_sequence_delete.....	100
edkit_find_first_not_in_forward.....	102	edkit_sequence_destroy.....	99
edkit_fini.....	100	edkit_sequence_empty_p.....	99
edkit_get_char.....	101	edkit_sequence_get_length.....	99
edkit_get_column.....	101	edkit_sequence_get_subseq.....	100
edkit_get_num_chars.....	101	edkit_sequence_insert.....	99
edkit_get_num_lines.....	101	edkit_sequence_move.....	100
edkit_get_string.....	101	edkit_set_column.....	101
edkit_init.....	100	edkit_swap_point_and_mark.....	101

## Índice conceptual

### ASCII, código

ASCII, código ..... 27

### ASCII, repertorio

ASCII, repertorio ..... 26

### Boyer-Moore, algoritmo

Boyer-Moore, algoritmo ..... 52

### buffer gap

buffer gap ..... 61

### buffer, estructuración externa

buffer, estructuración externa ..... 56

### buffer, estructuración interna

buffer, estructuración interna ..... 56

### buffer, vistas de un

buffer, vistas de un ..... 85

### buffers de texto

buffers de texto ..... 55

### caracter

caracter ..... 4

### caracteres, codificaciones de

caracteres, codificaciones de ..... 28

### caracteres, códigos de

caracteres, códigos de ..... 27

### caracteres, repertorios de

caracteres, repertorios de ..... 26

### CCS

CCS ..... 29, 57

### CES

CES ..... 29

### character encoding scheme

character encoding scheme ..... 29

### coded character set

coded character set ..... 29

### codigos, pagina de

códigos, página de ..... 31

### componente de display

componente de display ..... 18

### componente lexico

componente léxico ..... 10

### componente semantico

componente semántico ..... 10

### componente sintactico

componente sintáctico ..... 10

### Crowley, comparativa de

Crowley, comparativa de ..... 76

### de Montgomery, teclado

de Montgomery, teclado ..... 9

### dispositivo localizador

dispositivo localizador ..... 9

### dispositivo, de estado

dispositivo, de estado ..... 9

### dispositivo, de texto

dispositivo, de texto ..... 9

**documento objetivo**

documento objetivo ..... 4

**dvorak, teclado**

dvorak, teclado ..... 9

**ed**

ed ..... 11

**edicion, de texto**

edición, de texto ..... 4

**edicion, proceso de**

edición, proceso de ..... 4

**editor ,orientado a streams**

editor ,orientado a streams ..... 14

**editor, descomposicion funcional**

editor, descomposición funcional ..... 18

**editor, orientado a lineas**

editor, orientado a líneas ..... 13

**editor, orientado a paginas**

editor, orientado a páginas ..... 13

**editor, origen y evolucion**

editor, origen y evolución ..... 21

**editor**

editor ..... 4

**estructuracion fuerte**

estructuración fuerte ..... 13

**estructuracion impositiva**

estructuración impositiva ..... 13

**filtrado**

filtrado ..... 4

**Finseth, comparativa de**

Finseth, comparativa de ..... 70

**Fuerza bruta, algoritmo**

Fuerza bruta, algoritmo ..... 46

**gap, buffer**

gap, buffer ..... 61

**interactivo, editor**

interactivo, editor ..... 4

**interfaz de usuario**

interfaz de usuario ..... 5

**interfaz orientada a menus**

interfaz orientada a menús ..... 12

**Interfaz orientada a teclas**

interfaz orientada a teclas ..... 11

**interfaz orientada a texto**

interfaz orientada a texto ..... 11

**ISO 10646**

ISO 10646 ..... 32, 33

**ISO 646, codigo**

ISO 646, código ..... 27

**ISO 646, repertorio**

ISO 646, repertorio ..... 26

**ISO 8859-1**

ISO 8859-1 ..... 31

**ISO 8859-10**

ISO 8859-10 ..... 31

**ISO 8859-11**

ISO 8859-11 ..... 31

<b>ISO 8859-12</b>		<b>Knuth-Morris-Pratt, algoritmo</b>	
ISO 8859-12 . . . . .	31	Knuth-Morris-Pratt, algoritmo . . . . .	47
<b>ISO 8859-13</b>		<b>Latin 1, codigo</b>	
ISO 8859-13 . . . . .	31	Latin 1, código . . . . .	31
<b>ISO 8859-14</b>		<b>Latin 10, codigo</b>	
ISO 8859-14 . . . . .	31	Latin 10, código . . . . .	31
<b>ISO 8859-15</b>		<b>Latin 2, codigo</b>	
ISO 8859-15 . . . . .	31	Latin 2, código . . . . .	31
<b>ISO 8859-16</b>		<b>Latin 3, codigo</b>	
ISO 8859-16 . . . . .	31	Latin 3, código . . . . .	31
<b>ISO 8859-2</b>		<b>Latin 4, codigo</b>	
ISO 8859-2 . . . . .	31	Latin 4, código . . . . .	31
<b>ISO 8859-3</b>		<b>Latin 5, codigo</b>	
ISO 8859-3 . . . . .	31	Latin 5, código . . . . .	31
<b>ISO 8859-4</b>		<b>Latin 6, codigo</b>	
ISO 8859-4 . . . . .	31	Latin 6, código . . . . .	31
<b>ISO 8859-5</b>		<b>Latin 7, codigo</b>	
ISO 8859-5 . . . . .	31	Latin 7, código . . . . .	31
<b>ISO 8859-6</b>		<b>Latin 8, codigo</b>	
ISO 8859-6 . . . . .	31	Latin 8, código . . . . .	31
<b>ISO 8859-7</b>		<b>Latin 9, codigo</b>	
ISO 8859-7 . . . . .	31	Latin 9, código . . . . .	31
<b>ISO 8859-8</b>		<b>lenguaje de comandos</b>	
ISO 8859-8 . . . . .	31	lenguaje de comandos . . . . .	17
<b>ISO 8859-9</b>		<b>lenguaje de interaccion</b>	
ISO 8859-9 . . . . .	31	lenguaje de interacción . . . . .	10
<b>ISO-8859, familia</b>		<b>lexema</b>	
ISO-8859, familia . . . . .	29	lexema . . . . .	10
<b>KMP, algoritmo</b>			
KMP, algoritmo . . . . .	47		

<b>localizaciones</b>		<b>pantalla mapeada en memoria</b>	
localizaciones . . . . .	79	pantalla mapeada en memoria . . . . .	9
<b>marca estacionaria</b>		<b>PDP-1</b>	
marca estacionaria . . . . .	80	PDP-1 . . . . .	21
<b>marca regular</b>		<b>puntero de edicion</b>	
marca regular . . . . .	80	puntero de edición . . . . .	16
<b>marca sticky</b>		<b>puntero</b>	
marca sticky . . . . .	80	puntero . . . . .	79
<b>marca</b>		<b>QED</b>	
marca . . . . .	79	QED . . . . .	21
<b>memoria de video</b>		<b>querty, teclado</b>	
memoria de vídeo . . . . .	10	querty, teclado . . . . .	9
<b>memoria principal</b>		<b>region</b>	
memoria principal . . . . .	10	región . . . . .	79
<b>modal, modo operacional</b>		<b>secuencia</b>	
modal, modo operacional . . . . .	12	secuencia . . . . .	4
<b>modelo conceptual</b>		<b>secuencias editables</b>	
modelo conceptual . . . . .	5	secuencias editables . . . . .	57
<b>modo comando</b>		<b>stream editor</b>	
modo comando . . . . .	12	stream editor . . . . .	14
<b>narrowing</b>		<b>tablas de piezas</b>	
narrowing . . . . .	84	tablas de piezas . . . . .	67
<b>pagina de codigos</b>		<b>Tape editor and corrector</b>	
página de códigos . . . . .	31	Tape editor and corrector . . . . .	21
<b>pantalla basica</b>		<b>TECO</b>	
pantalla básica . . . . .	9	TECO . . . . .	21
<b>pantalla grafica</b>		<b>teletipo</b>	
pantalla gráfica . . . . .	10	teletipo . . . . .	9

**Text editor and corrector**

Text editor and corrector ..... 21

**texto, buffers de**

texto, buffers de ..... 55

**tramo**

tramo ..... 80

**tramos, árbol de**

tramos, árbol de ..... 82

**tramos, grafos de**

tramos, grafos de ..... 83

**tramos, lista de**

tramos, lista de ..... 81

**TTY**

TTY ..... 9

**UCS, repertorio**

UCS, repertorio ..... 32, 33

**UCS-2, codificación**

UCS-2, codificación ..... 36

**UCS-4, codificación**

UCS-4, codificación ..... 36

**unicode, codificaciones**

unicode, codificaciones ..... 36

**unicode**

unicode ..... 32, 35

**UTF-16, codificación**

UTF-16, codificación ..... 37

**UTF-8, codificación**

UTF-8, codificación ..... 38

**vista sesgada**

vista sesgada ..... 18

**vista**

vista ..... 4

**vistas**

vistas ..... 85

**widening**

widening ..... 84

**F**

FDL, GNU Free Documentation License ..... 105

## Bibliografía

- [AEN80] P. Anandan, D. W. Embley, and G. Nagy. An application of file-comparison algorithms to the study of program editors. *International Journal of Man-Machine Studies*, 13(2):201–211, 1980.
- [AHU76] A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23(1):1–12, January 1976.
- [All83] Lloyd Allison. Syntax directed program editing. *Software—Practice and Experience*, 13(5):453–465, May 1983.
- [AS83] R. B. Allen and M. W. Scerbo. Details of command-language keystrokes. 1983.
- [Ber69] Gerald M. Berns. Description on format, a text-processing format. 1969.
- [Cap85] Michael Caplinger. Structured editor support for modularity and data abstraction. 1985.
- [CL81] C. C. Charlton and P. H. Leng. Editors: Two for the price of one. *Software—Practice and Experience*, 11(2):195–202, February 1981.
- [Coh80] Ellis Cohen. Text-oriented structure commands for structure editors. 1980.
- [CRC00] Gregory Crane and Jeffrey A. Rydberg-Cox. New technology and new roles: The need for corpus editors. 2000.
- [Cro98] Charles Crowley. Data structures for text sequences. 1998.
- [Dav82] D. J. M. Davies. String searching in text editors. *Software—Practice and Experience*, 12(8):709–717, August 1982.
- [DHZ85] Frank J. Dudinsky, Richard C. Holt, and Safwat G. Zaky. SRE: A syntax recognizing editor. *Software—Practice and Experience*, 15(5):489–497, May 1985.
- [DJB87] Jr Donald J. Bagert. A multi-language syntax-directed editor. 1987.
- [Fin98] Craig A. Finseth. Craft of text editing. 1998.
- [Gos81] J. Gosling. A redisplay algorithm. In P. Abrahams, editor, *Text manipulation: Proceedings of the ACM SIGPLAN/SIGOA symposium (Portland, OR, June 8–10, 1981)*, pages 123–129, New York, NY, USA, 1981. ACM Press.
- [Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.
- [Hir77] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Jrnl. A.C.M.*, 24(4):664–675, 1977.
- [HN88] Harrison and Notkin. Benchmarking file differencing algorithms. Technical Report 88-06-03, University of Washington, 1988.
- [HS77] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.
- [KW92] Derek Kiong and Jim Welsh. Incremental semantic evaluation in language-based editors. *Software—Practice and Experience*, 22(2):111–135, February 1992.
- [MD82a] Norman Meyrowitz and Andries Van Dam. Interactive editing systems: Part i. 1982.
- [MD82b] Norman Meyrowitz and Andries Van Dam. Interactive editing systems: Part ii. 1982.

- [Mil87] Webb Miller. *A Software Tools Sampler*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.
- [MLSHP81] S. Nudelman M. L. Schneider and K. Hirsh-Pasek. An analysis of line numbering strategies in text editors. 1981.
- [MM85] Webb Miller and Eugene W. Myers. A file comparison program. *Software—Practice and Experience*, 15(11):1025–1040, November 1985.
- [MM88] Webb Miller and Eugene W. Myers. A simple row-replacement method. *Software—Practice and Experience*, 18(7):597–611, July 1988.
- [MM89] E. W. Myers and W. Miller. Row replacement algorithms for screen editors. *ACM Transactions on Programming Languages and Systems*, 11(1):33–56, January 1989.
- [MM02] Robert C. Miller and Brad A. Myers. Multiple selections in smart text editing. 2002.
- [MN94] Toshiyuki Masui and Ken Nakayama. Repeat and predict - two keys to efficient text editing. 1994.
- [MP81] M. A. MacLean and J. E. L. Peck. CHEF: a versatile portable text editor. *Software—Practice and Experience*, 11(5):467–477, May 1981.
- [PD94a] M. Paterson and V. Dancik. Longest common subsequences. *Lecture Notes in Computer Science*, 841:127–??, 1994.
- [PD94b] M. S. Paterson and Vlado Dancik. Longest common subsequences. Research Report CS-RR-268, Department of Computer Science, University of Warwick, Coventry, UK, May 1994. This report has been published, and is not available online; a citation of the published version is available in BiBTeX format (citation.bib) by anonymous FTP from ftp://ftp.dcs.warwick.ac.uk/pub/reports/rr/268/.
- [Pik87] R. Pike. The text editor sam. *Software Practice and Experience*, 17(11):813–845, November 1987.
- [PM81] J. E. L. Peck and M. A. Maclean. The construction of a portable editor. *Software—Practice and Experience*, 11(5):479–489, May 1981.
- [SJDR90] Elli Mylonas Steven J. DeRose, David G. Durand and Allen H. Renear. What is text, really? 1990.
- [Ste87] Tom Steppe. File comparison algorithms. *Dr. Dobb's Journal of Software Tools*, 12(9):28–??, September 1987.
- [Thi89] Harold Thimbleby. A review of Donald C. Lindsay's text file difference utility, em diff. *Communications of the Association for Computing Machinery*, 32(6):756–755, June 1989. See iteWyk:1989:LPA.
- [Ukk83] Ukkonen. On approximate string matching. *FCT: Fundamentals (or Foundations) of Computation Theory*, 4, 1983.
- [Val93] Ray Valdes. Text editors: Algorithms and architectures. *Dr. Dobb's Journal of Software Tools*, 18(4):38, 40, 42–43, 80, April 1993.
- [WL89] Christopher J. Van Wyk and Donald C. Lindsay. Literate programming: A file difference program. *Communications of the Association for Computing Machinery*, 32(6):740–755, June 1989. See review iteThimbleby:1989:RDC.
- [ZM86] R. J. Zavodnik and M. D. Middleton. The design of yet another language-based editor. 1986.