Una base léxica para sistemas de procesamiento de lenguaje natural

José María Cañas Plaza

Escuela Técnica Superior de Ingenieros de Telecomunicación
Universidad Politécnica de Madrid
Madrid, 1995

RESUMEN.

El *sistema diccionario* realizado permite almacenar y recuperar eficientemente toda la información lingüística asociada a un determinado léxico. Para ello vuelca toda esa información, que inicialmente está expresada en un fichero de caracteres, a unas estructuras especialmente escogidas por su eficiencia: los haces de rasgos y el trie. Los programas de aplicación pueden consultar la información de la base léxica accediendo a esas estructuras a través de unas *interfaces de acceso* que se ofrecen. Una vez generadas éstas estructuras se vuelcan en unos ficheros, permitiendo su reutilización en distintos instantes de tiempo. Dependiendo de la plataforma hardware el sistema diccionario puede trabajar incorporando total o parcialmente el trie y los haces de rasgos a memoria, con lo que aprovecha la memoria disponible para acelerar su funcionamiento.

PALABRAS CLAVE.

Modelo computacional, léxico, rasgo, formante, formalismo, segmentación de existencia, trie, concatenación, cache de lectura, buffer de compilación, puntero virtual, polisemia, diccionario de datos, alomorfía, expansión, compilación, diccionario objeto, base léxica fuente.

A María,
Maribel,
Juani
y José.

Indice

1 PROCESAMIENTO DE LENGUAJE NATURAL. 8
1.1 INTRODUCCIÓN. 8
1.2 SISTEMAS DE LENGUAJE NATURAL. 9
1.2.1 Ambigüedad. 10
1.2.2 Proceso de comprensión. 11
1.2.3 Fuentes de conocimiento. 11
1.2.4 Interfaces de lenguaje natural 13
1.2.5 Aplicaciones. 14
1.3 PROCESADOR MORFOLÓGICO. 15
1.3.1 Morfología y morfemas 15
1.3.2 Alomorfos 16
1.3.3 Procesador flexivo basado en concatenación. 16
1.3.4 Procesamiento morfológico integrado en un diccionario. 18

2 BASE LÉXICA 20
2.1 INTRODUCCIÓN. 20
2.2 MODELO MORFOLÓGICO COMPUTACIONAL 23
2.2.1 Flexión verbal 25
2.2.2 Flexión nominal. 28
2.3 FORMALISMO LÉXICO 32
2.3.1 Caracteres permitidos y especiales. 35
2.3.2 Diccionario de datos. 35
2.3.3 Tipos de rasgos. 35
2.3.4 Macros o clases. 37
2.3.5 Lemas. 38
2.3.6 Reglas de alomorfía 39
2.3.7 Reglas de filtrado 40

2.4 FORMATOS DE LA BASE LÉXICA 41
2.4.1 Formato expandido 42
2.4.2 Formato objeto o compilado 43
2.4.3 El compilador. 45

3 ESTRUCTURAS DEL DICCIONARIO 47
3.1 INTRODUCCIÓN. 47
3.2 HACES DE RASGOS. 48
3.2.1 Diccionario de datos. 49
3.2.2 Tipos de rasgos. 53
3.2.3 Árboles binarios y balanceo. 53
3.2.4 Polisemia. 56
3.2.5 Interfaces de acceso. 56
3.2.6 Interfaz de original y de copia. 57
3.3 ESTRUCTURA CON LAS ENTRADAS 59
3.3.1 Tabla ordenada y búsqueda binaria. 59
3.3.2 Entradas como índices, el hash. 61
3.3.3 El trie. 63
3.3.4 El trie como doble array 65
3.3.5 El segmentador 72

4 DISCOS Y MEMORIA 75
4.1 INTRODUCCIÓN 75
4.2 DICCIONARIO LÓGICO 77
4.2.1 Interfaz de original e interfaz de copia. 79
4.3 USO DE MEMORIA PARA EL TRIE Y LOS HACES DE RASGOS. 81
4.3.1 El trie y la memoria. 81
4.3.2 Haces en consulta: la cache. 82
4.3.3 Haces en compilación: el buffer. 83
4.4 CACHES 83
4.4.1 Ubicación de bloques en la cache. 84
4.4.2 Búsqueda de bloques en la cache. 85
4.4.3 Sustitución de bloques ante fallos. 86
4.4.4 Lectura y escritura con cache. 87
4.5 CACHE DE CONSULTA. 87
4.5.1 Unidades de cache 87

4.5.2 Cache y dispersión de memoria 89
4.5.3 Tamaño de la cache 90
4.5.4 Cache no de escritura 90
4.5.5 Cache por conjuntos 91
4.5.6 Lectura de la cache 92
4.5.7 Tiempo de carga. 95
4.5.8 Algoritmo de sustitución. 95
4.5.9 Anomalía del reloj. 96
4.5.10 Cache infinita. 97
4.5.11 Traducción de punteros lógicos en consulta. 99
4.6 BUFFER DE COMPILACIÓN 99

5 ANÁLISIS DE PRESTACIONES 103
5.1 INTRODUCCIÓN 103
5.1.1 Módulo de pruebas. 104
5.2 PORTABILIDAD 105
5.3 EFECTO DEL BUFFER 108
5.4 EFECTO DE LA CACHE DE LECTURA 114
5.5 BASES EXPANDIDAS DE DISTINTOS TAMAÑOS 120
5.6 SEGMENTACIÓN. 122
5.7 EFECTO DE LAS COPIAS 124
5.7.1 Consultas con copia. 124
5.7.2 Trie con copia. 126

6 CONCLUSIONES Y PROPUESTAS. 128
6.1 CONCLUSIONES. 128
6.1.1 Eficiencia temporal y espacial. 129
6.1.2 Diccionario objeto. 130
6.1.3 Uso de la memoria de la plataforma. 131
6.1.4 Portabilidad 132
6.2 PROPUESTAS. 133

Indice de Figuras

1.1 Sistema genérico con interfaz de lenguaje natural. 13

2.1 Distintos formatos de la base léxica. 45

3.1 Las dos partes del diccionario: el trie y el conjunto de haces de rasgos. 48
3.2 Conversión del código 40 a su nombre asociado. 51
3.3 Conversión del nombre masc a su código asociado. 52
3.4 Array modificado. 52
3.5 Estructura rasgo. 54
3.6 Haces de rasgos 54
3.7 Balanceo de árboles. 55
3.8 Polisignificado como lista de haces. 57
3.9 Jerarquía de estructuras y los módulos que las manejan. . . . 58
3.10 Búsqueda binaria en una tabla ordenada.. 60
3.11 Nombre como índice. 61
3.12 Tabla Hash. 62
3.13 Trie como árbol de nodos con múltiples hijos. 63
3.14 Trie con los hijos en array. 65
3.15 Nodo y trie con hijos en lista. 66
3.16 Nodo A de paso y su hijo M que sí está en el doble array. . . . 67
3.17 Nodo A de paso cuando su hijo M no está en el doble array. . 68
3.18 Nodo 40 como nodo terminal. 69

4.1 Módulos de los dos interfaces 80
4.2 Jerarquía de memoria 84
4.3 Tipos de cache 85
4.4 Cache: tabla y memoria dinámica 88

4.5 Tabla de traducción de la cache 88
4.6 Consulta a una unidad que sí está en cache. 93
4.7 Consulta a una unidad que no está en cache. 94
4.8 Lectura con cache infinita. 98
4.9 Buffer de compilación. 100
4.10 Acceso a un puntero que está en el buffer. 101

5.1 Tiempo de compilación en SparcStation en función del tamaño del buffer. 109
5.2 Rendimiento del buffer en SparcStation. 111
5.3 Tiempo de compilación en PC en función del tamaño del buffer. 113
5.4 Tiempo de consulta en SparcStation frente a tamaño de conjuntos. 115
5.5 Tiempo de consulta en SparcStation en función del tamaño de cache. 117
5.6 Tiempo de consulta en PC con conjuntos de distinto tamaño. 119
5.7 Tamaño del trie y del fichero con haces. 121

Indice de Tablas

2.1 Codificación de formas verbales. 26
2.2 Alomorfos de la raíz de *acertar*. 27
2.3 Ejemplo de concatenación. 28
2.4 Ejemplo de *presidente*. 30
2.5 Ejemplo de *mesa*. 31
2.6 Irregularidades en la flexión nominal. 32
2.7 Fichero base léxica fuente. 34
2.8 Parte del diccionario de datos 36
2.9 Declaración de macros. 38
2.10 Invocación de macros. 38
2.11 Declaración de reglas de alomorfía. 39
2.12 Invocación de las reglas de alomorfía. 40
2.13 Declaración reglas de filtrado. 40
2.14 Ejemplo de lema en la base fuente 43
2.15 Ejemplo de alomorfos en la base expandida 44

3.1 Parte de la tabla de códigos. 50

4.1 Tiempos típicos de acceso. 77

5.1 Menú que el módulo de pruebas presenta al usuario. 105
5.2 Tiempos de segmentación en SparcStation. 123
5.3 Tiempos de consulta y copia en SparcStation con cache infinita. 125
5.4 Tiempos de consulta y copia en SparcStation con cache finita. 126

PREFACIO.

Uno de los aspectos comprendidos dentro de la Inteligencia Artificial es el Procesamiento del Lenguaje Natural (NLP). Desarrrollar sistemas que sean capaces de comprender y generar el lenguaje hablado o escrito es uno de los principales objetivos del NLP. El presente Proyecto Fin de Carrera ha tenido como objetivo la realización de un sistema diccionario que almacene eficientemente toda la información lingüística que puedan necesitar los diferentes procesadores que se utilizan en el NLP (morfológicos, sintácticos, etc.), y que permita recuperarla rápidamente.

Para ello procesa toda la información, que inicialmente reside en un fichero y la lleva a unas estructuras especialmente elegidas que ofrecen un acceso rápido a su contenido. El principal objetivo del sistema es la eficiencia temporal, es decir, que pueda responder a las consultas que se le hagan en el mínimo tiempo posible. Debe cuidarse también que las estructuras no ocupen mucho espacio para que nuestro diccionario pueda funcionar sin problemas en la mayoría de las plataformas hardware convencionales.

Otro de los objetivos perseguidos es la portabilidad. Se quiere que el sistema diccionario pueda funcionar en distintos tipos de máquina y arquitectura. Para ello todos los programas fuente se han escrito en lenguaje estándar ANSI C.

En el capítulo primero se hace una introducción de los sistemas de lenguaje natural como marco general en el que se encuadra el presente proyecto. Se comentan sus características principales, algunas aplicaciones y el lugar que ocupa en ellos un sistema diccionario como el realizado. También se presenta la figura del procesador morfológico, que será el usuario típico del sistema diccionario.

En el capítulo segundo se comenta el entorno concreto en el que se va a utilizar el sistema diccionario realizado. En primer lugar se describe el

modelo morfológico computacional con el que se ha trabajado, con la intención de familiarizarse con el tipo de información que el diccionario contiene. En segundo lugar se describe el formalismo léxico que se ha utilizado para expresar la información lingüística original, que nuestro diccionario procesará y ofrecerá a los programas usuarios.

En el tercer capítulo se justifican y describen las estructuras elegidas para almacenar toda la información lingüística, que son las que proporcionan al diccionario la eficiencia espacial y temporal. Las estructuras que se han escogido son los haces de rasgos y el trie, como se explicará en esta sección.

En el cuarto capítulo se explica en detalle los sistemas empleados para acelerar el funcionamiento normal del diccionario: un buffer de compilación y una tabla cache de lectura. Básicamente estos mecanismos consiguen mayor rapidez haciendo que el diccionario se aproveche de toda la memoria principal de que disponga la plataforma hardware sobre la que se está ejecutando.

En el capítulo quinto se comentan las diferentes pruebas que reflejan las prestaciones y el comportamiento de nuestro diccionario en distintas condiciones. Además de la finalidad descriptiva se comentan también los resultados de algunas pruebas que comparan distintas opciones de realización de nuestro diccionario, y que sirven para justificar que la elegida se ha seleccionado por ser la más eficiente.

Finalmente en el último capítulo se recogen las conclusiones de este Proyecto Fin de Carrera, y algunas propuestas de ampliación y mejora que permitirían continuar la labor iniciada.

Agradecimientos.

Quiero expresar mi agradecimiento a José Miguel Goñi Menoyo, tutor de este Proyecto Fin de Carrera, por la inestimable e inagotable ayuda prestada en la elaboración del mismo.

También quiero agradecer con especial intensidad a Angela Ribeiro y Luis Barrios la dedicación y paciencia que me han regalado. Igualmente quiero expresar mi gratitud a los amigos y amigas que de algún modo u otro me ayudaron en todo lo posible, entre ellos: Angel Luis González, José Ruiz Cristina, Fernando Casado, Carlos García y Lourdes de Agapito.

Gracias a todos.

Capítulo 1

PROCESAMIENTO DE LENGUAJE NATURAL.

1.1 INTRODUCCIÓN.

Uno de los aspectos comprendidos dentro de la Inteligencia Artificial es el **Procesamiento de Lenguaje Natural** (NLP). El principal objetivo del NLP es desarrollar sistemas que sean capaces de comprender y generar el **lenguaje natural** (LN), que es el que utilizan normalmente las personas para comunicarse entre sí. Por ejemplo el castellano, el inglés, el italiano, etc. son lenguajes naturales. El NLP se ocupa de formular e investigar mecanismos *computacionalmente efectivos* para la comunicación por medio del lenguaje natural [CarHay87].

Los sistemas de lenguaje natural son sistemas software en los que un subconjunto de su entrada y/o su salida está codificado en LN. El procesamiento de la entrada y/o la generación de la salida está basada en conocimiento sobre aspectos lingüísticos del LN. No son sistemas de LN los que procesan el lenguaje exclusivamente como cadenas de caracteres, como por ejemplo los editores de texto, o paquetes estadísticos.

La lingüística, como teoría general de la Gramática, elabora modelos que caracterizan las lenguas humanas, por lo que el NLP está muy relacionado con ella. La lingüística no considera la eficiencia computacional de estos modelos, ni los mecanismos concretos que permiten generar o interpretar el lenguaje. Sin embargo estos aspectos son fundamentales para el NLP. En

otras palabras, la lingüística tiene un enfoque teórico, descriptivo, mientras que el NLP adopta un punto de vista más práctico y operativo.

En este capítulo presentaremos los sistemas de lenguaje natural para introducir el marco genérico en el que se encuadra el presente proyecto.

1.2 SISTEMAS DE LENGUAJE NATURAL.

La cuestión principal que el NLP trata es la **comprensión**, que consiste en traducir una expresión en lenguaje natural, potencialmente ambigua, a una representación interna no ambigua, sobre la que sea posible razonar, hacer inferencias, etc. A ese formato inequívoco lo llamaremos *representación semántica*. La mayor dificultad de esta traducción reside en la **ambigüedad** que aparece asociada a las expresiones en lenguaje natural.

En el habla coloquial se dice que una expresión es ambigua cuando puede entenderse de varios modos, cuando puede admitir distintas interpretaciones. Del mismo modo en NLP diremos por ejemplo que una frase es ambigua si puede admitir diferentes representaciones semánticas, entre las cuales no se puede distinguir cual es la adecuada sin ayudarse de información adicional.

El problema del análisis se ha dividido en varios niveles jerárquicos de procesamiento, tradicionalmente: el morfológico, el sintáctico, y el semántico. Cada uno de ellos aborda un aspecto distinto del lenguaje y se apoya en un tipo de conocimiento lingüístico. Estos niveles se aplican jerárquicamente a unidades cada vez mayores: monemas (formantes de palabras), palabras y frases respectivamente, hasta llegar a la representación final, no ambigua, de la expresión del lenguaje natural.

El problema de la síntesis o **generación** de LN es justo el inverso de la comprensión. En la generación se parte de una representación semántica y se quiere llegar a una representación en lenguaje natural que la exprese adecuadamente. Los niveles en los que se descompone este problema son los mismos que en la comprensión, pero ahora se aplican en orden inverso y al revés. Si en el análisis la entrada al nivel morfológico fuera por ejemplo `casas`, su salida sería `sustantivo femenino plural del nombre casa`. En la generación sería justo al revés: la entrada puede ser una descomposición como: `primera persona del singular del pretérito simple indicativo del verbo tomar`, y su salida debe ser la palabra del lenguaje natural `tomé`.

1.2.1 Ambigüedad.

Como se señala en [GazMel89], uno de las principales dificultades que aparecen en el NLP es la ambigüedad. En general se presentará ambigüedad en los distintos niveles de procesamiento, y un nivel elimina parte de las ambigüedades que dejan sin resolver los niveles inferiores. Se eliminan ambigüedades al utilizar más conocimiento del que estaba accesible en los niveles inferiores. Por ejemplo la palabra `amo` puede tener al menos dos interpretaciones morfológicamente correctas, la que la categoriza como nombre (dueño) y la que la categoriza como verbo (primera persona de presente de indicativo del verbo `amar`). Cuando aparece en la frase `yo amo a María`, el análisis sintáctico, a la luz de las reglas gramaticales de la oración, descartará que `amo` pueda ser un sustantivo, y sí aceptará la interpretación como verbo. Sin embargo el nivel morfológico no tenía información suficiente para descartar la interpretación como nombre porque estudia la palabra aisladamente, no en el contexto de la frase.

Como se explica en [Goñ et.al92], algunos tipos de ambigüedad son los siguientes:

- *Ambigüedad sintáctica.* Aparece por ejemplo en `el hombre vió un niño con un telescopio`, donde el sintagma `con un telescopio` puede complementar a `vió` o a `niño`.

- *Ambigüedad semántica.* Por ejemplo en la frase `fui al banco`, no se sabe si `banco` se refiere al asiento, o al lugar para guardar el dinero.

- *Ambigüedad referencial.* Como la que aparece en `cogió la manzana de su mesa y se la comió`, donde el segundo `la` podría referirse tanto a `mesa` o a `manzana`. Para las personas resulta muy fácil distinguir esto, porque saben que las mesas no se comen. Es decir, resuelven esta ambigüedad semánticamente. Sin embargo la máquina no lo sabe si no tiene conocimiento que se lo indique, y entonces esa referencia de `la` es ambigua porque sintácticamente puede referirse a ambos, la `mesa` o la `manzana`.

- *Literalidad.* Por ejemplo cuando se pregunta `¿ puedes cerrar la puerta?` no se está cuestionando la capacidad de cerrar la puerta. La representación semántica adecuada es que se está pidiendo que se cierre la puerta.

1.2.2 Proceso de comprensión.

Como dijimos anteriormente el proceso de la comprensión se aborda en distintos pasos que consideran distintos aspectos:

Análisis morfológico: discrimina dentro de las palabras los elementos mínimos dotados de significado, que se llaman **monemas** o morfemas. Este análisis busca una partición de la palabra en monemas que sea válida morfológicamente. Rechaza las palabras para las que no se obtiene una descomposición correcta. Este análisis por ejemplo descompondría `cantamos` en `cant + amos`.

Análisis sintáctico: transforma una cadena de palabras en una estructura que muestra las relaciones entre ellas. Normalmente su salida son **frases** gramaticalmente correctas que se corresponden con la cadena de palabras de entrada. Como hemos visto anteriormente, puede recortar la ambigüedad de las diferentes interpretaciones morfológicamente válidas para cada palabra.

Análisis semántico: asigna significado a las estructuras que le presenta el análisis sintáctico, relacionándolas con los objetos de un dominio. Este nivel elimina ambigüedades como la interpretación de `banco` en `fui a un banco a sentarme`, ya que es válido sentarse sobre un banco cuando significa asiento, pero no es semánticamente correcto hacerlo sobre un banco cuando significa lugar donde se guarda dinero.

Análisis pragmático: resuelve la ambigüedad que hemos llamado literalidad. Reinterpreta la estructura que expresa lo que se enunció para determinar lo que se pretendió decir.

Integración del discurso: el significado de una oración puede depender de las oraciones que la preceden y puede influir en el significado de las que la siguen. Incluye la resolución de dependencias distantes, como el sujeto en la segunda frase: `Juan vino. Me lo contó todo`.

1.2.3 Fuentes de conocimiento.

Cada uno de los distintos niveles de procesamiento aplica cierto conocimiento lingüístico. En los niveles clásicos de morfología y sintáxis este conocimiento

se divide en una gramática y en un léxico. El **léxico** almacena las unidades individuales que se van a analizar en ese nivel, junto con cierta información asociada a cada unidad. La **gramática** es un conjunto de reglas que enuncian las posibles combinaciones entre unidades, basándose en la información asociada a cada unidad. Existe cierta complementariedad entre la complicación de las reglas y la cantidad de información que hay que asociar al léxico. Si queremos que las reglas sean pocas y sencillas, habrá que incorporar mucha información al léxico. Este punto de vista se llama **enfoque lexicalista** porque tiende a concentrar la información, y por lo tanto la complejidad, en el léxico más que en la gramática.

En el nivel sintáctico el léxico, las unidades individuales, son las palabras, cada una de ellas con cierta información gramatical asociada, como por ejemplo su género, su número, su categoría gramatical (verbo, adjetivo, sustantivo, etc.). La *gramática de la oración* es el conjunto de reglas que enuncian las posibles combinaciones entre palabras. Además de decir que tal o cual cadena de palabras es gramaticalmente válida o no lo es, la gramática proporciona una estructura para las oraciones correctas.

En el nivel morfológico las unidades básicas son los monemas o formantes de palabras, cada uno de ellos con cierta información asociada. Esa información es la que aparece en la *gramática de la palabra* para especificar que tal combinación de monemas es correcta o no.

El presente Proyecto Fin de Carrera realiza un **diccionario** que almacena los formantes de palabras y toda su información asociada. También almacena toda la información léxica que se puedan necesitar en los niveles sintáctico y superiores del NLP. Está pensado para trabajar cooperativamente con un procesador morfológico.

En el nivel semántico el conocimiento se puede expresar como un modelo del mundo y un conjunto de *reglas semánticas* que pongan en relación las cadenas del lenguaje con los conceptos del mundo. El conocimiento semántico necesario depende del dominio del mundo real en el cual queremos que funcione nuestro sistema, y por lo tanto depende estrechamente de la aplicación concreta que se vaya a desarrollar. Dentro del modelo del mundo se incluyen ontologías (definición de categorías semánticas básicas), modelos del dominio, del usuario, etc que ayudan en el procesamiento de los significados de las expresiones en lenguaje natural.

1.2.4 Interfaces de lenguaje natural

Los sistemas de LN se pueden utilizar sobre otros sistemas software para darles capacidad de comunicarse en LN con su entorno, bien admitir como entrada texto en LN, bien expresar su salida en LN, o ambas a la vez.

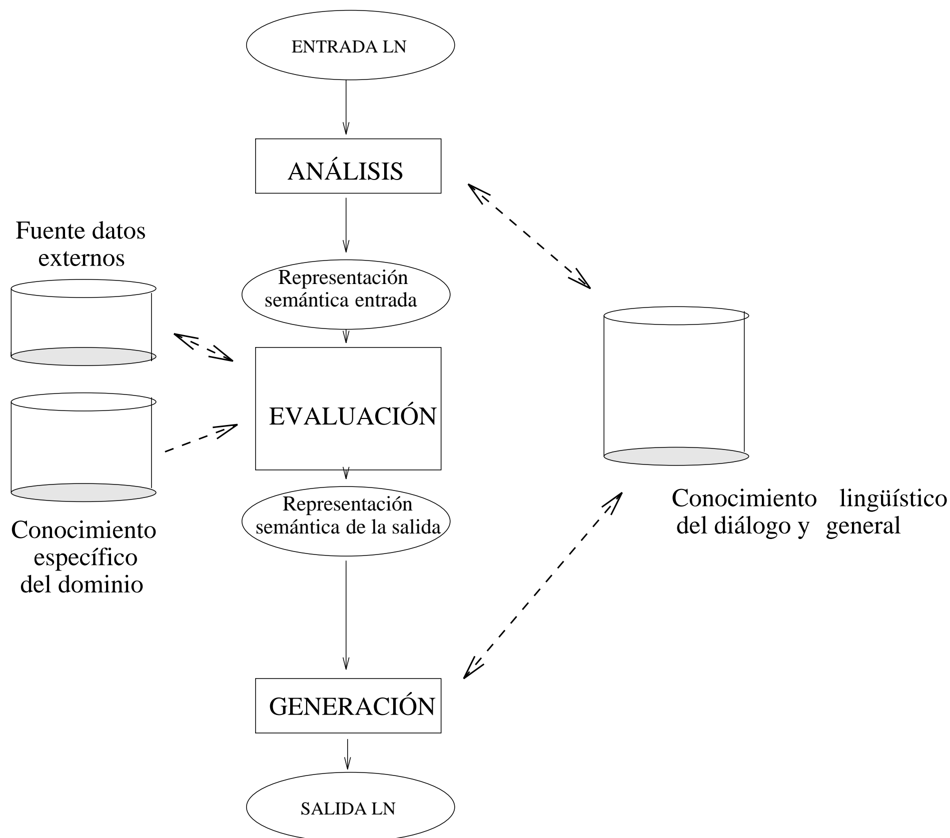

Figura 1.1: Sistema genérico con interfaz de lenguaje natural.

Tal como se explica en [Goñ et.al92], la figura 1.1 muestra la arquitectura de un sistema genérico con subsistema de LN. En la entrada del sistema global se tiene lenguaje natural, que el subsistema de LN analiza para llegar a su correspondiente representación semántica. Ésta se procesa para realizar la aplicación concreta, consultando para ello las bases de conocimiento pertinentes, dependientes del dominio de la aplicación. Una vez que este subsistema

evaluador procesa su entrada, puede producir una salida, en representación semántica. Desde ella se genera la salida en lenguaje natural correspondiente.

El subsistema de LN consulta sus bases de conocimiento lingüístico, entre las que se encuentra una base léxica o diccionario, tanto en su etapa de análisis como en la de generación.

En otras palabras tenemos dos lenguajes, el natural y el semántico del dominio de la aplicación, que es no ambiguo. El sistema de la aplicación sólo entiende ese lenguaje no ambiguo, pero como se quiere que comprenda y se exprese en lenguaje natural, se utiliza un subsistema de LN que traduzca en ambos sentidos: de LN a lenguaje de aplicación en las entradas, y al revés en las salidas. El subsistema de LN provee a la aplicación de una **interfaz de lenguaje natural**.

1.2.5 Aplicaciones.

Un grupo muy importante de aplicaciones de los sistemas de LN es precisamente el de ofrecer interfaces de LN a otros sistemas software. Así por ejemplo las *bases de datos* podrían presentar una interfaz en LN que aceptara consultas en lenguaje natural, con lo que serían más fáciles de manejar para las personas. Esas consultas se analizarían y traducirían por ejemplo a SQL (Soft Query Language), que es un lenguaje no ambiguo de consultas que la propia base de datos ya entiende directamente.

También se podría incorporar sistemas de LN a los *sistemas operativos* de manera que fuera más sencillo para las personas utilizar las máquinas. Con NLP parte de la complejidad del entendimiento entre el ordenador y su usuario humano se trasladaría a la máquina. El usuario ya no tendría que comunicarse con la máquina en el lenguaje de los comandos del sistema operativo, sino que le bastará hacerlo en su lenguaje humano, porque con el NLP la máquina le va a entender.

Otra aplicación muy útil de los sistemas de LN es de **generadores de explicaciones**. Por ejemplo para expresar las conclusiones de un sistema experto. Este sistema procesa una serie de datos y elabora una conclusión o análisis. Para que este resultado sea entendido más fácilmente por las personas que utilizan el sistema experto se puede expresar en lenguaje natural.

Una aplicación muy anhelada históricamente es la **traducción automática** de una lengua humana a otra. Este objetivo se encuadra en el esquema planteado como un proceso en dos pasos. Primeramente se analiza texto

en el lenguaje original, llegando a la representación semántica no ambigua. En un segundo paso se genera el texto en lenguaje destino que corresponde a esa misma representación semántica. Así ambos textos naturales son semánticamente equivalentes porque corresponden a la misma representación semántica, uno es la traducción del otro.

1.3 PROCESADOR MORFOLÓGICO.

1.3.1 Morfología y morfemas

Como hemos visto uno de los niveles del NLP es aquel en el que se procesan los aspectos morfológicos.

Desde un punto de vista clásico la Morfología se considera la parte de la Gramática encargada del estudio de la flexión, la composición y la derivación de las palabras, determinando categorías gramaticales. Por un lado la Morfología estudia las unidades existentes dentro de la palabra, menores que ésta, y las relaciones entre ellas. Por otro estudia la categorías léxicas, esto es, si las palabras se caracterizan como sustantivo, verbo, adjetivo, preposición, etc.

En los textos tradicionales los formantes morfológicos se definen como las unidades mínimas de análisis gramatical dotadas de significación, y se llaman monemas. Estos monemas se clasifican en lexemas y morfemas. Se dice que son lexemas si tienen significado pleno, es decir, representan un concepto o una idea. Por el contrario los morfemas no tienen significado pleno y relacionan a los lexemas o modifican su significación, teniendo además un significado gramatical. Los morfemas significan cosas como *plural, femenino, subjuntivo, etc.*. Por ejemplo, en la palabra *pensamos*, el lexema es *pens* y el morfema *amos*. En otros textos clásicos, sin embargo, se utiliza morfema como sinónimo de monema, y así se utilizará también en esta memoria.

Según Hockett [Mor92] se distinguen tres enfoques generales de la morfología:

Palabras y Paradigma : En este enfoque se clasifican las palabras por categorías y por paradigmas (grupo de palabras que sirven de modelo o clase, con las mismas particularidades morfológicas), y se establecen clase paradigmáticas sobre lexemas que hacen de modelo (por ejemplo

amar para la primera conjugación, etc.). Los formantes en este caso son las raíces o lexemas y las terminaciones o accidentes.

Elementos y colocación : En este modelo se considera que las palabras se pueden segmentar en unidades *abstractas*, que son los morfemas, relacionados entre sí por mera sucesión (un morfema tras otro). Según Moreno [Mor92] el análisis morfológico bajo este modelo se reduciría a:

1. Una especificación del inventario de morfemas.
2. Una especificación de las secuencias en las que estos morfemas pueden aparecer.
3. La especificación del morfo o alomorfo (variantes de un morfema) que pueden realizar cada morfema.

Elementos y Proceso : El modelo consta de elementos (morfemas léxicos), sobre los cuales actúan un menor número de morfemas (flexivos, derivativos) que modifican a los primeros. A estos cambios que afectan a la raíz léxica dando lugar a la llamada forma superficial, se les denomina procesos morfológicos.

1.3.2 Alomorfos

Los **alomorfos** son las variantes de un morfema en determinados entornos, las formas gráficas concretas en las que aparece un morfema, entendido éste como unidad morfológica abstracta. Tal como se explica en [Gon95], el morfema negativo {`in-`} aparece con la forma `i-` delante de `l-` y `r-`, como en *irrespirable* o *ilegal*; aparece con la forma `im-` delante de `p-` o `b-`, como en *improbable* e `imbatido`; y como `in-` en los demás casos. Estos son, por tanto los tres alomorfos del morfema negativo {`in-`}.

1.3.3 Procesador flexivo basado en concatenación.

Como ilustran [Goñ et.al95b] y [Goñ et.al95a], en el *Grupo de Procesamiento de Lenguaje Natural* de la ETSI Telecomunicación de la Universidad Politécnica de Madrid se ha desarrollado un procesador morfológico que trata la flexión del castellano, con el cual ha de cooperar la base léxica que realiza este proyecto. Dedicaremos este apartado a presentar ese procesador, porque

él determina en gran medida el contenido de la base léxica o diccionario que hemos realizado.

Este procesador morfológico utiliza un modelo computacional del tipo *elementos y colocación*, basado en la concatenación de morfemas y en la unificación de rasgos, que son los elementos portadores de la información léxica y gramatical. El procesador se describe con detalle en [Gon95] y en [Gon et.al95].

El procesador considera las palabras en su representación escrita, es decir, una cadena de caracteres (en particular, letras) separada por blancos. Trabaja con **palabras gráficas**. Esta decisión condiciona que el tratamiento sea distinto al que se daría manejando palabras fonológicas. Por ejemplo, al considerar las palabras escritas, sí se considera que hay variación alomórfica de la raíz entre `sac-o` y `saqu-e`, aunque esta variación no exista desde el punto de vista fonético: `/sak-o/` y `/sak-e/`.

Como se explica en [Gon95], el procesador morfológico dispone de dos componentes:

El diccionario o léxico: es un conjunto de entradas, que pueden ser palabras o formantes morfológicos de las mismas, así como sus alomorfos. Cada entrada del léxico tendrá asociada información como puede ser la categoría léxica, el tiempo, la persona, el número, etc. El procesador consultará el léxico con el objeto de reconocer los distintos formantes morfológicos que componen una palabra, y extraer la información asociada a los anteriores.

Las reglas : recogen las regularidades morfológicas de una determinada lengua, estableciendo esquemas de combinación de los formantes morfológicos para poder formar palabras (generación) o comprobar la estructura de las ya formadas (análisis). Especifican la *gramática de la palabra.*

Con la aplicación de las reglas los morfemas se *concatenan*, es decir, sus cadenas superficiales se yuxtaponen, una a continuación de la otra.

El análisis morfológico se divide en dos etapas. En una primera etapa se particiona la palabra en todas las posibles combinaciones de formantes que la completen. A estas particiones se las llama **segmentaciones de existencia**, porque lo único que se pide es que cada formante tenga existencia como unidad en el diccionario y que entre todos los formantes o segmentos

se complete la palabra. En esta primera fase no se considera la información asociada a cada morfema. Por ejmplo la palabra `maridos` se puede descomponer en `marido+s` y en `mar+idos` cuando todos esos formantes (`marido, s, mar, idos`) existen en el léxico.

En una segunda etapa se recupera la información léxica y gramatical asociada a la palabra aplicando las reglas morfológicas a cada descomposición generada en la etapa anterior. La aplicación de las reglas provoca que si la segmentación es morfológicamente válida se combinen los rasgos de los formantes para generar los rasgos con la información de la palabra. Si la descomposición anterior no satisgace las reglas entonces no es morfológicamente válida y se desecha. En el ejemplo anterior la aplicación de las reglas descartaría `mar+idos` y obtendría que `marido = marido + s` es un sustantivo masculino plural.

Dos de los procesos más significativos en la formación de palabras son la flexión y la derivación. La **flexión** es el proceso en el que se unen los morfemas flexivos a palabras o a temas (raíces de los vocablos que no constituyen palabras por sí mismas). Son morfemas flexivos del castellano los de género, número y las distintas desinencias verbales. El procesador desarrollado en [Gon95] es capaz de tratar computacionalmente la flexión y ello determina que el contenido del diccionario sean principalemente lexemas y morfemas flexivos. Concretamente distingue entre flexión verbal (o conjugación), que se realiza con verbos, y la flexión nominal (o declinación) si se hace con formas que admiten género y/o número, como nombres, adjetivos, etc.

La **derivación** se puede definir como la formación de palabras adjuntando morfemas derivativos [Yll et.al83], como por ejemplo `-able, -ero, -ción`. El procesador morfológico desarrollado no trata la derivación porque esta ofrece una mayor complejidad que la flexión y no se dispone de un modelo lingüístico satisfactorio, aunque si existiera podría ser incorporado a nuestro modelo computacional.

1.3.4 Procesamiento morfológico integrado en un diccionario.

Como se ha mencionado anteriormente, el nivel morfológico es la plataforma sobre la que trabaja el analizador sintáctico, porque le ofrece a éste toda la información gramatical y léxica que necesita. Por ejemplo la categoría grama-

tical de cada palabra. Desde este punto de vista el procesador morfológico más sencillo es aquel en el que todas las palabras ya están flexionadas en el diccionario. No tiene ninguna regla, una palabra será morfológicamente válida si aparece como tal en el diccionario y sus rasgos se ofrecen directamente al nivel sintáctico. Este enfoque es rapidísimo pero tiene el inconveniente de necesitar diccionarios extraordinariamente grandes.

En lugar de almacenar las distintas las raíces nominales y los morfemas de género y número, tendrían que guardarse todos las formas nominales flexionadas (`niño, niños, niña, niñas`). Con los verbos el factor multiplicativo del número de entradas sería aún mayor: en vez de almacenar las raíces y desinencias verbales, tendrían que guardarse para cada raíz todas las distintas maneras en que se conjuga según el modo, tiempo, modo, persona y número (`hablo, hablas, habla, hablamos, etc`). Esta opción se ha descartado por necesitar diccionarios muy grandes, y en su lugar se prefiere realizar la función de suministrar información gramatical y léxica al nivel sintáctico mediante un diccionario de formantes, más pequeño, trabajando conjuntamente con un procesador morfológico.

Capítulo 2

BASE LÉXICA

2.1 INTRODUCCIÓN.

La *base léxica* es una de las fuentes de conocimiento que se necesita en las distintas etapas del procesamiento del lenguaje natural. Básicamente es un conjunto de unidades léxicas o entradas, cada una de ellas con cierta información asociada. Nosotros distinguiremos entre base léxica y diccionario. Llamaremos **sistema diccionario** al conjunto de estructuras y herramientas que permiten almacenar una gran cantidad de información, recuperarla de módo rápido cuando se necesita y manipularla eficientemente. Llamaremos **base léxica** a la información concreta que el diccionario almacena, esto es, las entradas y sus datos asociados.

El diccionario no crea la información, sólo agiliza el acceso. Toda la información reside inicialmente en un fichero que llamamos base léxica fuente. El sistema diccionario la cambia de formato y la procesa convenientemente, de modo que la información pueda ser visible a los distintos programas de aplicación que la utilizan y todo ello pueda funcionar en una plataforma hardware convencional de un modo suficientemente veloz.

Rasgos.

La información que puede contener la base léxica podrá ser morfológica, sintáctica y semántica, aunque a efectos de formato todos se reflejan en **rasgos**, esto es, *variables* que toman tal o cual valor al particularizarse en cada item del diccionario. Por ejemplo el rasgo *num* puede indicar el núme-

ro y al particularizarse para una entrada del diccionario señalará si esta es singular (*sing*) o plural (*plu*), que son los posibles valores del rasgo.

Los rasgos son las unidades mínimas de información asociables a un vocablo. Cada item llevará ligado un conjunto de rasgos que caracterizan la información léxica y/o gramatical asociada a él. El sentido de estos rasgos lo darán las aplicaciones que los usen, la base léxica se limita a almacenarlos. Los rasgos con información sintáctica se utilizarán en el análisis y generación de oraciones, y los rasgos con datos morfológicos los empleará un procesador en el análisis y generación de palabras desde sus morfemas. Un mismo rasgo puede necesitarse en varios niveles del procesamiento de lenguaje natural.

Formalismo lexico, base lexica fuente y diccionario

Conviene distinguir entre formalismo léxico, base léxica fuente, sistema diccionario y diccionario objeto porque son términos relacionados pero no equivalentes.

La **base léxica fuente** es el fichero en el que reside toda la información lingüística que tenemos y podemos necesitar. Es decir en ella están todas las entradas con sus rasgos asociados.

El **formalismo léxico** es la especificación del fomato en el que aparece la información en el fichero base léxica fuente. Por ejemplo que para asociar el rasgo femenino a la terminación "a' basta poner en una linea "a' seguido de retorno de carro y en la siguiente "gen" (*género*) seguido de un "=' y seguido de "fem' (*femenino*). Con el formalismo se define qué sentido tienen determinados caracteres dentro del fichero base léxica fuente, y en general las reglas de formato para expresar la información lingüística que tenemos.

Como sistema software el **sistema diccionario** es un grupo de estructuras, funciones y utilidades que permiten el almacenamiento, el procesamiento y el rápido acceso a la información de una base léxica en los sistemas computadores convencionales.

Y se llama **diccionario objeto** a cada uno de los subconjuntos concretos, que de la base léxica fuente se pueden extraer y que residen en unas estructuras de almacenamiento más operativas que un simple fichero de caracteres. Se puede extraer el total de los datos que tiene la base léxica o subconjuntos suyos, bien seleccionando algunos items o entradas, bien podando los haces de rasgos de datos que no se necesiten. El sistema diccionario proporciona unas funciones para el rápido acceso al contenido de cada diccionario objeto.

Ni el formalismo léxico ni las estructuras del diccionario determinan el tipo de gramática que vamos a usar coadyuvantemente con nuestro diccionario. No condicionan que el peso y la complejidad del análisis morfológico y del sintáctico resida más en el léxico que en las gramáticas. De hecho ni siquiera condicionan que el diccionario se integre con un procesador morfológico. Perfectamente podrían utilizarse el formalismo y las estructuras para una base léxica que sólo incluyera palabras enteras y no formantes.

El formalismo simplemente permite posibilidades de asociar información en forma de rasgos a una cadena de caracteres o item. El formalismo no enuncia qué nombres de rasgos deben participar, y sin embargo sí provee la forma en que estos se definen y utilizan.

Por su parte las estructuras del diccionario simplemente realizan esas posibilidades. Le dan a la información de la base léxica otro formato de acceso más rápido que la simple declaración como concatenación de caracteres ASCII en el fichero base léxica fuente.

Es pues el contenido de la base léxica concreta el que determina la filosofía que se sigue. Por ejemplo, si se adopta una postura lexicalista la complejidad de los análisis se traslada al léxico, al cual se le incorporará mucha información para que las reglas, gramaticales o morfológicas, operen sin dificultad. En este caso cada item llevará muchos rasgos asociados.

Por ejemplo si se quiere que el diccionario funcione con un procesador morfológico se incluirán en la base léxica los rasgos que aporten la información morfológica, y los items almacenados serán morfemas y formates de palabras, más que palabras enteras. Esta es la postura que se adoptó en el *Grupo de Procesamiento de Lenguaje Natural* en el que se encuadra este proyecto: la base léxica debía servir al procesador morfológico flexivo que presentamos en capítulo 1, y suministrarle todos los datos que necesitara.

En este capítulo describiremos el entorno concreto en el que se va a utilizar nuestro diccionario. Por un lado el formato de los ficheros que están relacionados con él y por otro comentaremos el tipo de información que la base léxica desarrollada en el *Grupo de Procesamiento de Lenguaje Natural* comprende. En general, el contenido del diccionario, esto es, las entradas almacenadas y los rasgos que se les asocian, serán los que necesiten los distintos modelos computacionales que la explotan, no sólo el morfológico, sino también

el sintáctico y el semántico. Quizá sea el esquema morfológico el que más fuertemente influya, porque es el que está más próximo lógicamente al diccionario, el de más *bajo nivel.* Por ejemplo a parte de influir en que los items guarden datos de concatenación morfológica, determina que se almacenen las cadenas de los formantes, no de palabras enteras.

2.2 MODELO MORFOLÓGICO COMPUTACIONAL

La base léxica está pensada para guardar información lingüística asociada al léxico. Y en concreto la que se ha desarrollado en el *Grupo de Procesamiento de Lenguaje Natural* en el que se encuadra este proyecto se ha pensado para responder en primera instancia a un procesador morfológico, el cual elabora la información que puedan requerir otros niveles superiores partiendo de la contenida en el léxico de la base. Por lo tanto la base léxica que almacena nuestro diccionario tendrá como entradas morfemas y formantes.

Además se ha seguido un **enfoque lexicalista**, que concentra complejidad e información en el léxico para que las reglas o gramáticas de los distintos niveles de tratamiento del lenguaje sean lo más sencillas posible. Por lo tanto las entradas de nuestro diccionario concentrarán mucha información morfológica y sintáctica.

Se pretende que el procesador morfológico capte las regularidades de la flexión nominal, tanto de género como de número, y las de la flexión verbal, tiempos, modos y personas. Para ello se ha seguido un modelo morfológico computacional basado en el desarrollado por Antonio Moreno Sandoval en [Mor92] y en [MorGoñ95]. Este modelo no trata la derivación ni la composición, sólo la flexión y para ello las palabras se consideran descompuestas sólo en dos tipos de formantes: raíces y terminaciones. El modelo está compuesto por un diccionario y una gramática. Esta **gramática de la palabra** contiene las reglas que combinan los elementos léxicos. Está basada en la concatenación de morfemas y en la **unificación** de rasgos. El diccionario es el lugar en el que se localizan los elementos terminales con sus rasgos asociados. Los rasgos portan la información ligada a los morfemas, incluyendo la necesaria para poder combinarlos. Como vimos en el capítulo 1 el modelo morfológico computacional se basa en un léxico con unos determinados

rasgos ligados y en unas reglas que gobiernan la concatenación de formantes apoyándose en esos rasgos. El procesador morfológico utiliza el modelo, lo aplica al análisis de palabras o a la generación de palabras desde información léxica y gramatical.

El hecho de que la base léxica sirva a un procesador morfológico se refleja en la inclusión en cada item de ciertos rasgos que este procesador maneja, en que se guarde menor número de entradas que si almacenara vocablos ya flexionados, y en el tipo de items, que no son palabras sino formantes, con lo cual la longitud media de esas entradas es menor.

Las entradas que almacena la base son de tres tipos:

1. **Raíces** verbales o nominales y sus alomorfos.

2. **Morfemas flexivos** y sus alomorfos. Son elementos con información gramatical insuficiente, que necesitan concatenarse entre sí para configurar una palabra.

3. **Formas lexicalizadas**. Son elementos léxicos con información gramatical suficiente que no se analizan morfológicamentte puesto que se consideran palabras. Es el caso de formas verbales muy irregulares o formas nominales especiales, como por ejemplo las que sólo aparecen en plural, *pluralia tantum* (*tijeras*), o como las que sólo lo hacen en singular, *singularia tantum* (*informática).*

En esta sección describiremos el significado de algunos rasgos importantes en el análisis morfológico para conocer mejor la información que nuestro diccionario almacenará.

Por ejemplo, para diferenciar las distintas variantes y concatenar correctamente los términos léxicos nos ayudamos del rasgo **concat** (*concatenación*), que puede valer:

w: para palabras o formas lexicalizadas que no necesitan concatenarse con ningún morfema flexivo, ni verbal ni nominal. Llevan incorporados los rasgos *gen* y *num* para llevar la información de su género y su número. Es el caso por ejemplo de los sustantivos *singularia* y *pluralia tantum*, como *informática* y *tijeras* respectivamente.

wl: para palabras que admiten morfema de número. Llevan el género asociado a la raíz. Por ejemplo *sol, reloj, silla.*

vl: para las raíces verbales.

vm: para los morfemas flexivos verbales.

nl: para las raíces nominales.

ng: para los morfemas de género, es decir *-a, -e* y *-o*.

nn: para los morfemas de número, es decir *-s* y *-es*.

2.2.1 Flexión verbal

El verbo se ha considerado únicamente descompuesto en dos formantes: la raíz verbal y el morfema flexivo verbal. Este último incluye toda la información referida al tiempo, modo, aspecto, persona, número, etc. Estos dos morfemas aparecerán en el diccionario y juntos formarán el verbo. Este esquema no recoge las formas compuestas, y opta por que se traten a nivel sintáctico, limitándose a indicar la primera palabra como una forma flexionada del verbo *haber*, y la segunda como el participio pasado del verbo principal.

El modelo computacional agrupa los verbos del castellano en paradigmas de flexión o de conjugación. Los que pertenezcan al mismo grupo se flexionan del mismo modo. Por otro lado codifica las distintas formas verbales según tiempo, modo, persona, etc. e incorpora a cada morfema los códigos de las formas en que se utiliza a través de dos rasgos: *sut* y *stt*. Estos rasgos indicarán para un morfema con qué otras cadenas superficiales se puede concatenar.

El modelo construye esa codificación con una tabla de formas verbales para identificar con códigos numéricos la forma verbal de un modo, un tiempo, una persona y un número. Ésta tabla, la 2.1, resume todas las combinaciones de cuatro rasgos:

tense: sus posibles valores son los tiempos verbales: *pres* para presente, *impf* para imperfecto, *indf* para indefinido y *fut* para futuro.

mood: sus valores son los modos verbales: *ind* para indicativo, *subj* para subjuntivo, *cond* para condicional, *imper* para imperativo, *inf* para infinitivo, *ger* para gerundio y *part* para participio.

pers: (persona) *1*,*2* y *3* para primera, segunda y tercera persona respectivamente.

	1sing	2sing	3sing	1plu	2plu	3plu	
pres ind	11	12	13	14	15	16	
impf ind	21	22	23	24	25	26	
ind ind	31	32	33	34	35	36	
fut ind	41	42	43	44	45	46	
pres subj	51	52	53	54	55	56	
impf subj	61	62	63	64	65	66	
cond	71	72	73	74	75	76	
imper		82/83			85/86		
inf							0
ger							90
part							99

Tabla 2.1: Codificación de formas verbales.

num: (número) *sing* para singular y *plu* para plural.

El código 100, que no aparece en la tabla se reserva para las raíces únicas como las de los verbos regulares, para no tener que escribir todos los códigos de la tabla. Y los códigos 83 y 86 corresponden a las formas del *imperativo de cortesía*, del singular y del plural respectivamente. Por ejemplo en las formas `diga` (usted) y `hablen` (ustedes).

A la hora de incluir esa información de concatenación en los formantes verbales se utiliza esta tabla y los dos rasgos antes mencionados:

stt: (tipo de raíz) identifica las terminaciones que admite una raíz verbal mediante su código numérico. Por ejemplo, como ilustra la tabla 2.2 el verbo *acertar* tiene dos alomorfos de la raíz la cadena *aciert* a la que se le asocian unos cuantos códigos de la tabla 2.1 y la cadena *acert* que se usa para el resto de los códigos, es decir, para el resto de formas verbales de tiempo, modo, ...

sut: (tipo de desinencia) clasifica varios alomorfos de la desinencia. Sus valores son simples cadenas alfanuméricas con el propósito de distinguir entre las diferentes variantes alomórficas de la desinencia de una forma verbal. A saber: *reg, pres, pret1, pret2, fut_cond, imp_subj, imper,*

```
aciert
stt = 11 12 13 16 51 52 53 56 82

acert
stt = 14 15 21 22 ...  54 55 ...  85 90 99 0
```

Tabla 2.2: Alomorfos de la raíz de *acertar*.

infin, ger, part1 y *part2*. Con el valor *reg* se señala a las desinencias de los paradigmas regulares de las tres conjugaciones.

El objetivo de estos dos rasgos es que sólo puedan concatenarse cada raíz con su correspondientes terminación, y no con el resto, aunque existan otras terminaciones que tengan la misma representación gráfica, es decir, la misma cadena superficial.

Tanto las raíces como cada desinencia llevan el código de la forma verbal en que se utilizan. Una raíz podrá concatenarse con desinencias que se utilicen para la misma forma verbal, esto es, con el mismo valor de *stt*.

De entre todos las terminaciones que por ejemplo se pueden utilizar para el tiempo 31 y que por lo tanto tienen `stt = 31`, sólo una de ellas se utilizará para una raíz concreta. Por ejemplo si el verbo es regular de la tercera conjugación, para formar la primera persona del pretérito indefinido de indicativo se utiliza la terminación *-í* ,como en *partí*. Sin embargo para el verbo *decir* se utiliza la terminación *-e*,para dar *dije*. Para poder seleccionar entre la desinencia adecuada se incorpora el rasgo *sut*, tanto a las raíces como a las terminaciones. De nuevo sólo se pueden concatenar si en ambos morfemas coinciden sus valores de *sut*.

Como ejemplo, supongamos que se debe analizar o generar la tercera persona del indefinido de indicativo del verbo *pedir*. Para este verbo tenemos dos entradas en el diccionario según muestra la tabla 2.3. Es esa tabla también aparecen las desinencias posibles para la forma 33, que es el código de la tercera persona del indefinido de indicativo.

De las posibles combinaciones sólo tendrá éxito *pid+ió*, ya que coinciden sus valores `stt = 33, sut = reg`. Por el contrario *ped+ió* fallará puesto que no existe el `33` en el `stt` de *ped*. También fallará *pid+o* porque difieren en el valor del `sut`.

ped
```
stt = 0 14 15 ...
sut = reg
```

pid
```
stt = 11 12 ...  33 ...
sut = reg
```

ió
```
stt = 33
sut = reg
```

o
```
stt = 33
sut = pret1
```

Tabla 2.3: Ejemplo de concatenación.

2.2.2 Flexión nominal.

La flexión nominal afecta a las **formas nominales**, que se definen como las que tienen género y número. Los sustantivos, los adjetivos y algunos pronombres se flexionan de este modo.

El modelo morfológico seguido, basado en el de Antonio Moreno Sandoval, [Mor92], distingue entre morfemas de género y morfemas de número. Así la palabra *niñas* se divide en *niñ-a-s* y no en *niñ-as* que es como sería si sólo admitieramos una único formante tras la raíz, tal como se hace con las terminaciones verbales. Algunas raíces admitirán la concatenación de dos morfemas como *chic-o, chic-a, chic-o-s, chic-a-s*. Otras en cambio sólo admitirán morfema de número porque el género lo lleva la raíz inherentemente, como *mesa, mesa-s*. También hay palabras que se utilizan indistintamente para singular y para plural, como *crisis* ; y los nombres *epicenos* que se usan tanto para designar al género femenino como al masculino, como *gorila*.

Género

La información gramatical sobre el género se especifica mediante el rasgo **gen** que puede ser *masc* para masculino o *fem* para el femenino. Su valor se refiere a la concordancia gramatical, no al sexo de la persona o animal. Este rasgo aparecerá dentro del léxico básico en el morfema de género y en las palabras de género inherente. En aquellas palabras cuyo género se determine por concordancia, como por ejemplo *testigo, artista, grande* y *verde*, este rasgo *gen* no aparecerá, y su valor se determinará en el análisis sintáctico.

Hay tres alomorfos para el morfema de género. Para distinguir si el morfema puede concatenarse con una raíz, y qué alomorfo puede hacerlo se utiliza el rasgo **get** (tipo de género) que puede valer:

mas1: para las raíces que llevan -*o* como morfema de masculino.

mas2: para las raíces que llevan -*e* como morfema de masculino.

fem: para las raíces que llevan el morfema de femenino -*a*.

no: para las raíces que no admiten morfema de género.

Este rasgo se incorpora tanto a la raíz como al morfema. En las raíces indica qué terminación de género admite, y en los morfemas, qué tipo de terminación es. Para que se puedan concatenar, la raíz y el morfema deben tener el mismo valor de su rasgo *get*. Como en la tabla 2.4 que la raíz *president* se puede concatenar con la -*e* y con la -*a* para formar *presidente* y *presidenta* respectivamente, pero no con -*o* porque *get* no incluye el valor *mas1*.

Número

La información gramatical sobre el número está contenida en el rasgo **num**, que puede valer *sing* para singular o bien *plu* para el plural. Al igual que con el género, el número puede estar especificado en un morfema de número o puede estar incluído en la información de la entrada en la base léxica.

Como no existe un morfema de singular `num = sing` en el modelo se ha optado por incluir la información de número singular en el morfema flexivo de género. Así cuando se concatenan *niñ-o* o *niñ-a* ambas palabras heredan el

```
president
concat = nl
get = mas2 fem

e
concat = ng
gen = masc
get = mas2
```

Tabla 2.4: Ejemplo de *presidente*.

número singular del morfema *-o* y del *-a* respectivamente. El número singular también puede aparecer en determinadas palabras, como por ejemplo las de género inherente (*mano, sol, reloj*) que sólo lo perderán si se concatenan con un morfema de plural. También en los *singularia tantum*, aunque en este caso no se permite la concatenación de morfemas de plural.

La información de `num = plu` se transmite por el morfema de plural, que puede ser *-s* o *-es*. Cuando una raíz nominal se concatena con alguno de estos morfemas, la palabra hereda el número plural. También el número plural puede ya venir asociado a determinadas palabras (`concat = w`) como son los *pluralia tantum*.

Las palabras para las cuales no se distingue el plural del singular, y cuyo número se determina por concordancia se dejan sin rasgo *num*, que se determinará en el análisis sintáctico y no se les puede concatenar ningún morfema de número. Por ejemplo *crisis, lunes, virus*, etc.

Hay dos alomorfos del morfema de plural. Para distinguir si a una raíz se le puede concatenar morfema de plural, y en caso positivo determinar con qué alomorfo, se utiliza el rasgo **nut** (tipo de número), que puede valer:

plu1: plurales en *-s*.

plu2: plurales en *-es*.

no: para raíces que no admitan morfema de plural.

Este rasgo se incorpora tanto a las raíces como al morfema. En las raíces indica qué terminación de plural admiten, y en los morfemas qué terminación

```
mesa
concat = wl
gen = fem
nut = plu1

s
concat = ng
nut = plu1

es
concat = ng
nut = plu2
```

Tabla 2.5: Ejemplo de *mesa*.

es. Para que se puedan concatenar la raíz y el morfema tendrán que coincidir en su valor de *nut*. Como ejemplo, en la tabla 2.5 se muestra como *mesa* admite como morfema de plural la *-s* pero no *-es* porque su rasgo *nut* no vale *plu2*.

Irregularidades

La mayoría de las irregularidades de la raíz en flexión nominal se deben a cambios ortográficos, como la pérdida del acento gráfico (*camión, camion-es*) o el cambio de *z* por *c* (*pez, pec-es*).

En el caso típico de *pez, pec-es* se utilizan dos entradas en el diccionario como muestra la tabla 2.6. Una para `pez` como palabra masculina y singular, y otra para `pec` como raíz de género inherente masculino a la que sólo se le puede concatenar morfema de plural.

En el caso típico pérdida de acento, como en *león, leon-a, leon-a-s* y *leon-es*, se emplean dos entradas en el diccionario, según aparecen en la tabla 2.6. Una para *león* como palabra masculina y singular, y otra para *leon* como raíz a la que se le puede incorporar el morfema de número *-es*, o el morfema de género *-a* y entonces también la *-s*. Para el caso de *leon-es* se le incorpora `gen = masc`, pero este se anula si se concatena la *-a* porque hereda `gen =`

```
pez
concat = w
gen = masc
num = sing

pec
concat = nl
gen = masc
get = no
nut = plu2

león
concat = w
gen = masc
num = sing

leon
concat = nl
gen = masc
nut = plu2
get = fem
```

Tabla 2.6: Irregularidades en la flexión nominal.

`fem` de la terminación.

2.3 FORMALISMO LÉXICO

Como anticipamos anteriormente el *formalismo léxico* es el encargado de especificar el formato con el cual expresar los items o entradas y la información lingüística asociada que queremos almacenar, en un fichero de caracteres ASCII llamado *base léxica fuente*. El formalismo que se ha seguido es el que se describe con detalle en [Goñ et.al92] y en [GoñGon95].

El contenido de la base léxica es el conjunto de las entradas, cada una

de ellas con sus rasgos. Por lo tanto el formalismo permite enunciar que una determinada cadena de caracteres es una entrada y permite asociarle un conjunto de rasgos. Los rasgos se expresan en una línea, primero el nombre del rasgo, después un signo = y seguidamente el nombre del valor o los valores que toma ese rasgo. Los rasgos ligados a un item aparecen en líneas inmediatamente siguientes a la línea en la que se declara el nombre de la entrada.

Así por ejemplo para decir que *casa* es un sustantivo femenino, en el fichero aparecería la secuencia:

`casa`	*(nombre del item)*
`cat = n`	*(categoría nombre)*
`gen = fem`	*(género femenino)*

Desde el punto de vista lógico se puede ver al fichero base léxica fuente como una larga lista con declaraciones de este estilo: el nombre del item y detrás sus rasgos asociados. Así hasta que se declaren todas las entradas que queremos almacenar. Sin embargo en la práctica se han explotado diversas regularidades del léxico e incorporado otras facilidades para hacer más cómoda la generación de esta base léxica fuente, por lo que el aspecto del archivo es algo distinto a una simple lista, y aunque obviamente contiene la misma información, ésta aparece más compacta.

El formalismo divide al fichero de la base léxica fuente en apartados o secciones. Y permite la inclusión de otros ficheros con el comando `#INCLUDE <nombre.fichero>` de modo análogo a la inclusión que se permite en los programas en lenguaje C. La tabla 2.7 muestra el aspecto que puede tomar una base léxica fuente, donde cada sección empieza por su nombre, como `#LEXEMES, #DATA-DICT, #CLASSES`, etc y cada una de ellas incluye a los ficheros que realmente contienen los datos de esa sección.

La línea que empieza por el carácter % no se interpreta, por lo que se utiliza para poner en ella todo tipo de comentarios.

Según que las entradas sean raíces, morfemas flexivos o formas lexicalizadas se declaran en la sección `#LEXEMES, #MORPHEMES` o `#WORDS`, respectivamente.

```
% DECLARACION DE RASGOS Y VALORES
#DATA-DICT
#INCLUDE ddict.db

% DEFINICION DE CLASES
#CLASSES
#INCLUDE v-macros.db
#INCLUDE n-macros.db

...

% ENTRADAS LEXICAS (heredan el nombre de la entrada como rasgo lex)
#LEXEMES
#INCLUDE v-lex.db
#INCLUDE v-unip.db
...
#INCLUDE extr.db
#INCLUDE ABC.db

% FORMAS LEXICALIZADAS
#WORDS
#INCLUDE v-words.db
...
#INCLUDE abrevs.db

% MORFEMAS Y DESINENCIAS
#MORPHEMES
#INCLUDE v-morph.db
...
#INCLUDE clit-morph.db

% FIN DE LA BASE LEXICA
```

Tabla 2.7: Fichero base léxica fuente.

2.3.1 Caracteres permitidos y especiales.

Los caracteres permitidos para los nombres y valores de los rasgos son los alfabéticos y numéricos (`A .. Z, a .. z, 0 ..9`) y caracteres especiales (`- + _`). Los caracteres permitidos son todos caracteres ASCII representables en 7 bits.

Para utilizar caracteres de 8 bits, como por ejemplo la `ñ, á, é, í, ó, ú, ü` y las mayúsculas correspondientes, se emplean secuencias especiales para codificarlos. Así la ñ aparece en la base léxica fuente como dos caracteres: ' seguido de n. Es decir la entrada `niño` aparecerá en la base fuente como `ni'no`. Los acentos se expresan del mismo modo, precediendo el carácter ' a la vocal correspondiente. La diéresis se expresa con el carácter : precediendo a la u.

2.3.2 Diccionario de datos.

La sección `#DATA-DICT` de la base léxica fuente es la del **diccionario de datos** y en ella se declaran los nombres de los rasgos y los nombres de sus valores permitidos. Todos ellos se van a usar en el resto del fichero. Al declararse los valores posibles para cada valor se puede aprovechar para verificar la consistencia de la base léxica, y que ningún rasgo se utilice con un valor fuera de su dominio, esto es, no permitido.

La tabla 2.8 muestra una porción del diccionario de datos. El primer nombre es el del rasgo y tras el = aparecen los nombres de sus posibles valores. Hay rasgos cuyos valores quedan *abiertos* en el diccionario de datos porque pueden abarcar un conjunto infinito o muy grande de valores. A estos rasgos no se les impone ninguna restricción en cuanto a los valores que pueden tomar. Es el caso de *lex* en la tabla 2.8.

Los valores que aparecen entre paréntesis precedidos de un @ indican que ellos a su vez son rasgos. Así por ejemplo el rasgo *agr* puede tomar como valor los otros rasgos *gen, num* y *pers*. Se dice entonces que *agr* es un *rasgo complejo*.

2.3.3 Tipos de rasgos.

El formalismo define que el rasgo puede ser de varios tipos según sea su valor:

```
lex =
agr = @(gen num pers)
gen = masc fem
num = sing plu
pers = 1 2 3
vinfo = @(tense mood polite exist clit)
mood = ind subj cond imper inf ger part
tense = pres impf indf fut
polite = +
exist = +
clit = +
...
cat = n v a pn av at prep conj contr ij abrev sc
pronom = +
type = def indef npro pers pos dem indef num rel int c s
polite = + -
refl = + -
def = + -
subtype = cop disy dist advers sus adv ord card frac
```

Tabla 2.8: Parte del diccionario de datos

1. *Simple.* Cuando el valor del rasgo es un valor único y declarado en el diccionario de datos dentro del dominio de tal rasgo. Por ejemplo, para la base léxica que hemos empleado, "`gen = masc`'. Son muy frecuentes.

2. *Disyunción.* Cuando el valor del rasgo es un conjunto de valores atómicos, todos ellos declarados previamente en el diccionario de datos en el dominio de tal rasgo. Por ejemplo, para la base léxica con que hemos trabajado, los rasgos que almacenan el conjunto de posibles terminaciones que admite un lexema verbal: "`stt = 34 35 36 68 89`'.

3. *Literal.* Cuando el valor del rasgo es una única cadena de *caracteres permitidos*. En la base léxica que hemos empleado se utiliza por ejemplo cuando se quiere guardar la forma principal de una entrada: para "`tuv`', "`lex = tener`'.

4. *Cadena.* Cuando el valor del rasgo es una cadena de cualesquiera caracteres. Se distinguen de los literales en que se declaran entre un par de dobles comillas y admiten cualquier carácter en su interior, salvo las propias comillas o dos retornos de carro consecutivos. Por lo tanto admite espacios en blanco y retornos de carro simples. Se utiliza por ejemplo para incluir comentarios: `comment = ``Esta entrada se utilizaba ... '''.

5. *Rasgo complejo.* Cuando el valor del rasgo es otro conjunto de rasgos. Se utiliza para agrupar información relacionada. Por ejemplo en la base léxica usada toda la información de concordancia , como son los rasgos "`gen`', "`num`' y "`pers`' se agrupan como valor del rasgo complejo "`agr`' (*agreement*, concordancia).

2.3.4 Macros o clases.

Se permite la declaración de **macros** o **clases** para conjuntos de rasgos que se repitan con frecuencia. De esta manera se declaran sólamente una vez y luego basta invocarlos por su nombre (el nombre de la macro) al lado de una entrada para que ésta herede los rasgos de la macro. Permite incluso la herencia de varias macros en una entrada, por lo cual establece un orden de herencia (más prioridad la invocada más a la izquierda) para resolver posibles

N cat = n	*Macro de nombre N, se empleará para que las entradas que sean sustantivos hereden el rasgo* `cat = n`
NINO alo 1 stem = $rn alo 1 concat = nl alo 1 get = mas1 fem alo 1 nut = plu1	*Macro de nombre NINO, se empleará para que las entradas que se flexionan como niño hereden estos rasgos que le permiten concatenarse con las terminaciones acecuadas*

Tabla 2.9: Declaración de macros.

niño (NINO N) médico (NINO N)	*Los lemas niño y médico heredarán todos los rasgos que están ligados a las macro NINO y N.*

Tabla 2.10: Invocación de macros.

colisiones o inconsistencias entre clases distintas. Los rasgos que se declaran explícitamente tienen preferencia sobre los heredados.

Las macros se declaran en la sección `#CLASSES` de la base léxica fuente, pero se llaman cuando se definen las entradas, para que el item en cuestión incorpore sus rasgos. Por ejemplo se declaran como el la tabla 2.9. Y se invocan por nombre entre paréntesis al lado del nombre de la entrada, como ilustra la la tabla 2.10

2.3.5 Lemas.

Para mayor comodidad el formalismo permite declarar conjuntamente la información de las distintas variantes alomórficas de un morfema. Así bajo el mismo **lema** o cadena de caracteres se declaran todas sus variantes alomórficas con sus rasgos asociados. El haz de rasgos del lema incluye tanto los rasgos comunes a todos los alomorfos como los exclusivos de cada variante. De este modo la información común no se repite y se tiene agrupados los distintos alomorfos en un lema de la base léxica fuente. Por ejemplo los rasgos exclusivos del alomorfo 3 de un lema se incorporan en la rama `alo 3` del haz de rasgos del lema. Mientras que los rasgos comunes se declaran en el haz del lema como los que no cuelgan de ninguna rama `alo` ni `aux`.

```
rv0
{X = .*}
$X'ir     -> $X          % La "i" puede estar acentuada
$X[aei]r  -> $X          % Pierde la terminación
```

Tabla 2.11: Declaración de reglas de alomorfía.

Cada lema dará lugar a tantas entradas del diccionario objeto como alomorfos se hayan declarado en él. Cada uno de ellos tendrá su propio nombre y un conjunto de rasgos con las partes declaradas comunes en el lema, además de los rasgos genuinos que estaban declarados como exclusivamente suyos en el lema.

El conjunto de rasgos de cada alomorfo se obtiene aplicando al haz de los rasgos del lema unas **reglas de filtrado**. Estas reglas se declaran en una sección de la base léxica fuente. Básicamente definen operaciones de manipulación de conjuntos de rasgos, permitiendo añadir los rasgos genuinos y eliminar los que no corresponden al alomorfo concreto.

Cada alomorfo se obtiene con la aplicación de una **regla de alomorfía** a la cadena del lema. Estas reglas se declaran también en otro apartado de la base léxica fuente. Son reglas de manipulación de caracteres que generan la cadena del alomorfo desde la cadena del lema.

2.3.6 Reglas de alomorfía

Estas se declaran en la sección `#ALO-RULES` y tienen un aspecto como el de la tabla 2.11.

La primera línea en la tabla 2.11 es el nombre de la regla, por el cual se invocará. Entre llaves aparecen asignaciones de variables que se realizan antes de aplicar la regla. En este caso la variable X puede valer cualquier combinación de caracteres. En cada línea con un `->` hay una *regla de producción*. Tanto para la asignación de variables como para las reglas de producción se utilizan expresiones regulares. Cuando la cadena a la que se quiere aplicar la regla casa con una expresión regular a la izquierda del `->` entonces la cadena de salida se forma como indica la parte derecha del `->`. En este caso la regla provoca que las cadenas de entrada acabadas en `ar`, `er`, e `ir` pierdan esa

```
amar
cat = v
concat = vl
conj = 1
alo 1 stem = $rv0
```

Tabla 2.12: Invocación de las reglas de alomorfía.

```
LEXEMES
$$ = @ alo 1 stem
@ = @ (- alo - aux)
@ = @ alo 1 (- stem)
@ lex = $$
```

Tabla 2.13: Declaración reglas de filtrado.

terminación en la cadena de salida. Se aplica por ejemplo para obtener las raices verbales de los lemas que coinciden con el infinitivo.

Las reglas de alomorfía se invocan en la posición de un valor de un rasgo, por el nombre de la regla precedido de un signo $. Y se aplican sobre el nombre del lema. Por ejemplo en la tabla 2.12 se muestra como para el lema `amar` el valor de su rasgo `alo 1 stem` es `am` porque es la cadena que resulta de aplicar la regla `rv0` (que se muestra en la tabla 2.11) a la cadena del lema, `amar`.

2.3.7 Reglas de filtrado

Estas se declaran en la sección `#DICT-RULES` separadas por líneas en blanco y tienen un aspecto como el de la tabla 2.13.

Como indicamos anteriormente estas reglas tienen como entrada el haz de rasgos asociado a un lema y como salida el haz de rasgos de un alomorfo concreto. El signo `$$` se refiere a la cadena asociada al haz de rasgos, si aparece a la izquierda del = se refiere nombre del alomorfo, y si a la derecha al nombre del lema. El signo @ se refiere al propio haz de rasgos, si está a la

izquierda del =, al haz del alomorfo y si a la derecha, al haz del lema.

Veamos ahora lo que significan cada una de las líneas del ejemplo de la tabla 2.13:

`$$ = @ alo 1 stem`	*La entrada o cadena a la que se asocia el haz del alomorfo ($$), es el valor del rasgo* `alo 1 stem` *del haz del lema (@)*
`@ = @ (- alo - aux)`	*Copia en el haz del alomorfo (primer @) el haz del lema (segundo @) excepto las ramas alo* y *aux. Estos serán los rasgos comunes a todas los alomorfos del lema.*
`@ = @ alo 1 (- stem)`	*Copia en el haz del alomorfo (primer @) la rama* `alo 1` *del haz del lema, excepto el rasgo* `stem`. *Estos serán los rasgos exlusivos al alomorfo 1 del lema.*
`@ lex = $$`	*Copia como valor del rasgo* `lex` *en el haz de rasgos del alomorfo, el nombre del lema.*

En el ejemplo de la tabla 2.13, con `LEXEMES` indicamos que las dos reglas de filtrado de la tabla sólo se aplican en la sección `#LEXEMES` de la base léxica. Gracias a poder especificar en que sección se aplican estas reglas de filtrado éstas se pueden aprovechar para asociar a las entradas determinada **información implícita** por el hecho de residir en una sección. Los items se agrupan en secciones como `#LEXEMES` , `#MORPHEMES` o `#WORDS` y las entradas en cada una de ellas hereda determinados rasgos que se declaran en las reglas de filtrado asociados a la sección. Por ejemplo:

`#WORDS`	*Las entradas declaradas en la sección #WORDS*
`@ = @ (- aux)`	*heredan el rasgo* `concat = w` *para indicar que son*
`$$ = $$`	*palabras por sí mismas.*
`@ concat = w`	

2.4 FORMATOS DE LA BASE LÉXICA

EL objetivo del diccionario es almacenar gran cantidad de información lingüística y proporcionársela a los programas de aplicación cuando estos le consulten. Ya hemos visto que toda esa información reside en el fichero de caracteres

base léxica fuente. Sin embargo acceder a los datos lingüísticos directamente sobre ese fichero sería lentísimo. Para poder responder más eficientemente la base léxica cambia de formato hasta llegar al de **diccionario objeto** que es la apariencia bajo la cual los programas de aplicación tienen visible esa información de un modo rápido y eficiente. Sobre ese diccionario objeto se ha desarrollado una interfaz de consulta, que es la que los programas usuarios de la base léxica utilizan en realidad cuando quieren acceder a la información que ella contiene.

Por lo tanto, para que las aplicaciones puedan consultar la base léxica se debe construir antes el correspondiente diccionario objeto. El proceso de generación de un diccionario objeto se puede ver como un cambio de formato que se aborda en varios pasos.

El formato de la base léxica fuente viene especificado por el formalismo léxico de [Goñ et.al92]. Este formalismo principalmente está pensado para aprovechar al máximo las regularidades y evitar repeticiones de información común. Por ello permite el uso de macros, la información por defecto y la declaración conjunta de los distintos alomorfos de una entrada.

2.4.1 Formato expandido

Un primer paso de transformación es la **EXPANSIÓN** de la base léxica fuente. Consiste en explicitar toda la información que aparece compactada en la base fuente. Al realizar la *expansión* de la base fuente se genera la **base léxica expandida**. Esta base expandida es también un fichero de caracteres ascii en el que aparecen todas las entradas individuales del diccionario y cada una con todos sus rasgos explícitos. Su formato es muy parecido a como se declaran las entradas sin variantes alomórficas en la base fuente: una línea con el nombre del item y en las siguientes líneas los rasgos asociados, uno por línea. La entradas se separan unas de otras por líneas en blanco. La información por defecto de la base fuente se explicita y se resuelve la invocación de macros para que todos los rasgos de la clase aparezcan realmente en la base expandida en el item en que se llamó a la macro.

Una vez explicitadas las macros, las entradas con los lemas en la base fuente dan lugar a tantas entradas como alomorfos tenga, cada una de ellas con sus rasgos asociados. Cada alomorfo será una entrada individual en el diccionario. En este proceso se aplican las *reglas de alomorfía* para generar los nombres de los alomorfos, y las *reglas de filtrado* para generar los haces

```
acertar
cat = v
conj = 1
concat = vl
alo 1 stem = $rv0
alo 1 stt = 0 14 15 ...  46 54 ...
alo 1 sut = reg
alo 2 stem = $rv1-2
alo 2 stt = 11 12 13 16 51 52 53 56 82 83 86
alo 2 sut = reg
```

Tabla 2.14: Ejemplo de lema en la base fuente

de rasgos de los alomorfos desde los haces de rasgos del lema e incorporar la información implícita asociada a la sección en que se declaró. Por ejemplo agrupados en la base fuente bajo el lema `acertar`, tal como aparece en la tabla 2.14, hay declarados dos alomorfos, que emergen en la base expandida, como `acert` y `aciert`, según ilustra la tabla 2.15 ya en formato expandido. Al aplicar las reglas de alomorfía `rv0` y `rv1-2` al lema `acertar` se obtienen los nombres de los rasgos `acert` y `aciert` respectivamente.

El **expansor** será el código que transformará una base fuente en base expandida. Su entrada es un fichero de caracteres escrito según especifica el formalismo de [Goñ et.al92] y su salida otro fichero de caracteres, en formato expandido.

2.4.2 Formato objeto o compilado

El siguiente paso de transformación es la **COMPILACIÓN** de la base léxica expandida. Al compilar se genera la **base léxica compilada**, también llamada **diccionario objeto**, que se compone de varios ficheros en disco. La *compilación* consiste en pasar las entradas con sus rasgos que aparecen explícitos en la base expandida a unas determinadas estructuras abstractas de datos. Éstas se han elegido por su eficiencia en tiempo y en espacio; a describirlas y justificarlas dedicaremos el capítulo 3 de esta memoria. Estas estructuras se vuelcan en unos ficheros, donde residen entre sesión y sesión

```
acert
lex = acertar
cat = v
conj = 1
concat = vl
stt = 0 14 15 ...  46 54 ...
sut = reg

aciert
lex = acertar
cat = v
conj = 1
concat = vl
stt = 11 12 13 16 51 52 53 56 82 83 86
sut = reg
```

Tabla 2.15: Ejemplo de alomorfos en la base expandida

de uso de la base léxica. Así no hay que generar las estructuras cada vez que se va a utilizar la base léxica, sólamente *cargarlas* desde esos ficheros. Esos archivos no serán simples concatenaciones de caracteres, sino un reflejo binario de las estructuras de manera que **cargar** la estructura no requerirá procesamiento adicional, sino que simplemente consistirá en volcar ese fichero a memoria.

Una vez cargadas, estas estructuras son la **base léxica en tiempo de ejecución**. Son el formato que tiene la información de la base léxica cuando es realmente consultada por los programas de aplicación.

Poder almacenar estas estructuras en archivos es una gran ventaja porque construirlas es una operación muy costosa en tiempo, mientras que cargarlas es algo mucho más rápido. Por lo tanto cuando se quiera aprovechar una base léxica se genera la base compilada *una vez* y luego solo se cargan los ficheros con el diccionario objeto cada vez que se vaya a utilizar la base léxica. Este proceso es análogo al que se sigue con los programas software en cualquier lenguaje de alto nivel: al compilarse el código fuente se genera el código objeto, que es el que se carga para ejecutarse. No se compila cada vez

que quiero ejecutar, sólo se carga el código objeto, ya compilado.

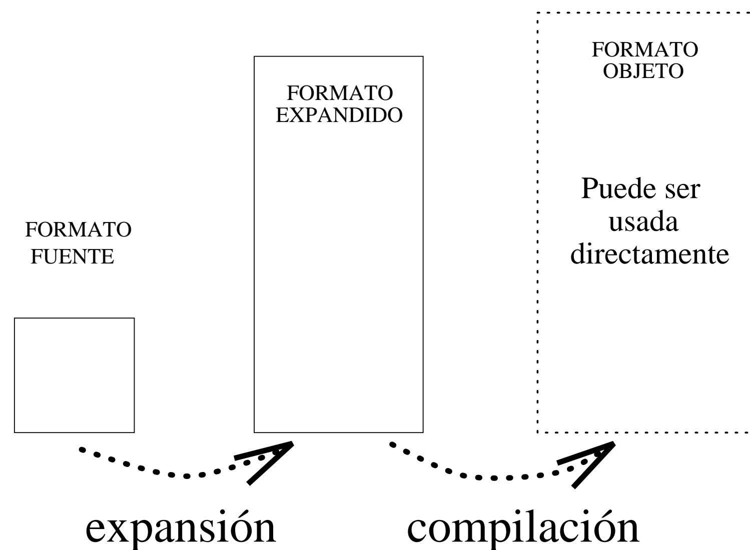

Figura 2.1: Distintos formatos de la base léxica.

2.4.3 El compilador.

El **compilador** es el código que transforma una base expandida en diccionario objeto. Por un lado entiende el formato expandido de una entrada seguida de lineas con sus rasgos asociados, y por otro entiende de crear estructuras de la base compilada.

El compilador que hemos desarrollado es un compilador *ad hoc* que aprovecha la sencillez del formato expandido. Lee secuencialmente el fichero de entrada ya que siempre le basta ver el caracter siguiente para saber como interpretar el que acaba de leer. Es un procesador recursivo desdendente. Entiende la base expandida como una lista de entradas. Las entradas se separan entre sí por una o más líneas en blanco. Cada entrada es un nombre en una línea y en las siguientes sus rasgos. Un rasgo se define en una línea con un signo =. Antes del = está el nombre del rasgo y después del = su valor.

En la elaboración de este compilador se ha preferido la sencillez y la eficiencia frente a la robustez. Es decir, no se ha complicado el código contem-

plando la recuperación frente a todo tipo de errores de formato en el fichero expandido. El compilador supone que ese fichero está correctamente escrito. Los esfuerzos de corrección de formato se centran en el expansor, que es la herramienta del sistema diccionario que genera las bases expandidas.

Las estructuras las crea y manipula a través de la interfaz que éstas ofrecen. A medida que lee un rasgo o una entrada del fichero de entrada el compilador crea su homólogo en las estructuras. Según lee los rasgos de una entrada va creando y enlazando las estructuras rasgo. Una vez acabada de leer la entrada inserta su estructura homóloga en la estructura del conjunto de entradas.

En pseudocódigo la estructura del compilador podría ser:

```
[compilador]:
   do{ lee_entrada }
   while (no EOF)

[lee_entrada]:
   lee_nombre_de_entrada
   do{ lee_rasgo }
   while (no lnea vaca)

[lee_rasgo]:
   lee_nombre_rasgo;
   lee_el_=;
   lee_valor_del_rasgo;
```

Capítulo 3

ESTRUCTURAS DEL DICCIONARIO

3.1 INTRODUCCIÓN.

El diccionario objeto se ha dividido en dos partes fundamentales: por un lado el conjunto de los significados, y por otro unas estructuras que almacenan todos los nombres de las cadenas. Estas últimas permiten asociar un significado a cada nombre de entrada. Todos los significados que contiene el diccionario se han agrupado en un conjunto, que llamaremos **datafile**. Así se llamara también el fichero en el cual residen estas estructuras cuando no se está usando el diccionario. Además se han desarrollado funciones y módulos que permiten manipular esos significados. Este código manipulador es el que accede a las estructuras de datos concretas y permite al usuario manejar los significados de forma opaca a su implementación.

La asociación entre una entrada, que se presenta como una cadena de caracteres y su significado se consigue con una estructura llamada **trie**. Según [Aho et.al83] el nombre de esta estructura proviene de las letras centrales de la palabra inglesa `reTRIEval`, y como no hemos encontrado una traducción castellana adecuada, mantendremos el término inglés original. Al trie se le pregunta por una cadena de caracteres y si la entrada está contenida en el diccionario devuelve un índice que apunta a la zona del datafile donde está el significado de esa entrada.

Como ilustra la figura 3.1 el acceso a la información asociada a una entra-

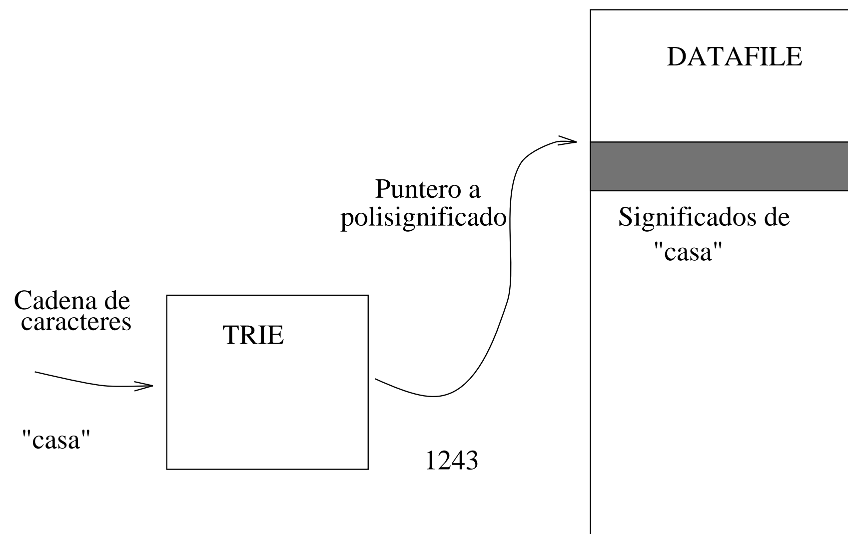

Figura 3.1: Las dos partes del diccionario: el trie y el conjunto de haces de rasgos.

da se consigue con dos pasos: primero se busca la entrada en el trie, y luego con el índice que éste nos devuelve podemos manejar el significado, a través del código manipulador, según requiera la aplicación, según la información que busquemos. La eficiencia del diccionario depende de la de cada una de estas dos partes en que lo hemos dividido: por un lado cuan rápido encuentre en el trie el significado asociado y por otro cuan veloz sea el código que me permite manipular y consultar esos significados. Estos tiempos son los que se ha buscado minimizar a la hora de elegir la organización y estructuras reales que soportan tanto al conjunto de entradas como al conjunto de sus significados.

3.2 HACES DE RASGOS.

Ya hemos comentado que la toda la información reside en la base léxica fuente y que basicamente lo que el diccionario hace es cambiarla de forma para que se pueda almacenar, recuperar y manejar de un modo eficiente tanto en tiempo como en espacio. Bajo este epígrafe vamos a explicar cuales son las estructuras que dan forma final a los significados de las entradas y sobre las

que realmente se accede. Es decir se van a describir y justificar el formato compilado que se ha escogido para los significados.

En el fondo, este formato, estas estructuras finales, son la implementación de todo lo que el formalismo léxico permite y la base léxica concreta explota.

El propio formalismo léxico ya plantea que el diccionario es un conjunto de entradas con su información asociada y que nosotros llamaremos significado. También especifica que cada significado es un conjunto de rasgos.

Cada significado expresa una posibilidad morfológica, sintáctica y/o semántica de su entrada, pero esto es irrelevante para el diccionario , eso sólo lo saben subsistemas superiores en el procesamiento del lenguaje. Estrictamente hablando el diccionario sólo entiende que un conjunto de bytes está asociado a una entrada, a una en concreto y no a otra.

3.2.1 Diccionario de datos.

El formalismo léxico obliga a que el propio creador de la base léxica declare los nombres de los rasgos que va a utilizar y además defina sus dominios, es decir los posibles valores que cada rasgo puede tomar. Hay una parte de la base léxica que se reserva para ello, es el llamado **diccionario de datos**.

Esta declaración previa permite que el diccionario no maneje internamente las cadenas de caracteres con los nombres de los rasgos y sus valores, sino que en su lugar trabaje con códigos numéricos equivalentes. A cada nombre de rasgo o de posibles valores se le asocia un código numérico único y el diccionario trabaja internamente con él. Cara a los usuarios el diccionario sigue hablando de "gen' y "num' y no de código 35 o 67, porque los usuarios no saben qué significan esos números. Por ejemplo los programas usuarios puede que le pregunten al diccionario por el rasgo "gen' (género) de la entrada "casita' y este responderá que es "fem' (femenino). Sin embargo no ha trabajado con las cadenas de caractes sino que primero traduce "gen' al código 35 y por él busca en el significado de "casita'. Encuentra que el valor del rasgo cuyo código es 35 es el código 28, que corresponde a "fem'. Para responder al usuario traduce 28 y le entrega "fem'.

Trabajar con códigos pretende acelerar el funcionamiento del diccionario, porque en general el procesador compara más rápidamente dos números que dos cadenas de caracteres. La incidencia de esta mejora es grande porque en el manejo de significados y rasgos se necesitan muchas comparaciones que de otro modo serían más lentas. Además de ser más rápido ocupa menos

Nombre	Código numérico
...	...
abrev	62
acc	63
adv	64
advers	65
agr	66
alo	67
apr	68
...	...
vm	184
w	185
wl	186
yes	187

Tabla 3.1: Parte de la tabla de códigos.

espacio, porque en general también un número ocupa menos memoria que un nombre.

No hay ningún inconveniente en utilizar números como nombres o valores de rasgos, porque se mantienen separados los dos espacios el de nombres y el de códigos. Por ejemplo el valor "38' no es lo mismo que el código 38.

Puesto que sólo se manejan números internamente es necesaria una estructura que traduzca de cadenas de caracteres a su código correspondiente y viceversa. Además es imprescindible que sea lo más veloz posible, puesto que se usará con mucha frecuencia. La estructura que se ha creado es un *array modificado*.

Para averiguar qué cadena le corresponde a un código simplemente se utiliza el código como índice dentro del array. Así para el código 15, en la posición 15 del array está el puntero a la cadena correspondiente.

Para la operación inversa se lanza una búsqueda en un árbol binario en el que se han ido insertando nodos con los nombres de los rasgos o valores simples y su código a medida que se leían del diccionario de datos. Se busca por la cadena y cuando se encuentra el nodo en el que reside, en él está también su código asociado. El árbol está equilibrado para minimizar

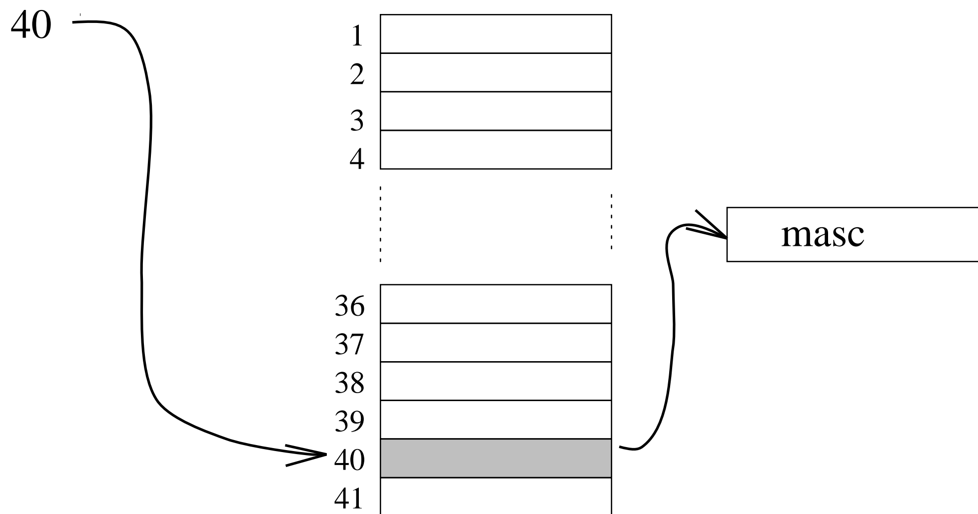

Figura 3.2: Conversión del código 40 a su nombre asociado.

el tiempo de búsqueda.

El array y el árbol binario se han fusionado en una sola estructura que permite ambas búsquedas, y que es el *array modificado*. En la figura 3.4 se observa que este es un array en el cual también se utilizan los códigos como índices, pero con dos campos más aparte del puntero a cadena. Cada posición de este array modificado se puede ver además como un nodo del árbol binario. Entonces esos dos campos adicionales son los punteros a los nodos *izquierda* y *derecha* en el árbol binario, y como los nodos son posiciones, serán dos índices dentro del array modificado. Para convertir de código a cadena utilizo el array modificado como un array normal, indexando con el código. Para traducir de cadena a código se recorre como un árbol binario siguiendo los índices izquierdo o derecho hasta llegar a la posición con la cadena que se busca, cuyo índice será el código. El nodo raíz del árbol se ha situado en la posición inicial del array modificado. A la hora de construir el árbol binario sobre el array modificado se cuida que salga equilibrado, para minimizar el tiempo de acceso. Tanto la estructura como las funciones de traducción necesarias se han reunido en el módulo *conv.cc* (de conversión).

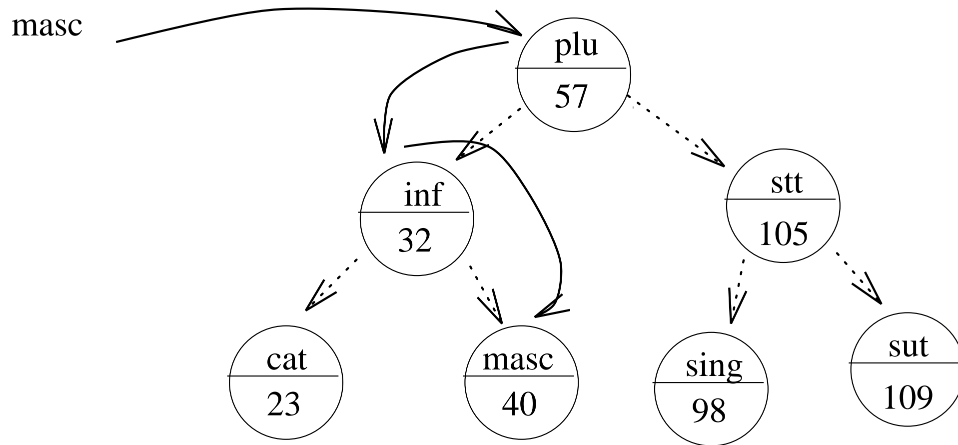

Figura 3.3: Conversión del nombre masc a su código asociado.

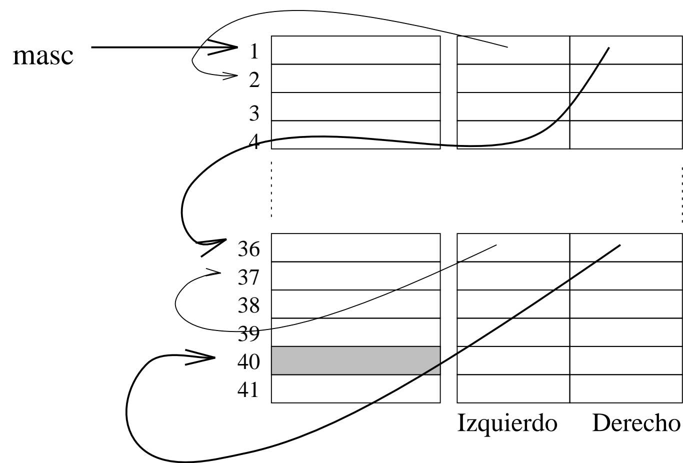

Figura 3.4: Array modificado.

3.2.2 Tipos de rasgos.

Como vimos en el capítulo 2 el formalismo define que el rasgo puede ser de varios tipos *simple, en disyunción, literal, cadena* o *complejo* según sea su valor. A la vista de estos tipos la estructura en que se implementa un rasgo, como muestra la figura 3.5, es un nodo con varios campos:

- El campo del **nombre del rasgo**, que será el nombre del nodo, guarda el código del nombre, no su cadena de caracteres.

- Un campo con el **valor** que toma ese rasgo. Este campo es una unión de lenguaje C cuyo contenido depende de la clase de rasgo que sea. Si es simple tendrá el código del valor; si es en disyunción tendrá un puntero al conjunto de valores atómicos; si es literal o cadena, un puntero a la cadena de caracteres; y si es complejo, un puntero al conjunto de rasgos que constituyen el valor de este rasgo. Hemos escogido una unión para ahorrar espacio porque una unión reserva memoira para el mayor de los tipos que puede contener, que siempre será menor que si se guardara espacio para todos ellos.

- Se necesita un campo que nos indique cómo interpretar el contenido de la únion. Es el campo **clase**. Sin él no sabriamos si leer el valor como un código o como un puntero a un conjunto de rasgos.

- Dos campos **izquierdo** y **derecho**, que aparecen por la necesidad de agrupar varios nodos rasgo en un conjunto. Son punteros a otros nodos.

3.2.3 Árboles binarios y balanceo.

El formalismo especifica que un significado es un conjunto de rasgos. La manera que hemos escogido para implementar esos conjuntos es la de árboles binarios.

La principal ventaja que conlleva esta organización frente a otra como pudiera ser un array o lista encadenada es el tiempo de acceso. Supongamos que en un conjunto tenemos n rasgos. Como hemos visto cada rasgo tiene un nombre. Si se realiza el conjunto como array o como lista, cuando se quiere acceder a un rasgo concreto hay que buscar linealmente entre los del array o lista hasta que el nombre del que ocupa la posición que se está explorando coincida con el nombre del que se busca. Esto lleva a que en media haya que

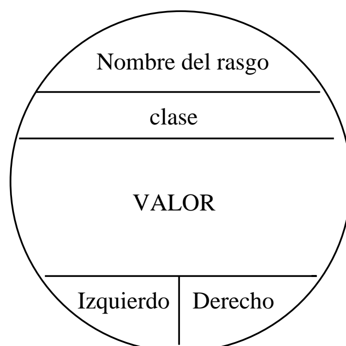

Figura 3.5: Estructura rasgo.

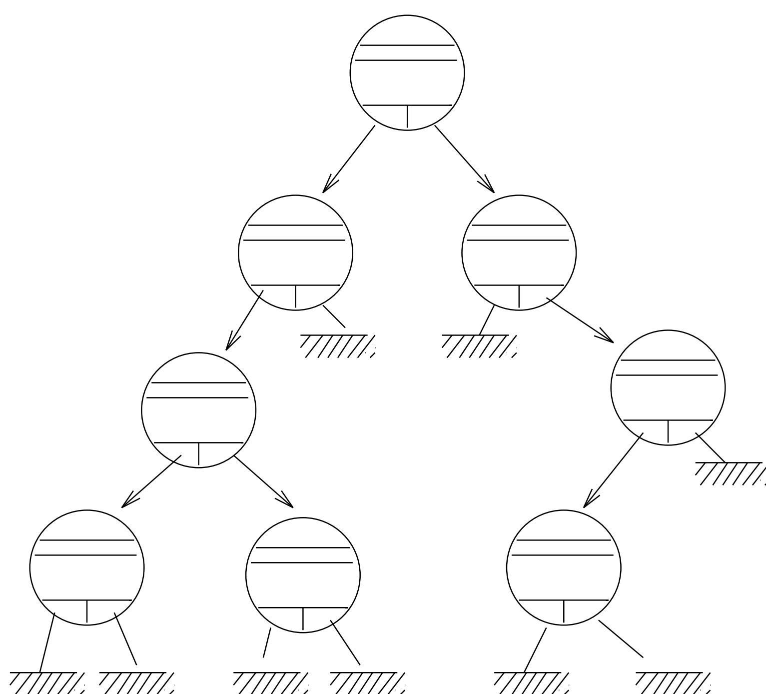

Figura 3.6: Haces de rasgos

consultar $\frac{n}{2}$ rasgos para encontrar el que se necesita. Si se realiza el grupo como árbol binario se consulta al nodo de la raiz y si el que se busca es alfabéticamente mayor se sigue explorando por la rama de la izquierda, y si es menor se sigue por la rama de la derecha. Así recursivamente hasta que se encuentre el rasgo que se busca. Si el árbol está equilibrado este proceder arroja en media $\log n$ accesos antes de encontrar al que se necesita, donde el logaritmo es en base 2 al ser un árbol binario. Ocurre que $\log n < \frac{n}{2}$ por lo cual el acceso en árbol binario es más rápido en media que el acceso en array. Y la diferencia es mayor cuantos más nodos tenga el conjunto.

La *profundidad* de un árbol es el máximo número de saltos que hay que dar para llegar desde el nodo raiz a cualquier nodo del árbol. Se dice de un árbol que está **equilibrado** o **balanceado** cuando es de profundidad mínima. Conviene que los árboles binarios de los haces esten equilibrados, porque si no lo estan tendrán más profundidad y el tiempo medio de acceso a un rasgo suyo será mayor. En el caso extremo un árbol se convierte en lista ligada si inserto los nodos en orden alfabético, y su tiempo de acceso empeora hasta ser como el de un array.

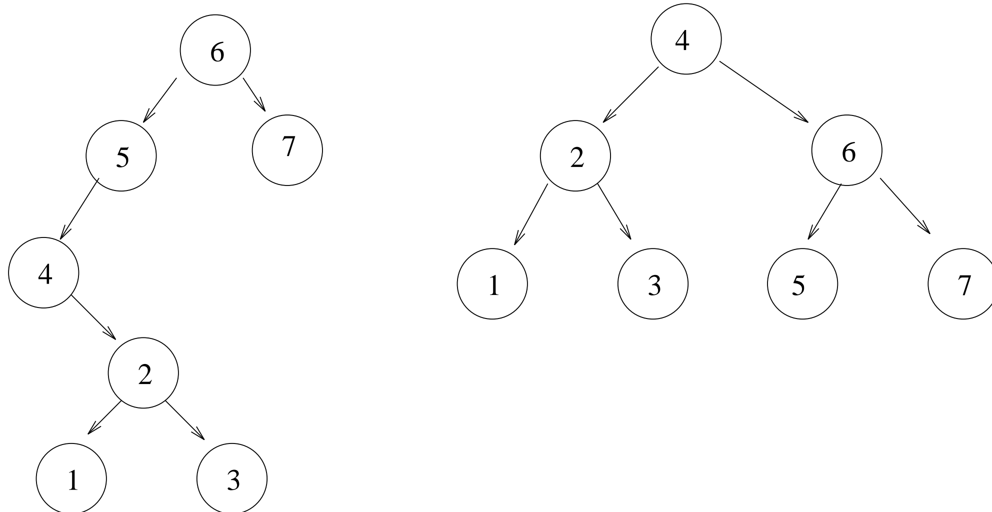

Figura 3.7: Balanceo de árboles.

Los significados de nuestro diccionario son árboles binarios donde cada

nodo es un rasgo; serán por lo tanto *haces de rasgos*. Estos se balancean en la compilación, de modo que la compilación genera los haces de rasgos equilibrados. De esta manera todas las consultas al diccionario manejaran árboles de tiempo acceso mínimo.

El algoritmo que se ha empleado de balanceo trabaja directamente sobre árboles no equilibrados retocando los enlaces entre nodos, de manera que no duplica memoria ni pierde tiempo en ello. Básicamente consiste en leer el árbol entero para conseguir una tabla ordenada con los nombres de los rasgos y los punteros donde estos nodos residen. Se recorre esa tabla en el orden en que una búsqueda binaria lo haría y se retocan los punteros a la vez, lo que acaba construyendo un árbol "nuevo' balanceado.

3.2.4 Polisemia.

Del mismo modo que "gato' puede referirse al animal o a la herramienta es fácil entender que cada entrada en el diccionario puede tener varios "significados' asociados. Por lo tanto la estructura que elijamos debe recoger esta posibilidad. A cada entrada no se le asigna un haz de rasgos, sino varios, uno por cada significado distinto que le pueda corresponder. Ese grupo es el **polisignificado** de la entrada. Este conjunto de haces se ha realizado como una simple lista porque no hay ningún criterio que distinga a un haz de otro salvo su contenido. En otras plabras, los haces no tienen nombres, ni se acceden por ellos. Organizarlos en un árbol binario no hubiera acarreado ninguna ventaja. Asi que decidimos ponerlos uno tras otro, sencillamente.

3.2.5 Interfaces de acceso.

Por encima de todas las estructuras de los haces hemos desarrollado unas funciones de acceso, tanto de lectura como de escritura. Su propósito es ocultar a los programas usuarios la estructura real y que éste sólo las manipule a través de estas funciones. Por ello se han programado funciones que realizan todo lo que se puede necesitar para consultar un polisignificado. No sólo los programas de consulta, también el compilador creará y manipulará las estructuras que genere a través de esta interfaz de llamadas.

Las funciones se han distribuido en tres módulos que captan la jerarquía en las estructuras: uno manipula rasgos (*d_rasgos.cc*), otro haces de rasgos (*d_sig.cc*) y el tercero polisignificados (*d_poli.cc*). Estos tres módulos definen

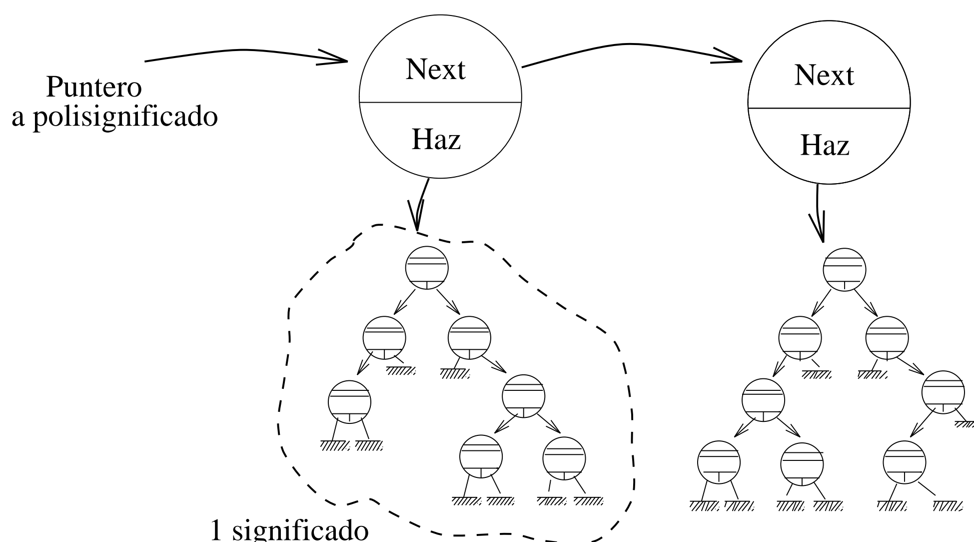

Figura 3.8: Polisignificado como lista de haces.

estructuras y ofrecen funciones básicas de creación, manipulación y consulta de esas estructuras. Esas funciones constituyen la parte "**básica**" de la interfaz, en el sentido que son de bajo nivel, herramientas con las que construir otras funciones de acceso más sofisticadas. Precisamente se ha creado un módulo (*d_coci.cc*) que combinando elementos de esa interfaz básica ofrece funcionalidades de acceso más generales y para las que no es necesario conocimiento alguno de las estructuras. Este módulo presenta funciones sencillas para el usuario, que conforman una interfaz "**elaborada**" del diccionario.

3.2.6 Interfaz de original y de copia.

Las aplicaciones pueden utilizar al diccionario de dos maneras distintas: en *modo consulta del original* y en *modo copia*.

En el modo consulta del original los accesos son sólo lecturas. Por ello el diccionario nunca entrega copia de sus respuestas cuando se le pregunta por los significados de una entrada, sino que devuelve punteros sobre su estructura original. El usuario se limita a no hacer un uso destructivo, porque está accediendo al original, a través de los interfaces (básica o elaborada) que le ofrece el código manipulador de datos. En este modo el diccionario traba-

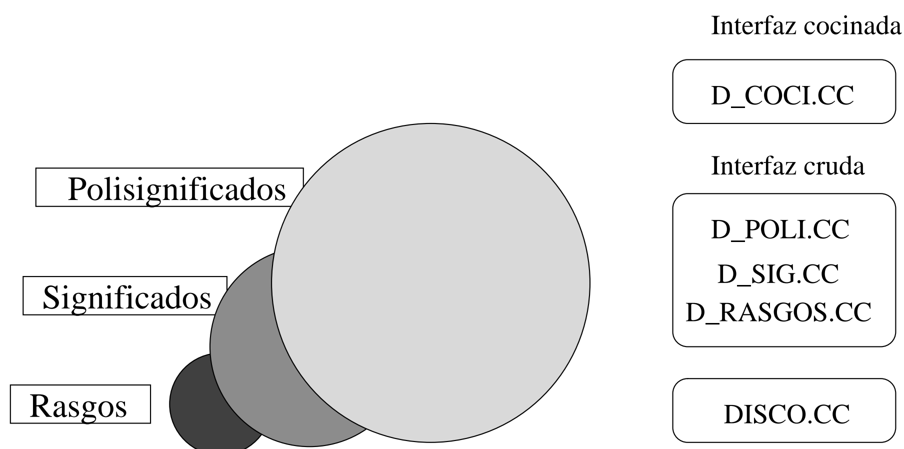

Figura 3.9: Jerarquía de estructuras y los módulos que las manejan.

ja muy velozmente porque no hace copias de sus respuestas y hacer copias lleva tiempo. Este sería el caso por ejemplo de un corrector de textos sencillo que diera por correcta una palabra si se puede descomponer en formantes que existen en el diccionario. A este corrector sólo le interesa saber si una entrada está o no en la base léxica. Para esto no hace falta copiar sus significados.

El otro empleo del diccionario es el empleo destructivo de sus respuestas. En este caso el programa usuario tiene previsto modificar las estructuras con los significados de los items que consulta. Por lo tanto cuando le preguntan por una entrada, el diccionario la busca y si la encuentra hace copia de su estructura de significados. Esa copia es la que entrega al programa usuario que ya podrá alterar esas estructuras sin peligro para la integridad del diccionario. Se han desarrollado funciones distintas para manipular la copia y el original, porque como en el capítulo 4 se verá estas son diferentes en la práctica.

Una aplicación que utiliza destructivamente el diccionario es el analizador morfológico que ha desarrollado Angel L. González en [Gon95]. Este uso del diccionario es más lento porque hacer copia de los haces ralentiza mucho las respuestas. Sin embargo si el programa usuario pretende modificar las estructuras, entonces no hay otro camino que el diccionario entregue copia.

3.3 ESTRUCTURA CON LAS ENTRADAS

Ya hemos visto las estructuras que soportan los significados de las entradas, ahora veremos las que soportan el conjunto de los nombres de las entradas y permiten enlazar esos nombres con sus haces de rasgos, o polisignificado asociado.

Las entradas vienen dadas como cadenas de caracteres y los significados se identifican por la posición que ocupan en el *datafile*. El trie es la estructura que permite asociar una cadena de caracteres con el índice del datafile que apunta a su polisignificado. Esto que parece un problema sencillo no lo es tanto cuando tenemos por ejemplo 40.000 entradas y fuertes exigencias de velocidad en lo que puede tardar en encontrar el índice asociado.

En este epígrafe voy a explicar las distintas alternativas que hemos estudiado, como por ejemplo la tabla hash, el árbol binario, etc y la que finalmente hemos adoptado: el trie en doble array. Analizaremos todas ellas tanto desde su eficiencia espacial como sobre todo, la temporal.

3.3.1 Tabla ordenada y búsqueda binaria.

En principio podríamos almacenar en una tabla de dos columnas cada entrada con su puntero a significado. Para buscar en ella la tendríamos que recorrer linealmente hasque que encontraramos la entrada por la que preguntamos. Esta búsqueda, muy sencilla, nos llevaría $\frac{n}{2}$ comparaciones como media, siendo n el número de entradas que tenemos. Para 40.000 entradas, que es aproximadamente el número de items de la base léxica con la que hemos trabajado, tendríamos que hacer 20.000 comparaciones cada vez que consulto un item. Desde luego resulta inaceptable, pero es una primera manera de situar el problema.

Para agilizar esta búsqueda se podría ordenar la tabla alfabéticamente. Con la tabla ordenada puedo lanzar una búsqueda binaria sobre ella, de modo que en media necesito $\log n$ (logaritmo en base 2) comparaciones para acceder al índice ligado a un item. Para 40.000 entradas, 91 comparaciones. La reducción con respecto a la tabla sin ordenar es impresionante. Sin embargo todavía arroja un tiempo excesivo. En cada posición que se consulta de la tabla se cotejan las cadenas del item y del que ocupa esa posición, lo que se traduce en varias comparaciones de caracteres individuales.

Las prestaciones de esta búsqueda binaria son independientes de que se

organice sobre un array o sobre un árbol ordenado y equilibrado en el cual cada nodo contiene la cadena y su índice asociado del datafile.

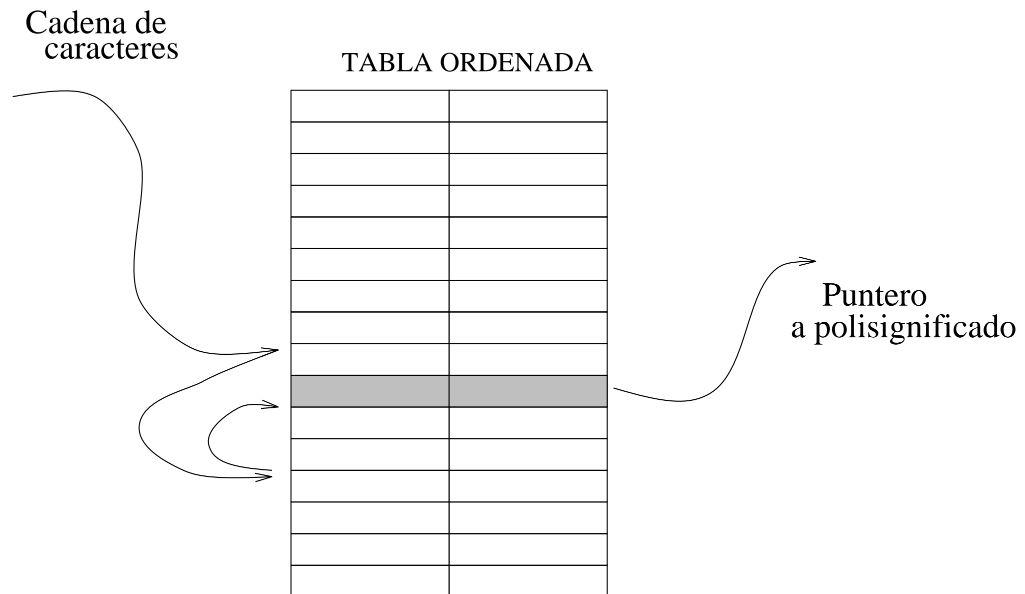

Figura 3.10: Búsqueda binaria en una tabla ordenada..

Como cada entrada puede ser tan larga como quiera en la tabla no hay cadenas directamente sino punteros a ellas. Los punteros sí tienen un tamaño fijo.

Tanto si la tabla está ordenada como si no, el tamaño de esta estructura es proporcional al número de items del diccionario:

$$mem.estruct = mem.tabla + mem.cadenas \quad (3.1)$$

$$mem.tabla = n(tam.puntero.cadena + tam.indice.datafile) \quad (3.2)$$

$$mem.cadenas \approx n(long.media.cadena) \quad (3.3)$$

$$mem.estruct \approx n(long.med.cadena + tam.puntero + tam.indice) \quad (3.4)$$

Para un caso real el tamaño total en bytes podría ser $40000(5+4+4) =$ 520.000 bytes, que es soportable. Sin embargo su lentitud descarta completamente a estas opciones.

3.3.2 Entradas como índices, el hash.

Si tuvieramos una memoria infinita podríamos construir un array muy grande y utilizar la cadena de caracteres directamente como índice dentro de ese array en el que estarían los índices de polisignificado. Cada cadena se puede interpretar como un número único, distinto de los números del resto de las cadenas. En la posición del array que señala ese número insertaríamos el puntero a polisignificado asociado. Como la correspondencia entre número y cadena es biunívoca, a ese puntero solo puede llegar su item asociado.

Esta es la manera más rápida de llegar a un puntero a polisignificado desde el nombre del item, pero es imposible de realizar en la práctica por su gasto en espacio. Como cada cadena va unívocamente a una posición dentro del array, este debe tener tantas posiciones como posibles cadenas. No todas las cadenas que aparecen en el diccionario, sino todas las posibles cadenas que puedo formar con el juego de caracteres que tenga. Si por ejemplo el máximo número de caracteres de las entradas del diccionario es 15 y hay 28 distintos caracteres, tendríamos que tener un array de $28^{15} \approx 10^{22} \approx 2^{72}$ posiciones, que es una cifra inabarcable.

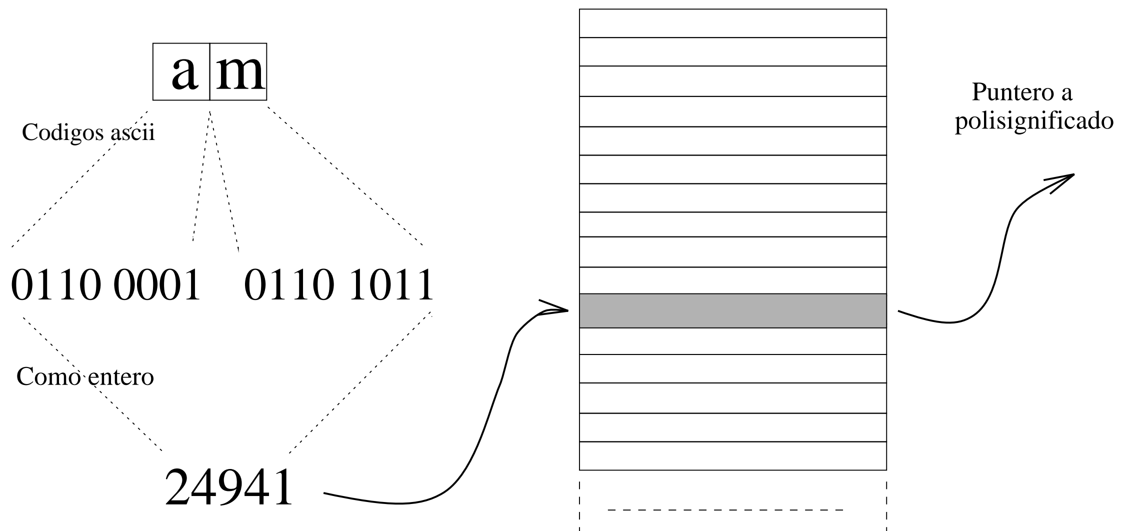

Figura 3.11: Nombre como índice.

La conversión más rápida a número se realizaría interpretando los bytes de la cadena en ASCII como bytes que conforman un número entero. Así

el item “am’, como a es 97 (61hex) y m es 109 (6Dhex) el índice sería la interpretación como entero de $6D61(hex) = 28001$. No obstante en este caso habría 256 posibles caracteres, lo que aumenta aun más el tamaño del array.

El esquema de utilizar los nombres de las entradas como índices directos no es viable en la práctica, sin embargo una aproximación suya con arrays *finitos* sí es posible. Es la **tabla hash**. En esta aproximación se tiene una tabla de un tamaño predeterminado en la que al menos caben todos los punteros a polisignificado de todos los items. Sobre posiciones de esa tabla mapean los nombres de las entradas, pero no directamente, sino a través de una *función hash*. Esta función reparte aleatoriamente los nombres en posiciones. Y lo hace de manera que minimiza la probabilidad de que dos nombres de items distintos mapeen sobre la misma posición del array. No obstante ello sigue siendo posible y en ese caso se dice que hay *colisión*. Para resolverlas cada posición lleva una etiqueta o campo con la cadena a la que está asociada.

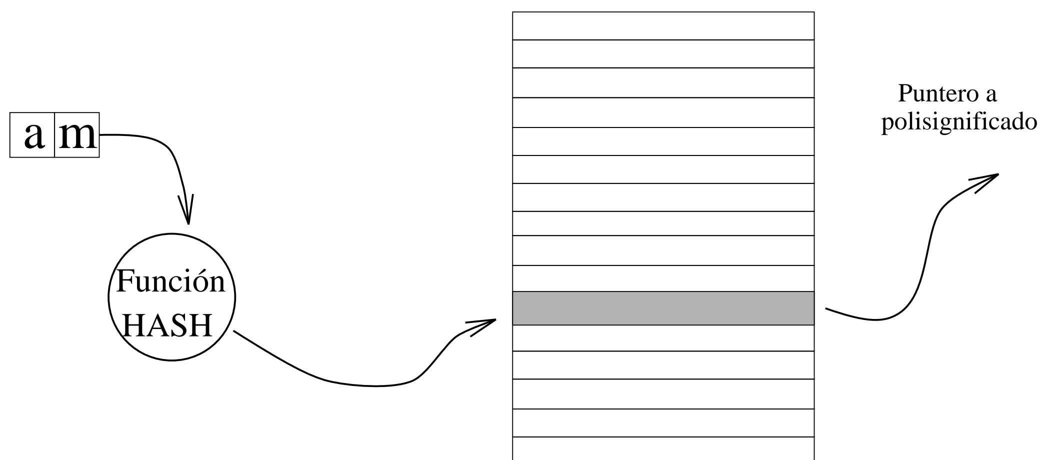

Figura 3.12: Tabla Hash.

Cuando se busca un item y su posición en el array ya está ocupada por otra (se sabe que es por otra porque la etiqueta no coincide con su nombre) entonces se puede por ejemplo buscar linealmente en las posiciones adyacentes o que otra función hash nos dé otro índice en el que buscar. Son distintas maneras de resolver una colisión. Cuando ocurre la colisión el acceso es un poco más lento pero las colisiones son muy improbables si el tamaño de la

tabla es suficientemente grande.

Este esquema de la tabla hash funciona bien incluso con un gran número de entradas. El tiempo de acceso no depende del número de items que almacene, sino más bien del número de colisiones, que se minimiza con un array grande. Sin embargo no se ha seguido porque hemos encontrado otra estructura más ventajosa, el trie.

3.3.3 El trie.

El trie es un árbol no binario de nodos. Cada nodo lleva asociado una letra, y puede tener tantos hijos como letras haya. Para buscar una entrada se sitúa en el nodo raíz y salta al hijo cuya letra es la primera de la entradas, desde ese nodo se salta al hijo suyo cuya letra es la segunda de la entrada y así sucesivamente hasta que se ha recorrido todo el nombre de la entrada. Si en algún momento de este recorrido el hijo con la letra que se busca no existe, entonces tal entrada no está en el trie.

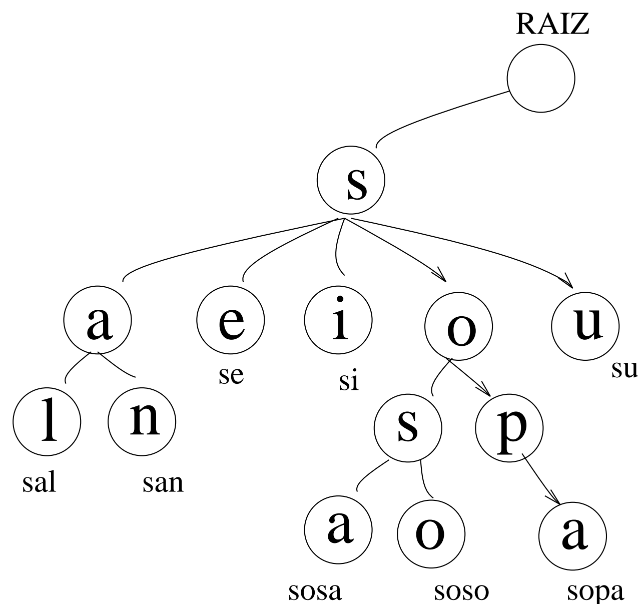

Figura 3.13: Trie como árbol de nodos con múltiples hijos.

En el trie hay dos tipos de nodos los **nodos de paso** y los que son **nodos**

terminales. Los nodos terminales no tienen más hijos y en ellos acaban las palabras.

Esta estructura se puede utilizar para asociar un item con cualquier cosa, por ejemplo un puntero a polisignificado, si guardo ese puntero en el nodo de la última letra del item, en el nodo terminal de ese item. Así para llegar al puntero se va recorriendo el trie, dando tantos saltos como letras tenga la entrada asociada, hasta llegar al nodo terminal. Sólo el mismo item es capaz de llegar a su puntero asociado, porque cada entrada es un camino distinto por el trie, que acaba en un nodo terminal diferente.

La principal ventaja que ofrece el trie es que traduce en muy poco tiempo el nombre de una entrada a su índice asociado. Tarda tanto como saltos tenga que hacer, tanto como letras tenga el item. Haciendo en cada salto comparaciones de 1 solo caracter, no de una cadena. Además este timepo es independiente del número de items que almacene el diccionario.

Además el tamaño del trie no crece proporcionalmente al número de entrads. Cada nodo lleva una letra, pero no hay tantos nodos como letras tiene el conjunto de los nombres de las entradas. Por ejemplo si introduzo "casi' y "casa' no habrá ocho nodos, sino cinco: los de c,a, s, i y el de la a como hijo de la s. Sucede que el camino de "casa' en el trie comparte con el de "casi' las tres primeras letras.

A la hora de plasmar la idea del trie en la práctica hay varias posibilidades, cada una de ellas con distintas características. Antes de ver la que hemos elegido veremos en este epígrafe dos alternativas que desechamos: el trie con los hijos en array y el trie con los hijos en lista.

Trie con los hijos en array.

En esta implementación los nodos, a parte de la letra y el puntero, tienen un array con tantas posiciones como posibles hijos puedan tener, donde caben los punteros a cuantos hijos tenga, para enlazar con ellos.

En esta opción saltar a un nodo hijo es muy rápido, sólo tenemos que utilizar la letra siguiente como índice en ese array y seguir al puntero. Sin embargo esta opción es prohibitiva en espacio porque los nodos ocupan demasiado. Además es ineficiente: un nodo no tendrá en general todos sus posibles hijos, sino un número más pequeño porque por ejemplo a la letra k difícilmente le siguen otra cosa que vocales en las palabras del castellano. Por lo tanto la mayor parte de los arrays de estos nodos estará vacía.

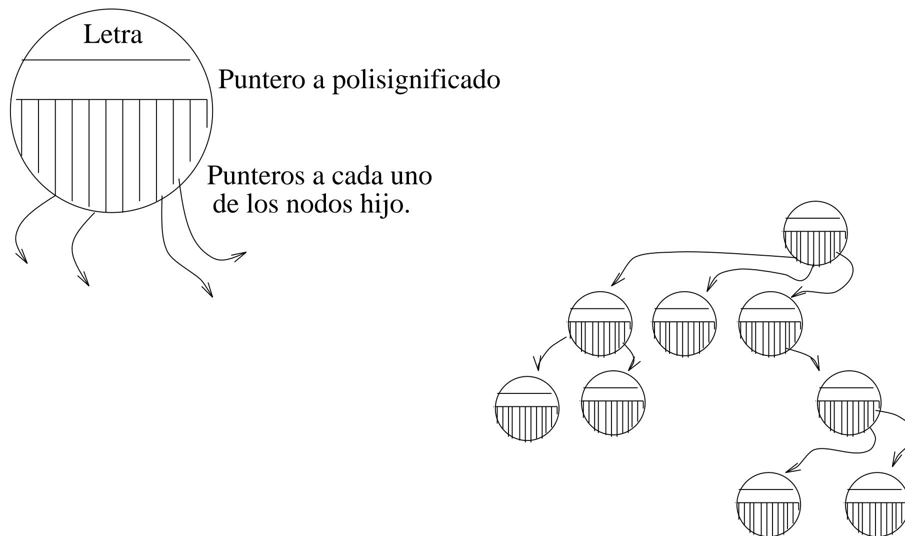

Figura 3.14: Trie con los hijos en array.

Trie con los hijos en lista.

Otra implementación del trie es la que ordena a los hijos de cada nodo en una lista ligada. Ocupa mucho menos que con los hijos en array, porque cada nodo tiene que guardar sólo dos enlaces, en lugar de los 28 del array. De esos dos punteros uno apunta a la lista con sus propios hijos y el otro apunta a su siguiente hermano porque el nodo estará en la lista de hijos de su padre.

Lo que empeora aquí es el tiempo de acceso porque el salto a un hijo supone recorrer la lista de hijos hasta que encuentre aquel por el que quiero continuar la búsqueda. Comparando en cada posición de la lista si el carácter del nodo es el mismo que el del hijo que se busca.

3.3.4 El trie como doble array

La idea del doble array es original de Jun-Ichi Aoe, Katsushi Morimoto y Takashi Sato, y la exponen con detalle en [Jun et.al92] además de comparar sus prestaciones frente a otras implementaciones del trie.

Básicamente esta implementación plantea dos arrays de enteros, el array **base** y el array **check**. Cada posición se ve como un nodo; los distintos campos del nodo son el contenido de los distintos arrays en esa posición. El

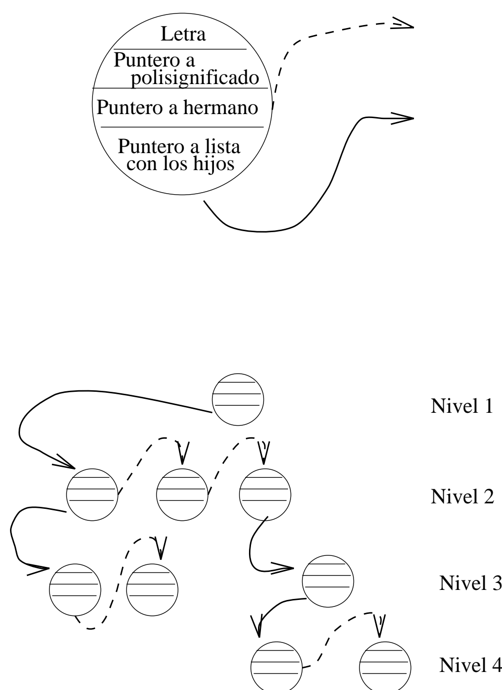

Figura 3.15: Nodo y trie con hijos en lista.

campo check tiene el número del nodo padre. El campo base de un nodo tiene un número que indica la posición a partir de la cual se colocan los nodos de sus hijos. Esa posición se llama **base del nodo**.

Los hijos de un nodo son otros nodos, otras posiciones en el array. Supongamos que tenemos un nodo A y queremos navegar por el trie a través de su hijo M. Los hijos del nodo A se colocan a partir la *base del nodo* A, de manera que para acceder al hijo M basta utilizar el código ASCII de M y sumárselo a la base del nodo A. El resultado es una nueva posición en la que se localiza ese nodo hijo M si este existe. La figura 3.16 ilustra como pasar en el nodo array de un nodo a sus nodos hijo.

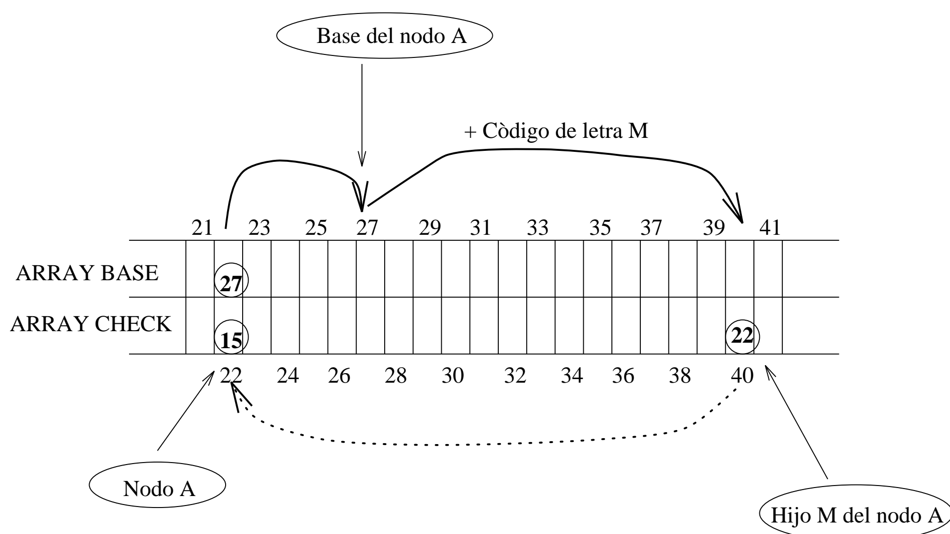

Figura 3.16: Nodo A de paso y su hijo M que sí está en el doble array.

Una vez que mediante este proceso se llega a una posición en la que debe estar el hijo M que se busca del nodo A, para saber si realmente existe ese hijo se verifica el campo check de esa posición. Como el campo check tiene el número del padre, si el de esa posición coincide con el número del nodo A, entonces ese es el hijo M buscado de A. Si no coincide, esa posición está ocupada con otros nodos, no con el hijo M del nodo A, y se dice que el acceso ha fallado. La figura 3.17 ilustra está situación.

Cuando un nodo es de paso efectivamente el contenido del array base es positivo e indica las bases de los nodos. Sin embargo en los nodos termina-

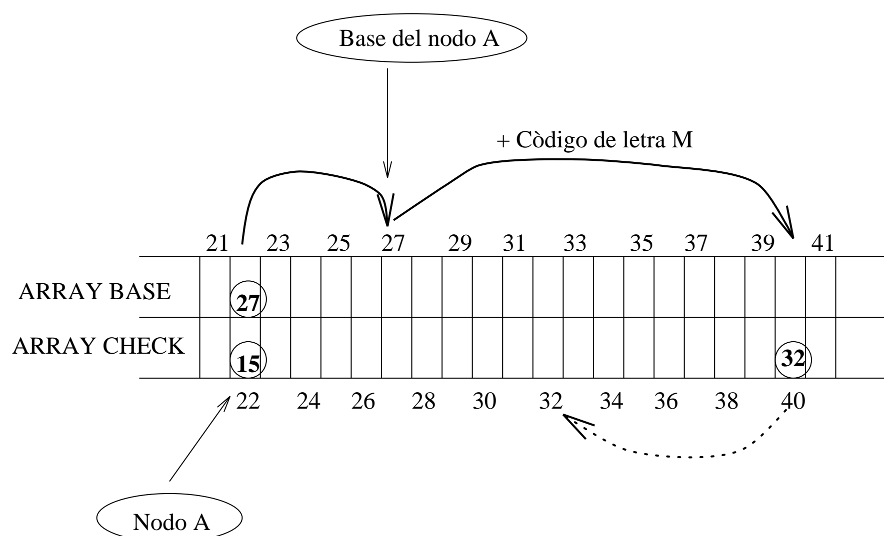

Figura 3.17: Nodo A de paso cuando su hijo M no está en el doble array.

les, como no tienen hijos, no tiene sentido que tengan base. En su lugar se aprovecha este campo para guardar un índice a otro array, el array **entradas**, que guarda los punteros a polisignificados que se asocian a las entradas. Para distinguirlo del uso cuando el nodo es de paso, si es terminal se pone el índice en negativo. De esta manera cuando llegamos a un nodo terminal (y sabemos que es terminal porque su campo base es menor que cero) en el campo base tenemos un índice, negado, que señala la posición del array entradas en la que está el puntero a polisignicado asociado a la entrada que acaba en ese nodo terminal.

Aunque pueda parecer complicado el acceso al trie, es muy ágil, lo complicado es explicarlo. Los arrays no guardan ningún caracter sino que estos están implícitos en la posición que ocupan a partir de la base de su nodo padre. Como no guarda ningún carácter ahorra el espacio que ocuparían todas las cadenas con los nombres de las entradas. Además de ocupar poco es muy rápido porque los propios códigos de las letras se utilizan como incrementos de posiciones en el array. Cada salto a un hijo requiere: mirar el campo base del padre, sumar a ese campo la letra del hijo, y mirar en esa posición el campo check. Este proceso es mucho más rápido que explorar en una lista de hijos por el nodo hijo que buscamos (trie con hijos en lista).

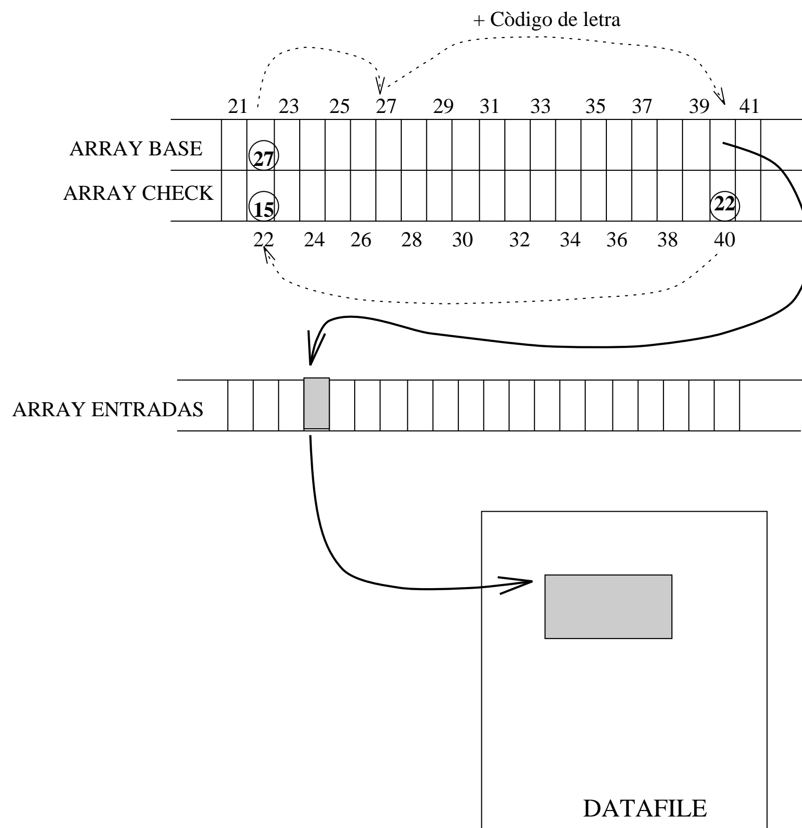

Figura 3.18: Nodo 40 como nodo terminal.

Las prestaciones que el trie en doble array ofrece vienen documentadas en [Jun et.al92] y frente a la realización del trie con hijos en listas (que es la única implementación realmente realizable en la práctica que habíamos visto) el ahorro en espacio es del 15% y el tiempo de acceso ¡*4 veces menor*.

Consulta al trie doble array

La consulta al trie con doble array sigue la misma filosofía que con cualquier trie. Situarse en el nodo raíz y recorrer el trie saltando de letra en letra hasta completar toda la entrada. Si se consigue completar toda la entrada y justo en la última letra se encuentra un nodo terminal, entonces ese nodo terminal guarda, a través del *array entradas* el índice a polisignificado asociado a la entrada. Si durante el recorrido por el trie no se consigue completar la entrada, porque se encuentra antes de tiempo un nodo terminal, o no encuentro el nodo hijo que se busca en alguna letra intermedia entonces tal entrada no está almacenada en el trie y por lo tanto no tiene ningún polisignificado asociado.

Carácter terminador

Si por ejemplo vamos a guardar las entradas "casa' y "casamiento' ocurre que el nodo del trie con la segunda *a* de "casa' debe servir por un lado de nodo terminal para "casa' y por otro de nodo de paso para "casamiento'. Esta doble funcionalidad no se puede satisfacer con un nodo porque su campo base no puede tener a la vez un valor positivo, como correspondería por ser nodo de paso, y un valor negativo, como correspondería por ser nodo terminal.

Para solucionar esto cada entrada está acabada en un carácter extraordinario, el **carácter terminador**, que se toma como parte del nombre. De este modo, si el caracter terminador es por ejemplo "#', entonces en el trie se almacenan "casa#' y "casamiento#'. Ya no hay colisión, el nodo con la segunda *a* será nodo de paso con los hijos m y #. Precisamente su hijo # será el nodo terminal de "casa#' y como por definición el carácter terminador no puede tener hijos nunca se le pedirá que también sea nodo de paso. En la práctica hemos adaptado la idea de [Jun et.al92] para que el carácter terminador sea el propio carácter fin de cadena normal en lenguaje C. Este carácter tiene por código ASCII el 0, no es imprimible y se representa como

'\ 0'. Gracias a utilizarlo como terminador no hay que hacer copia de cada item por el que se pregunta, porque ya acaban en '\ 0'. Si utilizaramos otro carácter terminador que no fuera '\ 0', tendríamos que hacer una copia de cada cadena que se busca, para incorporarle el carácter terminador y buscar adecuadamente en el trie.

Array tail

En realidad se optimiza aun más el trie con la incorporación de un cuarto array, el **array tail**. Este vector guarda las terminaciones de algunas entradas, de manera que en el doble array no se tengan todas las letras, sólo las iniciales que sean suficientes para identificar unívocamente cada entrada entre el resto. Así por ejemplo, si sólo tuvieramos las entradas "santo", "sala" y "sacarina" entonces la terminación *-arina* no estaría en el doble array porque la raíz *sac-* ya distingue a "sacarina' del resto de las entradas.

El *array tail* guarda las **colas** o terminaciones no imprescindibles. Es un array de punteros a cadenas de caracteres. Cuando voy a buscar una entrada que no está completamente en el doble array, pero que sí está en el trie, entonces aparece un nodo terminal antes de recorrerla por entero. Como hemos visto ese nodo terminal guarda en su campo base un número negativo, con cuyo valor absoluto indexo en *array entradas* para encontrar el puntero a polisignificado. Si con ese mismo valor absoluto indexo en el *array tail* se obtiene el puntero que apunta a la *cola* del item que hay en el trie y tiene ese nodo terminal. Si la *cola* coincide con el resto de la entrada que me queda por recorrer entonces la entrada sí está en el trie, aunque no completamente en el doble array. Si *cola* y resto difieren en lo más mínimo es que esa entrada no está en el trie.

Cuando el camino en el trie hacia el nodo terminal acaba exhaustivamente con todas las letras de la entrada, entonces la posición del *array tail* no guarda nada relevante.

Valoración

Esta estructura del trie en doble array ofrece un tiempo de consulta mínimo, porque se recorre fácilmente. Por ser trie sólo hay que dar tantos saltos como letras tenga la entrada (menos si tiene parte en el array con las *colas*). Y por ser doble array los saltos son tan sencillos como consultar dos posiciones de

un array.

Por el contrario el tiempo de construcción de un trie es relativamente mayor. Esta es la desventaja de los tries, que son lentos de construir. En cierta medida el precio de unas consultas tan rápidas es una generación lenta de la estructura. El diccionario que use el trie será rápido en responder a las preguntas pero lento en compilar una base léxica, porque al compilar se genera el trie. Sin embargo esto no supone un grave inconveniente para nuestro diccionario porque las bases léxicas se compilarán *una vez* y luego las estructuras creadas se vuelcan a ficheros. Cada vez que se vaya a utilizar el diccionario no hay que generar el trie, basta cargar el trie ya construido que reside en los ficheros y esto es muy rápido. Por lo tanto desplazar tiempo desde las consultas a la construcción es beneficioso porque el coste de la lentitud de la compilación no se paga cada vez que utilizo el diccionario.

Por estas ventajas de espacio soportable y consultas rápidas hemos elegido esta implementación como estructura para almacenar todos los nombres de las entradas.

3.3.5 El segmentador

Como vimos en el capítulo 2 la base léxica que se ha elaborado en el *Grupo de Procesamiento de Lenguaje Natural* de la ETSIT es una base léxica de formantes, no de palabras enteras. Se deja la composición de palabras a un procesador morfológico.

Una **segmentación** de una palabra es una partición de la palabra en formantes, un conjunto ordenado de segmentos disjuntos que concatenados forman la palabra. Para encontrar la información lingüística de una asociada a una palabra el procesador morfológico busca segmentaciones posibles y una vez halladas la información del vocablo se construye desde la de sus segmentos según la gramática de la palabra.

El análisis morfológico se suele abordar en dos pasos. En primer lugar mediante consultas de si tal segmento existe en el diccionario se calculan las **segmentaciones de existencia** del vocablo. Lo único que se asegura en ellas es que cada pedazo tiene significado propio asociado en el diccionario. En segundo lugar se usa la gramática de la palabra para descartar de esas *segmentaciones de existencia* aquellas que no satisfagan las regras de formación de la palabra. Al final de este segundo paso sólo quedan segmentaciones morfológicamente correctas de las palabras.

Algoritmo de segmentación

Las segmentaciones de existencia pueden conseguirse generando todas las posibles combinaciones de segmentos y preguntando al diccionario si cada uno de esos trozos tiene sentido. Primero se prueba la primera letra de la palabra como segmento inicial, si no tiene sentido se prueba con la primera y la segunda juntas, si tampoco tiene significado pruebo con el segmento formado con las tres primeras letras, y así repetidamente. Se prueban todos los posibles segmentos iniciales hasta el segmento que abarca la palabra entera.

Es un algoritmo recursivo. En cuanto un segmento inicial tiene sentido se lanza la segmentación del resto de la cadena con una llamada recursiva al algoritmo. Si el resto se puede segmentar entonces se ha encontrado una segmentación válida para toda la cadena: *segmento inicial + segmentos del resto.*

Este algoritmo se puede lanzar externamente al diccionario y entonces éste debe limitarse a responder sí o no a cada segmento por el que se le pregunta si tiene significado. En este caso tendríamos un **segmentador universal** porque funciona independientemente de cual sea la estructura del diccionario. Sin embargo si se traslada la tarea al interior del diccionario, esto es, si se le incorpora la funcionalidad de segmentar palabras, entonces se puede aprovechar la estructura trie para acelerar el cálculo de segmentaciones. Sobre el trie es más rápido que con un segmentador universal.

Supongamos que en el diccionario sólo tenemos los formantes *a*, *mar*, *cal*, *am* y *ar*. Cuando se quiere segmentar *calamar* se comienza probando con el segmento *c*. Como no tiene sentido pruebo con *ca*. Como *ca* tampoco tiene sentido pruebo con *cal*, que sí lo tiene. También pruebo con *cala*, *calam*, *calama* y *calamar*. Si se segmentara universalmente una vez encontrado *cal* se recorrería el trie desde el principio para preguntar por *cala* y otra vez desde el principio para *calam*, etc porque cada consulta al diccionario se responde independientemente. Sin embargo si se segmenta sobre el trie, una vez encontrado *cal* se explora si el nodo *l* tiene un hijo *a* para conseguir *cala*. Como no lo tiene no se sigue explorando segmentos mayores por ahí porque el trie me garantiza que entonces *calam*, *calama* y *calamar* no tienen existencia en el diccionario como trozos independientes. Por lo tanto se ahorra ese tiempo.

Como es mucho más rápido hemos decidido que el procesador morfológico no segmente y que sea el código del diccionario quien lo haga, para así aprovechar las ventajas de hacerlo sobre el trie.

En el segmentador universal es indiferente el sentido en el que se empiece a explorar los posibles segmentos de la palabra, si de derecha a izquierda o al revés. Pero para sacar partido del trie debe hacerse de izquierda a derecha porque las entradas están en el trie en ese sentido desde la raíz.

Capítulo 4

DISCOS Y MEMORIA

4.1 INTRODUCCIÓN

Hasta ahora nos hemos ocupado de explicar la estructura lógica del diccionario: por un lado el trie, que permite asociar una cadena entrada con sus significados, y por otro esos mismos significados que son haces de rasgos. Pero no nos hemos preocupado de explicar dónde reside realmente toda esa información, ni por ejemplo qué tipo de punteros enlazan entre sí los distintos rasgos de un haz. Y sin embargo es importante porque dónde se ubiquen estas estructuras afecta sobresalientemente a la eficiencia temporal del diccionario. A ello se dedicará este capítulo.

Tanto para poder utilizar el trie como los haces de una sesión a otra se hace inevitable el uso del disco. El disco guarda en unos archivos toda la información necesaria para no tener que compilar la base léxica cada vez que inicio una aplicación que consulta el diccionario. Para esto se dividió el proceso de utilizar la base léxica en varios pasos, de manera que las estructuras se generan de una vez con la expansión y la compilación, y después para consultarlo sólo es necesario cargar las estructuras compiladas que residen en esos archivos. El empleo del disco es pues ineludible, pero no es este uso que vamos a discutir ahora, sino la posiblidad de consultar y trabajar las estructuras en el disco directamente, cuando se compila o consulta el diccionario. En principio podríamos ubicar las estructuras en el disco físico o bien en la memoria principal. Si se ubican completamente en disco los accesos a ellas serán lentos, no se ocupará memoria principal y la *carga* de estructuras

se limitará a abrir los ficheros correspondientes del trie y de los haces. La segunda opción es trabajar con ellas en memoria. Entonces los accesos serán más rápidos y la carga copiará en memoria, para ser accedidas desde allí, las estructuras que están en los discos.

Repasaremos las plataformas actuales sobre las que es deseable que corra el código del sistema, pues las restricciones y condiciones que nos impongan condicionan la decisión final. Por un lado las estaciones de trabajo más comunes (Sparc-stations de Sun, Hewlett-Packard,..) vienen equipadas al menos con 32 MB de memoria física principal, aunque el espacio de direccionamiento sobre ellas sea mucho mayor (4 GB) porque utilizan memoria virtual. Por otra parte, las configuraciones más usuales de ordenadores personales suelen tener 8 o 16 MB de memoria principal. El mensaje de fondo que nos envían las plataformas hardware reales es que la memoria principal no es un recurso ilimitado. Si lo fuera guardaríamos en ella todas la estructuras del diccionario, porque los accesos son mucho más rápidos que los accesos a disco, y nos olvidaríamos del asunto. La otra solución extrema, de guardar todo y trabajar desde la información en los discos tampoco es razonable. La ventaja principal consistiría en que el disco tiene un tamaño virtualmente ilimitado con lo que habrá espacio suficiente para el diccionario. Los discos duros más pobres en la actualidad tienen unos 500 MB, mientras que los tamaños conseguidos con bases léxicas suficientemente completas rondan los 12 MB. Otra ventaja es que no consumiríamos memoria principal, lo que haría a nuestro código menos exigente con la plataforma hardware, al necesitar menos memoria para funcionar. Sin embargo estos pros no compensan la lentitud que conllevan los accesos a disco, que haría a nuestro sistema inútil en la práctica para muchas aplicaciones por excesivamente lento.

Hemos buscado una solución intermedia que recoja las virtudes de ambos extremos: la velocidad de la memoria y el tamaño ilimitado de los discos. El diccionario puede funcionar completamente desde disco, que es la ubicación por defecto, pero en la medida en que nuestra plataforma tenga memoria principal podremos hacer que nuestro diccionario la use, residiendo parcialmente en ella y acelerando el funcionamiento. Cuanta más memoria haya, más partes del diccionario podrán aprovecharse de ello y más rápidamente se ejecutará el código. En cuanto haya cantidad suficiente el trie se copiará en memoria. Para agilizar las consultas la memoria se utilizará como cache del disco, y para agilizar la compilación se empleará como buffer del disco.

Para comprender la incidencia tan notable que sobre la velocidad de

Tabla 4.1: Tiempos típicos de acceso.

	MEMORIA	DISCO
tiempo de acceso (ns)	100	20.000.000
capacidad transferencia (MB/s)	133	4

respuesta del diccionario tiene que éste resida en el disco o en la memoria principal basta observar la tabla 4.1 con valores típicos de los tiempos de acceso y anchos de banda de ambos soportes.

Aunque el Sistema Operativo pueda suavizar con caches de entrada/salida el tiempo de consulta al disco, como por ejemplo en Unix, la diferencia sigue siendo notable. Así lo confiman además las pruebas realizadas, que ya comentaremos en el capítulo siguiente.

4.2 DICCIONARIO LÓGICO

Conviene resaltar que en principio todo el diccionario puede funcionar desde el disco, y es el disco el que almacena toda las estructuras completamente, tanto el trie como el conjunto de haces de rasgos. Cuando se va a utilizar memoria ésta se empleará como si fuera un espejo del disco. Si un trozo del diccionario, bien sea el trie, un haz de rasgos o cualquier otro, se va a copiar en memoria para agilizar la ejecución, entonces el original en disco y su homólogo en memoria son identicos byte a byte.

En C el disco es accesible a traves de los ficheros, los cuales tienen dos modos de utilización: el modo ascii y el modo binario. Utilizaremos el disco en MODO BINARIO porque en este modo el sistema operativo no interfiere en las transferencias entre memoria y disco, y puede conseguir copias fieles. En el modo ascii el sistema operativo interpreta las transferencias de bytes como de caracteres e inserta caracteres especiales, por ejemplo retornos de carro al final de las líneas.

Como vimos en el capítulo 3, la estructura trie guarda arrays de enteros largos, de caracteres y de punteros a significado. Tanto si se decide trabajar con el trie en memoria como si se hace desde disco el contenido es transparente a su ubicación. Es decir que los caracteres siguen viendose como

tales independientemente de que los bytes que ocupa los lea del disco o de la memoria. En cuanto a los enteros largos, estos se van a tomar como índices de array. Si cada posición ocupa 4 bytes, la posición 30 comenzará en el byte 30 * 4, tanto si el array está en disco como si está en memoria.

El contenido de los significados son haces ligados de rasgos. Esa ligazón se consigue con punteros. Como la ubicación por defecto de los haces es el disco, esos punteros son **punteros de disco** y no direcciones de memoria física. Estos enlaces son números simplemente que se pueden ver como direcciones de un espacio lógico o virtual donde ubicamos nuestro diccionario. El tamaño de este espacio virtual depende del que se elija para los punteros sobre él. Si se trabaja con punteros de 4 bytes, como hemos hecho, el espacio comprende $2^{32} = 4$ GB, que resulta más que suficiente para los diccionarios utilizados en la práctica. Por lo tanto el fichero con los haces guarda el DICCIONARIO VIRTUAL, pero también podemos llevarnos partes suyas (el trie, los haces más accedidos, etc) a memoria. Con esta manera de verlo estamos separando el diccionario lógico de su localización físca porque los **enlaces virtuales** se pueden traducir en la práctica a posiciones de memoria o a posiciones en el disco, dependiendo de donde resida la unidad del diccionario a la que apunta. La traducción a posiciones en disco es inmediata porque el mismo puntero lógico es un índice dentro de un archivo y señala a una posición en el disco. Si la zona del diccionario se ha llevado a memoria el puntero se traduce, por ejemplo a través de una tabla, a la zona homóloga de memoria.

El trie y la estructura con los haces siempre residen físicamente en el disco, que es el soporte básico y permite la reutilización de una sesión a otra del diccionario compilado. Aparte, el diccionario se puede incorporar parcialmente a memoria para acelerar la ejecución. Esta incorporación puede ser gradual: si no tenemos memoria se puede trabajar con todo en disco; si tenemos poca, el trie en memoria y seguir trabajando con los haces en disco; si tenemos un poco más, con el trie y parte de los haces (los más usados) en memoria; y si tenemos mucha, con todo en memoria. De esta manera el diccionario se adapta a lo que la plataforma hardware le ofrezca. Aprovecha toda la memoria disponible para situar más partes en ella y acelera así el funcionamiento, pero es capaz también de funcionar cuando sólo tenga disco.

Sin esta indirección del diccionario virtual hubiera sido imposible despegarse de los inconvenientes de los punteros directos a memoria. Si las estructuras de los haces estuvieran ligadas con ellos, en el archivo que guarda el diccionario compilado de una sesión a otra habría direcciones de memoria

enlazando los nodos de los haces. Esto implicaría que nuestra aplicación ocupará siempre la misma zona de memoria, lo cual es inadmisible. Además requeriría que toda la estructura de haces estuviera en memoria principal, y ya hemos visto que hay plataformas que no tienen suficiente memoria para ello. Al separar diccionario lógico de su ubicación física real estamos abriendo la puerta a la flexibilidad de localización y haciendo independiente de ella al manejo del diccionario, que siempre se usa con punteros virtuales. Luego esos punteros se traducen a un soporte u otro dependiendo de lo que tengamos disponible.

Un esquema dual al utilizado sería el de utilizar direcciones de memoria como punteros virtuales. De este modo la traducción de un puntero lógico a dirección de memoria sería directa puesto que él mismo lo es. Los enlaces virtuales que sólo residan en disco se traducen a una posición de disco con una tabla, una especie de *anticache*. Este esquema no aporta ninguna ventaja adicional. Para poder utilizar esa traducción directa tendríamos que recorrer toda la tabla de la *anticache* para darnos cuenta de que tal puntero lógico no está y entonces usarlo como dirección de memoria, con lo cual deja de ser una traducción tan rápida. Además no se libera de las ataduras de que tal zona del diccionario tenga que residir siempre en tal zona de memoria si se tiene que incorporar, lo cual hace muy desaconsejable esta opción.

4.2.1 Interfaz de original e interfaz de copia.

Ya hemos visto que las aplicaciones pueden utilizar al diccionario en modo consulta del original y en modo copia. Las estructuras originales con los haces están enlazadas con punteros virtuales, mientras que las copias estarán enlazadas con punteros de memoria, porque los programas usuarios no entienden de enlaces lógicos.

En el modo consulta del original el diccionario nunca entrega copia de sus respuestas cuando se le pregunta por los significados de una entrada, devuelve punteros sobre su estructura original. Esos punteros son punteros virtuales. Por ejemplo si preguntan por los significados de *caracola* y estos empiezan en la posición 1023 del datafile, entonces el diccionario devuelve ese puntero virtual 1023. Para consultar todas las estructuras de los haces a través de punteros virtuales se ha desarrollado el código manipulador que ya comentamos, el cual ofrece al usuario una interfaz con los niveles básico y elaborado. Obviamente este código entiende de punteros lógicos y los sigue

sin problemas.

El otro empleo del diccionario es el empleo destructivo de sus respuestas. En este modo cuando le preguntan por una entrada, el diccionario la busca y si la encuentra hace copia de su estructura de significados. Al copiar los punteros que enlazan la réplica son punteros de memoria. El usuario podrá alterar estas estructuras sin afectar al diccionario. Esta traducción de enlaces no ocupa tiempo, porque se hace a la misma vez que se recorre la estructura original para replicarla. Para manejar estas estructuras en memoria se ha desarrollado un código manipulador análogo al que trata las estructuras originales y ofrece tambien una interfaz homóloga con sus niveles básico y elaborado. Obviamente este código entiende los punteros de memoria.

Los módulos del código manipulador se han *duplicado*, para tener esas dos interfaces: la de estructuras lógicas y la de estructuras en memoria.

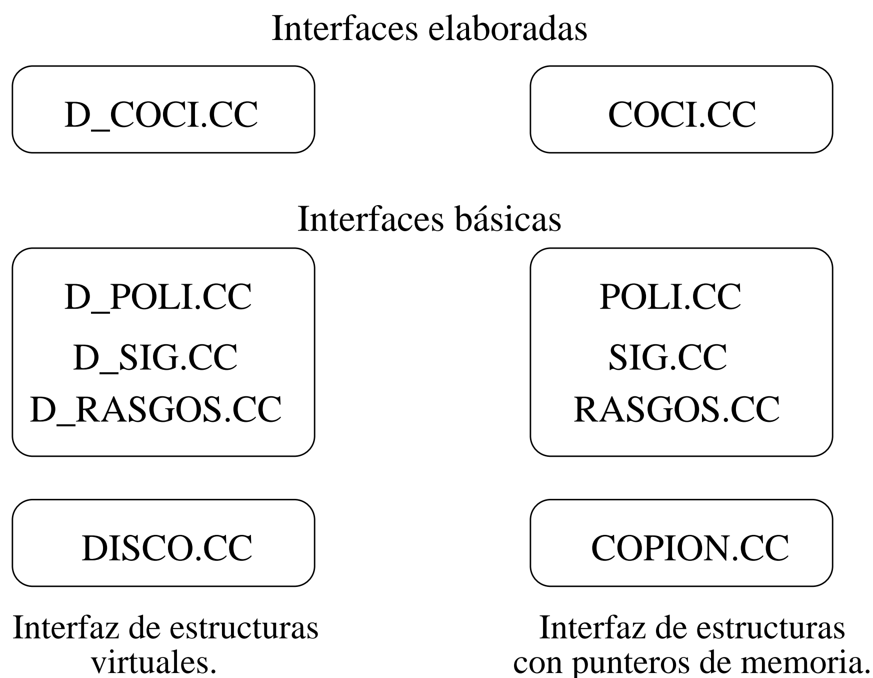

Figura 4.1: Módulos de los dos interfaces

Como se observa en la figura 4.1 la interfaz del original utiliza al módulo *disco.cc* para resolver la localización física final de los punteros lógicos. Sin

embargo en la base de la interfaz de memoria está el módulo *copión.cc* que crea la estructura de memoria replicando su homóloga lógica.

Resumiendo este epígrafe podemos decir que existe el uso de solo consulta y el uso modificativo del diccionario. Paralelamente existen las estructuras originales y las copias en memoria. Cada una de ellas se manejan con códigos manipuladores e interfaces distintas porque en la original los punteros son virtuales mientras que en la copia son direcciones normales de memoria.

4.3 USO DE MEMORIA PARA EL TRIE Y LOS HACES DE RASGOS.

Por el momento no hemos diferenciado entre la estructura trie y la estructura de los haces, pero ahora sí que vamos a hacerlo porque se acceden de forma muy distinta y esto determina que se traten de distinto modo en cuanto a su ubicación.

4.3.1 El trie y la memoria.

El trie se accede muy frecuentemente tanto en consulta como en compilación; sobre todo los arrays base y check, a través de los cuales se navega. Como se utiliza tan a menudo conviene optimizar su tiempo de acceso. Por ello es muy beneficioso que resida en memoria, sobre todo en la compilación donde el uso del trie es más intensivo. Los accesos, especialmente los de lectura, son muy deslocalizados en cuanto a posiciones de los arrays que son consultadas, es decir, se distribuyen pseudoaleatoriamente de modo uniforme a lo largo de toda la longitud del array. Por ello se determinó tratar los arrays monolíticamente, esto es, o todo en memoria o todo en disco.

Los tries conseguidos para bases léxicas respetablemente completas ocupan unos 1'6 MB y como no son de crecimiento exponencial sino lineal con el tamaño de la base léxica, no son esperables tamaños exageradamente mayores. Dado este orden de magnitud y que se acceden constantemente en la búsqueda creemos que es muy recomendable emplear esa memoria en el trie. No obstante para plataformas pobres se incluye la posibilidad de trabajar con el trie en el disco. La posibilidad se puede habilitar con una directiva de compilación que escoge la parte del código adecuada. En las plataformas con poca memoria se compilarán los módulos del código con la

opción USAR_DISCO_PARA_EL_TRIE, y en las que tengan suficiente, sin ella porque por defecto se situa al trie en memoria.

4.3.2 Haces en consulta: la cache.

El acceso a los haces es radicalmente distinto al del trie. Incluso es diferente si estamos en tiempo de compilación o de consulta. Por ello se han optimizado por separado ambos usos.

En consulta los haces se consultan una vez en cada entrada, para buscar sus significados. Obviamente habrá unas entradas que serán más frecuentes que otras, por lo cual sus significados serán los más frecuentemente accedidos. En particular las terminaciones verbales, los morfemas de género y número, así como preposiciones y adverbios más comunes serán consultados muy frecuentemente, mientras que por ejemplo la raíz *otorrinolaringolog* será consultada rara vez. Además el fichero con los haces es muy grande. Los archivos que se han conseguido para bases léxicas respetablemente completas ocupan unos 8'6 MB. Es previsible que su tamaño aumente linealmente a medida que crezca la base léxica (bien porque se añadan más entradas, bien porque se incorporen nuevos rasgos...).

Asi pues, tenemos un fichero muy grande con haces que tiene unas zonas (los significados de las entradas más solicitadas) mucho más accedidas que otras. Al ser de gran volumen no es conveniente tenerlo todo en memoria, porque tendríamos muchas partes ocupadas con significados que nunca o rara vez se necesitarán. Por ejemplo en arquitecturas PC con 8 MB de memoria física no cabe toda la estructura de haces. Lo ideal sería tener en memoria (y por ello acceder muy rápidamente) sólo las zonas que se consultan más frecuentemente, mientras que las zonas menos solicitadas pueden residir en disco (acceso lento) porque como se consultan poco no deterioran mucho la rapidez del sistema.

La ubicación de las haces en tiempo de consulta será mixta: una parte de ellos residirá en memoria y otra en disco. La localización por defecto será el disco, pero para agilizar las respuestas se incorpora una cache del archivo en la memoria, que retendrá los significados más frecuentemente accedidos.

4.3.3 Haces en compilación: el buffer.

Durante la compilación los accesos a los haces son poco repetitivos y la operación fundamental consiste en escribir secuencialmente los haces en el disco. La manera de acelerar este procesamiento con memoria es utilizarla como buffer. Con esto se minimiza el numero de accesos reales a disco y se agiliza la compilación. El tiempo de acceso al disco es comparativamente muy costoso, más que el de transferencia, porque hay que posicionar las cabezas lectoras y esta es una maniobra física relativamente lenta. Con el buffer se aprovecha más cada posicionamiento para transferir más cantidad de bytes a disco. Al mininimizar el número de accesos la compilación se realiza más rápidamente.

4.4 CACHES

La idea de las caches nace en el entorno de las arquitecturas de ordenadores y trata de acelerar la jerarquía de memoria de los sistemas computadores. Hay varios tipos de dispositivos de almacenamiento en un sistema computador, que se ordenan en la llamada jerarquía de memoria, como se ilustra en la figura 4.2. Una jerarquía normal tiene memoria cache, memoria convencional y disco o almecenamiento secundario. A medida que descendemos en la jerarquía los dispositivos tienen mayor capacidad pero son más lentos. Cuando la CPU necesita un dato o una instrucción se lo pide al nivel mas alto de la jerarquía, si está allí lo utiliza, pero si no, se traslada la petición al nivel inmediatamente inferior

Las memorias cache, son más rápidas que las memorias normales, pero son de una capacidad más limitada y sensiblemente más caras; cuanto mayores, más caras. Incorporandolas al nivel mas alto se persigue reducir el tiempo de respuesta de la jerarquía de memoria a tiempos parecidos a los que ofrece una cache por si sola. Y ello sin que el coste global subiera mucho, es decir, sin que los 32 Mb de memoria principal de nuestro sistema fueran de las carísimas memorias cache.

Se observó que durante la ejecución del código de cualquier programa, los accesos a la memoria tienen mucha localidad espacial, es decir, datos o instrucciones cuyas direcciones son próximas tienen alta probabilidad de ser referenciadas en instantes de tiempo próximos. En otras palabras, tras

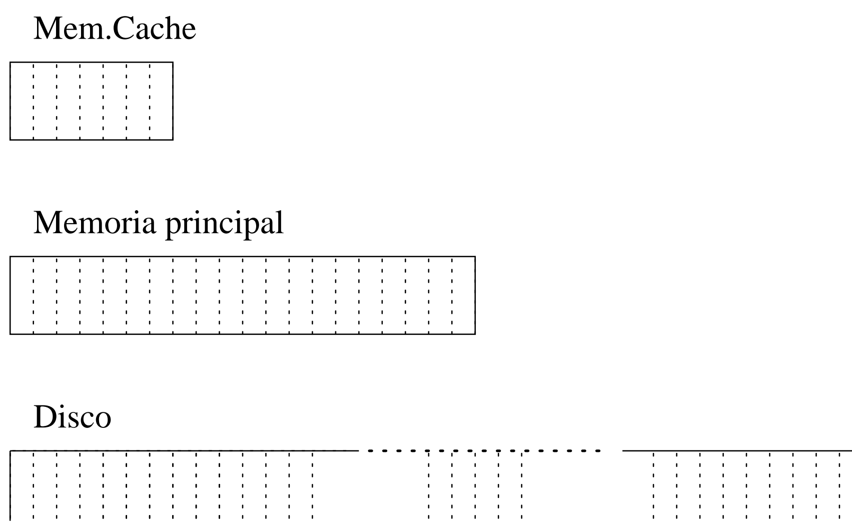

Figura 4.2: Jerarquía de memoria

acceder a la posición 1000, en el instante siguiente las posiciones adyacentes tienen más posibilidades de ser accedidas que posiciones más alejadas. Para aprovechar esta localidad se divide la memoria principal en bloques, y en cada momento se tienen copiados los bloques de más frecuente acceso en la cache. No se llevan todos porque no caben. No obstante, como se tienen los bloques más solicitados se consigue que la mayoría de los accesos se sirvan desde la cache. Así el tiempo medio de acceso se parece más al de la cache que al de la memoria convencional. Se llama *fallo* al hecho de no encontrar en la cache al bloque buscado, y *acierto* cuando sí se encuentra. Si denotamos p a la probabilidad de acierto, $1-p$ a la probabilidad de fallo, $t.mem$ al tiempo en que tarda la memoria principal en servir un acceso y $t.cache$ al tiempo que tarda en hacerlo la cache, entonces el tiempo medio de acceso a la memoria ($t.medio$) podemos escribirlo como

$$t.medio = (p * t.cache) + ((1-p) * t.mem) \tag{4.1}$$

4.4.1 Ubicación de bloques en la cache.

Hay tres clases de organización de los bloques dentro de la cache:

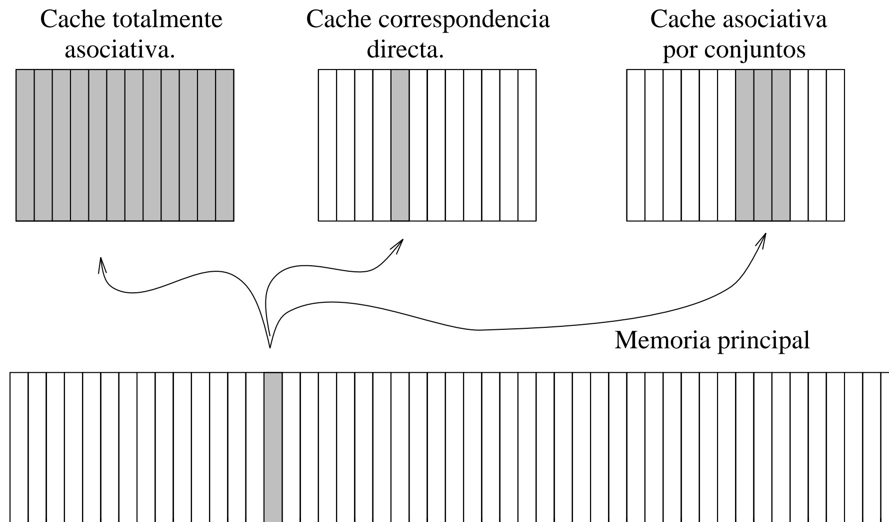

Figura 4.3: Tipos de cache

1. **Correspondencia directa**. Todo bloque tiene un único lugar donde se puede almacenar.

2. **Totalmente asociativa**. Un bloque se puede situar en cualquier lugar de la cache.

3. **Asociativa por conjuntos**. Un conjunto es un grupo de dos o más bloques. Un bloque se asocia a un conjunto y solo puede estar en él, en cualquier posición dentro del conjunto.

4.4.2 Búsqueda de bloques en la cache.

Las posiciones de la cache incluyen un identificador que señala qué bloque de memoria está contenido en esa posición. Cuando se quiere, por ejemplo, acceder a una dirección del bloque 302 de memoria:

- En *correspondencia directa* se calcula la posición de cache asociada al bloque 302 y se mira el identificador del bloque que ocupa esa posición. Si coinciden se sirve el acceso desde el bloque copiado en la cache, y si no el 302 no está en la cache.

- En *totalmente asociativa* se exploran todas las posiciones de la cache hasta encontrar alguna cuyo identificador de contenido sea el del bloque que buscamos. Si una vez exploradas todas las posiciones no hay ninguna que coincida el bloque no está en la cache.

- En *asociativa por conjuntos* se calcula el conjunto correspondiente a ese bloque y se explora, solo en las posiciones de ese conjunto, si alguna contiene el bloque que buscamos.

4.4.3 Sustitución de bloques ante fallos.

Cuando se trae un nuevo bloque desde memoria principal en caches de correspondencia directa va a ocupar su posición asociada. En caches asociativas o asociativas por conjuntos sin embargo hay que decidir dónde ubicarlo. Si hay alguna posición vacía se ocupa, pero si dentro de los posibles destinos todos están ocupados entonces hay que echar a algún bloque para hacer sitio al nuevo. Un criterio puede ser elegir esa víctima a desalojar aleatoriamente entre los posibles destinos (criterio random). Otro, muy usado, es el de elegir como víctima el blque que haya sido accedido menos recientemente (criterio LRU Least Recently Used). Este criterio se basa en la suposición de que si un bloque fue accedido hace poco tiempo, probablemente será accedido en un futuro inmediato, por eso no lo expulsa de la cache. En la práctica el criterio LRU se suele implementar bien con contadores de tiempo o bien con una pila ordenada. Con los contadores de tiempo se marca el ultima instante en que cada bloque fue accedido, de modo que se puede hallar fácilmente el accedido hace más tiempo. Con la pila se tiene una lista por cada conjunto en la que se ordenan las posiciones según el último acceso, manteniendo la más reciente en cabeza de la lista y la menos en la cola. Existen multitud de otros criterios como FIFO, los que se basan en bits de referencia, o los que se basan en contar el numero de accesos a cada bloque en un cierto periodo de tiempo. Cada uno de ellos ofrece una política distinta de sustitución y una sobrecarga de instrucciones que al final se reflejan en la probabilidad de fallo y el tiempo en servir desde cache.

4.4.4 Lectura y escritura con cache.

La operación de leer de un bloque cuando la memoria tiene cache consiste en buscar si el bloque está en la cache; si está se lee de la copia en la cache y si no, cargo el bloque desde memoria convencional en la cache y desde allí leo la posición buscada.

La escritura de un bloque cuando se utiliza la cache consiste en buscar el bloque en la cache de igual modo que una lectura, escribir en la copia de la cache y escribir en el original de la memoria convencional. Caben aquí varias estrategias, como escribir a la vez en ambos niveles de la jerarquía (write through) o sólo escribir en la cache y aplazar la escritura en memoria hasta que el bloque se expulse de la cache (write back).

De la misma manera en que las memorias cache se utilizan para acelerar los accesos a memoria principal, la memoria principal se puede utilizar para acelerar los accesos al disco. Es en este marco en el que se encuadran las caches de entrada/salida que algunos sistemas operativos implementan, como por ejemplo Unix. También con esa idea, paralela a la de memorias cache, se ha realizado la *cache de lectura* para los datos de nuestro diccionario.

4.5 CACHE DE CONSULTA.

La cache que se ha implementado es una cache sólo de lectura, asociativa por conjuntos y con algoritmo de sustitución LRU. Como es una cache de lectura de haces solo agiliza las consultas, no la compilación. De hecho en la compilación la cache está desactivada y solo se habilita cuando va a usarse el diccionario para consultas.

La *cache* realizada consiste en una tabla de traducción y unas zonas de memoria dinámica ocupada, como ilustra la figura 4.4. La tabla de traducción (figura 4.5) asocia los punteros virtuales con sus traducciones a punteros de memoria física.

4.5.1 Unidades de cache

Los haces del diccionario se pueden incorporar a la memoria por trozos, o unidades. Serán las porciones mínimas que se pueden llevar a la cache, unidades de trasiego entre disco y memoria. Para nuestro sistema estas serán los

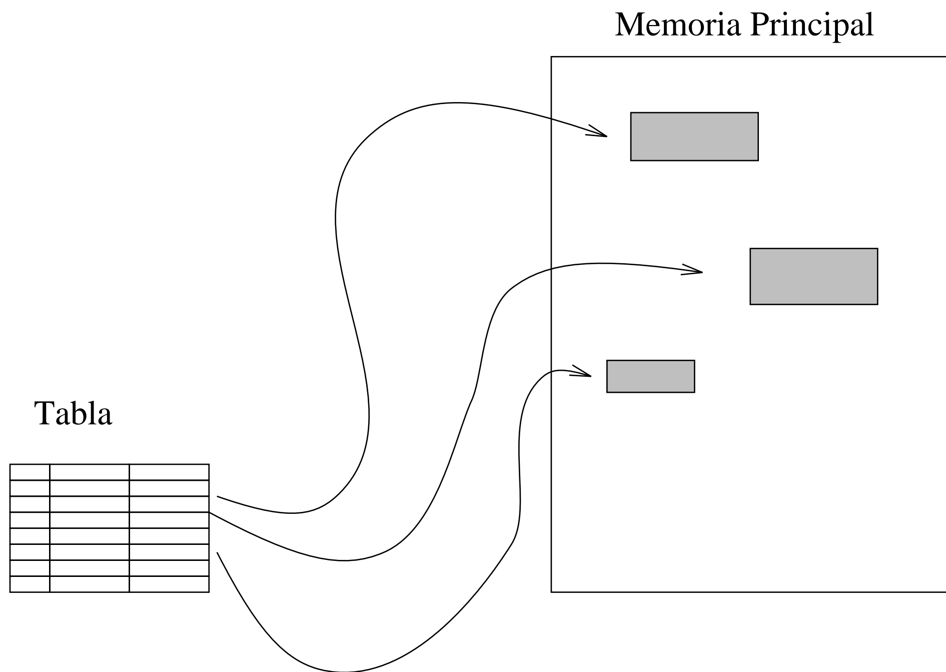

Figura 4.4: Cache: tabla y memoria dinámica

Reloj	Punteros virtuales	Direcciones de memoria

Figura 4.5: Tabla de traducción de la cache

nodos rasgo, los nodos significado, los nodos atómico y las cadenas de caracteres. Es decir, cada uno de los nodos de las estructuras ligadas que tenemos y cadenas. Para mover todo un haz a memoria tenemos que mover todos los nodos y cadenas de que se compone. Podría haber considerado unidades mayores pero se ha escogido la granularidad fina porque permite mayor flexibilidad en el ajuste de la cache a lo que realmente se está accediendo con mayor frecuencia. Por ejemplo si hay una aplicación a la que sólo le intersan determinados rasgos, en la cache sólo entrarían esos, mientras que los no usados no ocuparían ese valioso espacio. Además la granularidad gruesa no hubiera aportado ninguna ventaja. Si la unidad fuera, por ejemplo, un haz de rasgos, como la memoria tiene que ser copia exacta de la unidad en el diccionario lógico, habría punteros lógicos en la copia en cache y en la tabla tendría que incorporar traducciones para todos los punteros virtuales del haz, puesto que por todos es posible preguntar cuando se maneja el haz. También se tendría que recorrer todo el haz en el disco, saltando de un nodo rasgo a otro, accediendo al disco tantas veces como rasgos tenga el haz, para copiarlo en memoria. Hay que recorrer el haz saltando porque a priori no se sabe cuantos rasgos tiene un haz y entonces no se puede copiar de disco a memoria todo un haz como un bloque, con un único acceso al disco. Puesto que con la granularidad gruesa no reduzco el número de posiciones empleadas en la tabla de traducción, ni se ahorran accesos al disco, no supone ninguna mejora con respecto a la granularidad fina. Es más con las unidades pequeñas puede que me ahorre los accesos a disco de los nodos rasgo de un haz que no se utilizan frecuentemente.

4.5.2 Cache y dispersión de memoria

Cuando se quiere incorporar una parte del diccionario lógico a la cache se reserva memoria dinámica para ella, se copia allí y se ingresa en la tabla el par puntero lógico/puntero de memoria que apunta a la copia. Las copias están en memoria principal que se pide dinámicamente por lo cual la cache no ocupa un bloque monolítico y fijo, sino trozos dispersos. Esta fragmentación hace a nuestra cache más eficiente en el uso de memoria que si reservara un gran bloque de memoria y lo gestionara nuestro código privadamente. Es más eficiente porque si nuestra cache ocupa globalmente 50 KB, puede que el sistema no tenga un bloque libre de 50 KB, pero sí tenga tres bloques de 20 KB que la cache dispersa sí aprovecha pero que resultarían insuficientes

para una cache monolítica. Otra ventaja al pedir la memoria dinámicamente por cada unidad o bloque que se incorpora a la cache es que la gestión de la memoria se deja en manos del sistema operativo. Si hubieramos pedido un único bloque, necesariamente grande, en el que nuestro código situara las unidades de cache entonces la complejidad de manejar esa memoria recaería en nuestro código.

4.5.3 Tamaño de la cache

En cuanto al tamaño de la cache tenemos el que ocupa la tabla y el que ocupan las copias de las unidades del diccionario lógico. El tamaño de esas zonas no se controla rígidamente, sino que se regula a través de numero de posiciones de la tabla de traducción, que se deja como parámetro de funcionamiento. Cuantas más posiciones de tabla mayor será la zona de memoria ocupada por la cache. Grosso modo, posiciones de la tabla y memoria máxima ocupable por toda la cache están relacionadas a través de una constante de proporcionalidad:

$$mem.cache = mem.tabla + mem.dispersa \tag{4.2}$$

$$mem.tabla = (num.posic)(tam.posic) \tag{4.3}$$

$$mem.dispersa \approx (num.posic)(tam.medio.unidad) \tag{4.4}$$

$$mem.cache \approx (num.posic)(tam.posic + tam.medio.unidad) \tag{4.5}$$

La elección de ese parámetro depende de la cantidad de memoria disponible en la plataforma donde se está ejecutando el código. Siempre sera deseable ajustarlo al máximo que se pueda porque cuanto mayor sea la tabla mayor es el número de porciones del conjunto de haces que se acceden rápidamente porque residen en memoria, y entonces el diccionario responde más agilmente. El comportamiento de la velocidad del diccionario frente a distintos tamaños de cache se estudia con detalle en capítulo 5.

4.5.4 Cache no de escritura

Una cache de lectura-escritura desde luego sería posible pero no merecía la pena complicar el código para no ganar gran cosa. Sólo se escribe en el fichero de los haces en tiempo de compilación, y en ese proceso los accesos consisten

básicamente en ir escribiendo secuencialmente,no son nada repetitivos. La porción principal del tiempo manejando los haces se la lleva la escritura física en el disco y esto no se agiliza con la cache, sólo se aplaza.

4.5.5 Cache por conjuntos

Se ha preferido una cache asociativa por conjuntos porque ofrece mejores prestaciones en tiempo que una cache totalmente asociativa.

Una cache totalmente asociativa se puede ver como un caso extremo de asociativa por conjuntos en la que solo hay un único conjunto del tamaño de la cache entera. El comportamiento de las caches totalmente asociativas se deteriora a medida que su tamaño es mayor. La explicación es que para localizar la traducción de un puntero lógico en la tabla inicio una búsqueda lineal a lo largo de toda la tabla. Cuando el puntero lógico no está en la cache, no tiene traducción, la búsqueda se ha recorrido entera la tabla, consultando todas las posiciones, y eso conlleva tiempo. Sin embargo en las asociativas por conjuntos la búsqueda se restringe a un conjunto. Así cuando no tiene traducción se exploran todas las posiciones de ese conjunto, que siempre serán menos que las del total de la tabla. Cuando sí la tiene también se consultan menos posiciones porque el conjunto es más pequeño que la tabla entera. Por lo tanto con una cache con conjuntos, si un puntero está en la cache se encuentra antes su traducción, y si no está, también se descubre antes, porque sólo se busca en un conjunto, no en todos los de la tabla. La contrapartida de los conjuntos es que puede que habiendo posiciones libres en el global de la cache, un bloque tenga que residir en el disco porque su conjunto está repleto. Por lo tanto para acceder a ese bloque tenga que acudir al disco aun cuando hay sitio libre en la cache; en otros conjuntos que no son el suyo, claro. Es decir que con conjuntos aumento la probabilidad de fallo, de que un bloque no esté en la cache porque su conjunto está repleto. Ese aumento en la probabilidad de fallo se soluciona con un adecuado tamaño de los conjuntos, y con un mayor tamaño de la cache de manera que cada conjunto tenga que soportar menos carga potencial, menos posibles punteros que van a él. Esas dos variables, el número de posiciones de la cache y el tamaño de sus conjuntos, son los parámetros con los cuales se busca el compromiso entre las ventajas y desventajas de los conjuntos, tratando de minimizar el tiempo que se tarda en servir una petición de lectura.

4.5.6 Lectura de la cache

Cuando se quiere leer de un puntero lógico primero se comprueba si está en la tabla de traducción de la cache, es decir, si la unidad a la que apunta se ha llevado a memoria. Si lo está se sirve la petición desde memoria. Si no lo está se introduce la unidad a la que apunta en la cache y desde allí se sirve la lectura. Se incorpora a la cache cualquier porción del conjunto de haces que se solicite y no esté en ella, porque que se acceda una vez es ya indicio de que se va a acceder en un futuro inmediato. Para localizar un puntero lógico en la cache de conjuntos primero se calcula el conjunto dentro del cual puede estar. A cada puntero solo le corresponde un conjunto de entre todos los existentes. Es deseable que esta correspondencia reparta uniformemente el espacio de los punteros virtuales entre todos los conjuntos, de manera que no haya ningun conjunto con más posibles punteros que otro y todos soporte aproximadamente la misma carga potencial. Este reparto se ha hecho con una función pseudoaleatoria, que además cumple con el requisito de ser rápida. Esta velocidad es necesaria porque esta parte del código, la de los accesos a la cache se va a usar constantemente y cualquier optimización aquí tiene gran reflejo en la velocidad global. Una vez localizado el conjunto se explora linealmente si el puntero que nos han dado tiene traducción dentro de él. Esta búsqueda se aprovecha para seleccionar el puntero de fecha más vieja dentro del conjunto. Si se recorre todo el conjunto y no se encuentra traducción entonces la unidad requerida no está en la cache y el puntero que he seleccionado con la fecha más antigua se convierte en víctima. Obviamente la víctima siempre tiene que ser y será del mismo conjunto en el que se busca el puntero. Es conveniente aprovechar ese recorrido por el conjunto porque si no tendríamos que hacer dos pasadas: una para localizar el puntero y otra, cuando no estuviera, para elegir víctima; con lo que retardaríamos la ejecución.

El proceso entero de consulta cuando la unidad realmente está en la memoria de la cache lo ilustra la figura 4.6.

El proceso de consulta cuando la unidad sólo está en disco lo ilustra la figura 4.7.

Hay que recordar que cuando una unidad lógica con los punteros que pueda tener, por ejemplo un nodo rasgo, está en la cache, los punteros que tiene la copia de memoria siguen siendo punteros lógicos. No son punteros de memoria, aunque la zona a la que apunta esté luego en memoria. Se mantie-

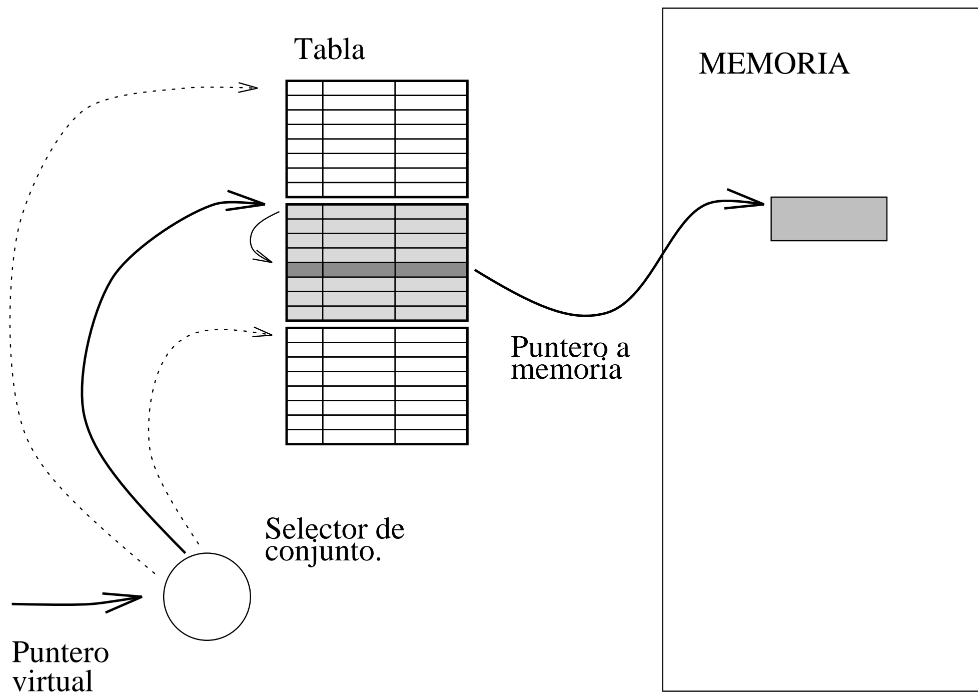

Figura 4.6: Consulta a una unidad que sí está en cache.

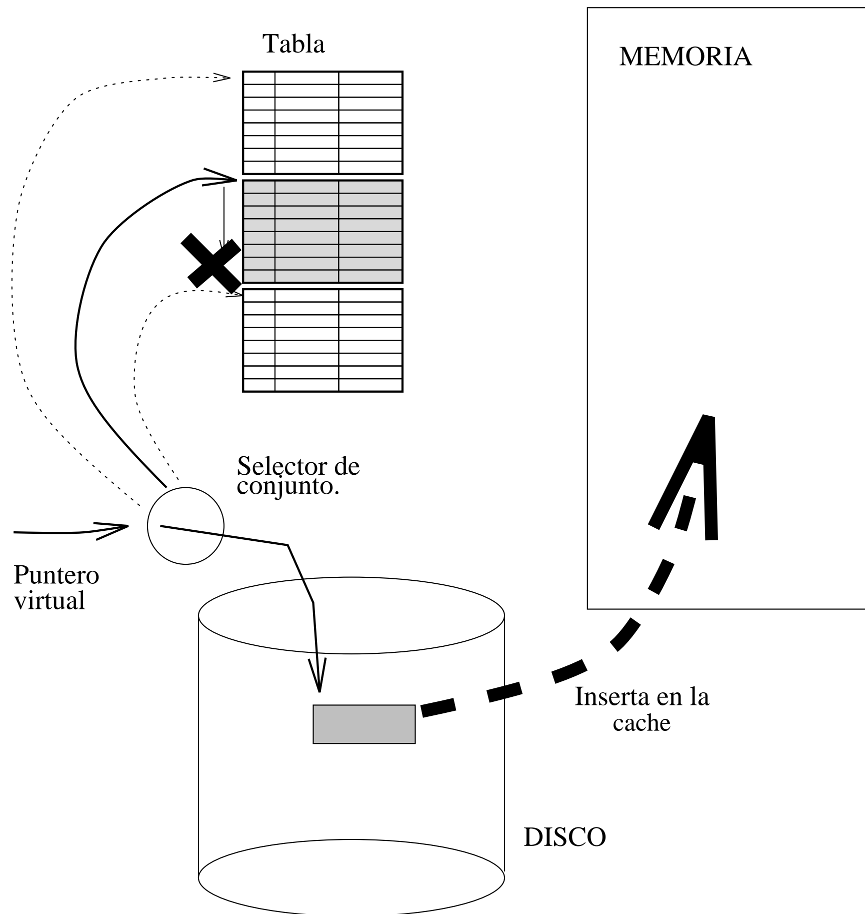

Figura 4.7: Consulta a una unidad que no está en cache.

ne siempre el manejo de las estructuras con punteros lógicos, por lo cual la manipulación se hace independiente de la ubicación final de las estructuras, que se resuelve en otro nivel.

4.5.7 Tiempo de carga.

Inicialmente la cache está vacía y tarda un tiempo en llenarse y llegar a un estado estable. Éste se caracteriza por ser una especie de equilibrio dinámico en el cual las unidades más frecuentemente accedidas ya están incorporadas, y sólo hay actualizaciones en las que se sustituye una unidad que se accede esporádicamente por otra. Estas actualizaciones se denominan *actualizaciones friccionales*. Son los propios accesos los que van llenando y configurando la cache. Este tiempo de estabilización viene a ser un coste fijo que hace que los primeros accesos al diccionario sean más lentos porque cargan la cache desde el disco. Pero este coste permite que los accesos siguientes sean velocísimos al servise desde la memoria de la cache. Si la aplicación hace muchas consultas, este coste inicial queda rápidamente diluído y el tiempo medio de acceso se acerca enseguida al de un acceso en memoria. No hemos encontrado modo alguno de eliminar este tiempo de carga de la cache. Parece que una manera de eliminarlo sería partir ya en el inicio de una cache configurada para los accesos más frecuentes. Sin embargo es un silogismo porque el tiempo que se tarda en *preconfigurarla*, en llenarla con esa configuración es aproximadamente el tiempo en que ella sola se configura adecuademente. Siempre correríamos el riesgo de que nuestra preconfiguración no fuera correcta, no fuera realmente la de las unidades más accedidas. Además complicaríamos el código para calcular esa preconfiguración, que habría que estimar y es función del texto real al que se enfrente el diccionario. En definitiva que lo menos complicado y más eficiente es dejar que la propia cache alcance su estado estable.

4.5.8 Algoritmo de sustitución.

Cuando quiero incorporar una unidad del conjunto de haces lógicos a la cache puede que en el conjunto correspondiente haya algún hueco libre y puede que no. Si hay un sitio vacío la incorporación se reduce simplemente a copiar la unidad en memoria e insertar el par puntero lógico- homólogo de memoria. Si no hay ningún hueco en la tabla de traducción hay que expulsar a algu-

na unidad existente para que la nueva ocupe su lugar. A esto se le llama elegir víctima y buena parte de la eficiencia de la cache reside en el criterio y algoritmo de elección de víctima porque ellos determinan quien se queda en la cache y quien no cuando esta se llena. Como es una cache asociativa por conjuntos tanto la búsqueda de un hueco como la elección de una victima a la que reemplazar se realizan dentro del conjunto asociado al puntero virtual por el que se pregunta, y no en toda la tabla. El algoritmo de sustitución que se ha preferido es el LRU, que escoge como víctima la unidad accedida menos recientemente. Para implementarlo nuestra cache lleva un reloj interno, y cada unidad que tiene lleva asociada la *fecha* en la que fue accedida por última vez. De este modo escoger víctima sólo es mirar cual de ellas tiene la fecha más antigua. Los bloque de uso frecuente entran en la cache muy pronto, la primera vez que se solicitan, y aunque estén en la cache desde hace mucho tiempo como son accedidos muy frecuentemente se refrescan continuamente y su *fecha* nunca envejece mucho. Por el contrario las unidades que se acceden esporádicamente entran en la cache cuando se consultan pero salen enseguida que se necesite su espacio porque su *fecha* envejece con prontitud. Este algoritmo tiende a tener en la cache las unidades más frecuentemente accedidas, así se sirve desde ella el mayor número posible de consultas, con lo que se minimiza el tiempo de consulta. Otro aspecto positivo de este algoritmo es que la cache se autoconfigura según las consultas que se le piden. Por esto el diccionario se adapta para responder rápidamente al texto con el que se enfrenta. Por ejemplo si el diccionario se utiliza para soportar el análisis de un texto, y en una parte de este se repite mucho la palabra *guerra*, entonces los formantes de *guerra* ingresarán en la cache y en ella permanecerán mientras se sigan consultando con frecuencia. Pero si ocurre que a partir de un punto la palabra *guerra* deja de aparecer en el texto, entonces sus formantes envejecerán y pronto dejarán su sitio en la cache a otras entradas de acceso más frecuente. Es una adaptación dinámica, más eficiente que una adaptación estática porque se amolda más estrechamente a las regularidades del texto.

4.5.9 Anomalía del reloj.

Como hemos visto la cache lleva un reloj interno para implementar el criterio LRU de sustitución de unidades. El reloj es simplemente un contador que se incrementa con cada acceso. Puesto que el tamaño de la variable *reloj*

es finito, habrá un momento en que el reloj *dé la vuelta*. Por ejemplo si el reloj sólo tiene dos dígitos va de 0 a 99, y al incrementarse para pasar a 100 vuelve a 0. Si esto no se soluciona, cuando el reloj diera la vuelta la cache llena dejaría de funcionar correctamente porque estaría plagada de accesos con fechas como *90*, *95*, *98* y los nuevos accesos *101*, *102* se datan como *01*, *02* y serían los más propicios a ser sustituidos. Es decir se sustituirían los más recientemente accedidos, los de uso más frecuente, que es justo lo opuesto a lo que pretendemos. Con ello el tiempo de respuesta se degradaría mucho porque habría que estar ingresando en la cache desde el disco continuamente. Esta es la anomalía del reloj. La solución que se ha tomado para evitar esta situación es poner todas las fechas de la cache a cero cuando el reloj dé la vuelta. De esta manera no sufre disfunción alguna y las unidades más frecuentes enseguida se refrescan.

4.5.10 Cache infinita.

La cache normal está pensada para la situación típica en la que no todo el conjunto de haces cabe en memoria. Entonces sólo los más frecuentemente accedidos son seleccionados para ocupar la poca memoria que haya. Para todos estos punteros la traducción de puntero virtual a dirección física en memoria se hace con la tabla. Si estamos en la ventajosa situación de una plataforma con mucha memoria principal y se quiere trabajar con todas las estructuras del diccionario en ella, sería absurdo mantener ese esquema de traducción de la cache porque hay otro más rápido. En este ventajoso caso podemos pedir un único bloque de memoria dinámica y volcar en él el contenido del fichero con los haces, para consultar los haces desde ese bloque de memoria. Así el puntero lógico 33, que en el disco señala al byte 33, se mapearía en memoria en la dirección 48+33 si 48 es la dirección de memoria donde empieza el bloque. Con esta ubicación la traducción queda reducida a sumar al enlace virtual la dirección base de la zona de memoria que se utiliza como espejo del disco. De este modo se consigue trabajar con el conjunto de los haces en memoria de un modo más veloz que si traducimos las direcciones con una tabla. Pero sólo se puede emplear cuando tenemos mucha memoria en la plataforma hardware, y todo el fichero con los haces nos cabe en ella.

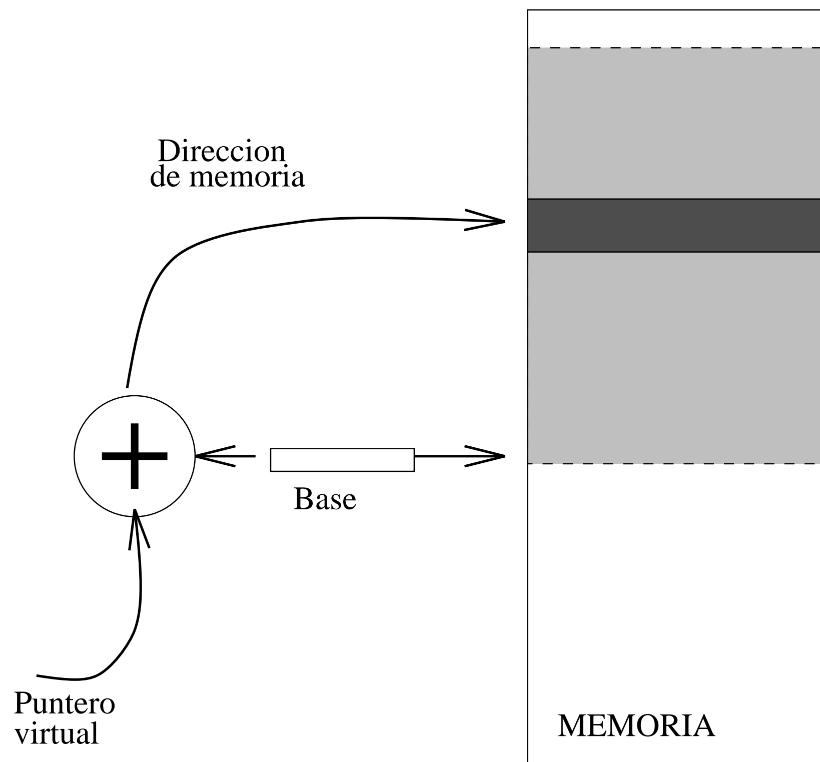

Figura 4.8: Lectura con cache infinita.

4.5.11 Traducción de punteros lógicos en consulta.

Resumiendo esta sección podemos decir que el en tiempo de consulta el módulo *disco.cc* tiene distintas posibilidades para acceder al contenido de un puntero lógico. Si no está en memoria acudirá la disco. Si está en memoria según la cache sea infinita o no la traducción del enlace virtual a una dirección de memoria se conseguirá aritméticamente o a través de una tabla.

4.6 BUFFER DE COMPILACIÓN

Para agilizar las consultas a los haces se emplea la cache. Pero también podemos utilizar la memoria en tiempo de compilación para acelerar este procesamiento. Durante la compilación el acceso a los haces es poco reptitivo y la operación fundamental consiste en escribir casi secuencialemente los haces en el disco. Una manera de acelerar este procesamiento con memoria es utilizarla como buffer. Con ello las escrituras son siempre en el buffer. En él se va escribiendo hasta que se llena y entonces lo vuelco al disco. De este modo minimizo el numero de accesos al disco físico que ralentizan la ejecución, como no hay que posicionar tantas veces las cabezas lectoras del disco, la escritura global es más rápida. En vez de un acceso por cada unidad que quiero escribir tenemos un acceso por cada buffer lleno, y en cada buffer caben muchas unidades del diccionario. Además si en tiempo de compilación se plantea una lectura esta suele ser en posiciones muy cercanas a la última escrita, con lo cual también caen dentro del buffer y se leen desde él en lugar de bajar hasta el disco físico.

Al igual que con la cache lo que tenemos en el buffer es una parte del conjunto lógico de los haces, esto es, los punteros que haya son punteros virtuales. Si se quiere acceder a un puntero lógico en tiempo de compilación hay dos posibilidades: que caiga dentro del buffer o en el disco físico. Caerá dentro del buffer si el puntero virtual es mayor que la última posición realmente escrita en el disco. Cuando sea menor, cae fuera del buffer, y en ese caso apunta a una zona que realmente está en el disco, por lo que se interpreta el puntero lógico como dirección dentro del disco y se accede a él. Si cae dentro del buffer accedo a la posición equivalente de memoria. Esta dirección equivalente se calcula aritméticamente de un modo muy rápido:

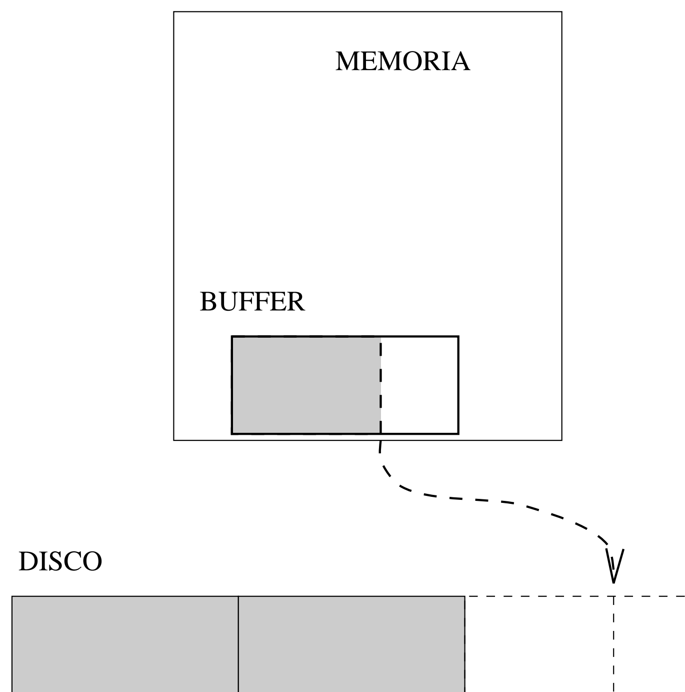

Figura 4.9: Buffer de compilación.

$$dir.memoria = base.buffer + incremento \quad (4.6)$$
$$incremento = puntero.logico - ultima.posicion.real.disco \quad (4.7)$$

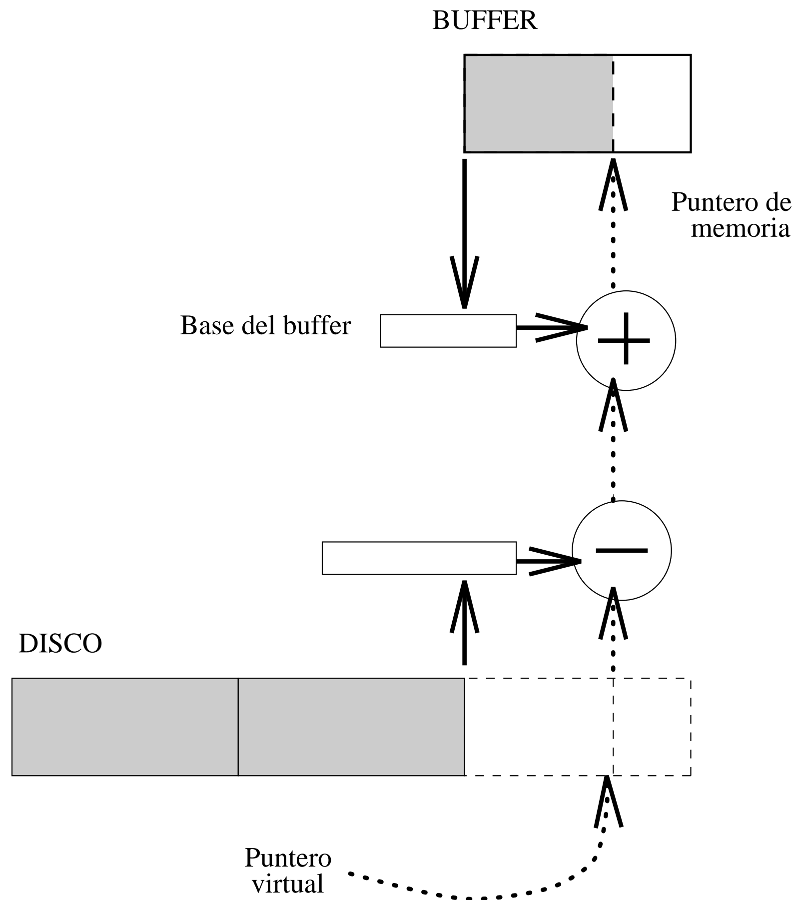

Figura 4.10: Acceso a un puntero que está en el buffer.

El direccionamiento que ilustra la figura 4.10 es válido tanto para accesos de lectura como de escritura porque el buffer *tapa* al disco en ambas operaciones.

Al calcularse tan rápidamente la equivalencia en memoria los accesos al buffer son realmente veloces. El contenido del buffer se vuelca al disco cada

vez que se llena o cuando se acabó de compilar la base léxica. Se vuelca de una pieza, porque en el buffer tenemos diccionario lógico.

El comportamiento del tiempo de compilación en función del tamaño del buffer se ha estudiado para distintas plataformas hardware, y se analiza con detalle en el capítulo 5.

La filosofía de aceleración de los haces con memoria es muy distinta en la compilación que en la consulta porque los accesos son totalmente distintos. Por ejemplo plantear un buffer en consulta no sería nada eficiente por ser los accesos muy aleatorios en cuanto a la posición leída.

Capítulo 5

ANÁLISIS DE PRESTACIONES

5.1 INTRODUCCIÓN

Este capítulo lo dedicaremos a exponer las diversas pruebas sobre hardware real que se han realizado a lo largo de todo el proyecto. Por un lado estas pruebas tienen un objetivo descriptivo porque describen cómo se comporta el sistema diccionario que se ha construido. Estas prestaciones determinaran en buena medida la utilidad del diccionario y el tipo de aplicaciones para las que resulta satisfactorio. Por otro lado algunas pruebas tienen una finalidad justificativa porque comparan el funcionamiento con distintas opciones de implementación en diversas cuestiones, por lo cual justifican la elegida como la más conveniente. Por ejemplo se justifica la elección de caches asociativas por conjuntos en lugar de totalmente asociativas o de correspondencia directa. Igualmente se justifica la decisión de incorporar la función de segmentación dentro de las ofrecidas por los módulos que manejan el trie. Además en este capítulo se comenta la cuestión de la portabilidad que era uno de los objetivos del proyecto, y los distintos problemas que han surgido hasta conseguirla.

Las pruebas se han realizado en dos sistemas hardware distintos. La primera plataforma es una estación *Sun Sparc10*, con 32 MBytes de memoria RAM y sistema operativo Unix (SunOs 4.1.3). El compilador que se ha empleado aquí es el *gcc* (versión 2.6.3) de GNU. La segunda plataforma es

una arquitectura PC, con procesador 486DX a 33 MHz, con 8 MBytes de memoria RAM. El compilador empleado en este sistema es el *djgpp* de GNU.

5.1.1 Módulo de pruebas.

Para efectuar las pruebas se ha desarrollado un módulo **probador**, que hemos llamado `tester.c`. Este módulo permite consultar el contenido de un diccionario objeto, es decir, los rasgos asociados a las entradas del diccionario y verlos por pantalla. También permite calcular y ver las distintas *segmentaciones de existencia* de cualquier palabra. Además permite trabajar con las dos interfaces del diccionario, la que accede directamente a las estructuras originales y la que trabaja sobre estructuras copia en memoria.

Como muestra todos los resultados de las consultas y segmentaciones por pantalla, este módulo se ha utilizado profusamente para comprobar si funcionaban o no determinadas partes del código. Por ejemplo para ver si ambas interfaces de acceso al diccionario funcionaban correctamente. Básicamente el `tester.c` realiza lo que cualquier aplicación que pretenda usar la base léxica debe hacer: cargar el diccionario y consultarlo a través de las interfaces que éste ofrece.

El módulo probador permite la consulta o la segmentación de palabras reunidas en fichero, lo que resulta muy útil para medir tiempos cuando se consultan o segmentan un elevado número de palabras.

El `tester.c` también se puede utilizar en modo interactivo. El programa presenta un menú para interactuar con el usuario por consola y que sea éste el que dirija las consultas que se requieren del diccionario. En la tabla 5.1 se muestra el **menú interactivo de pruebas** que esté módulo presenta al usuario.

La opción 3 en el ejemplo de la tabla 5.1 sirve para cambiar de interfaz, de la que trabaja sobre estructuras originales a la que lo hace desde estructuras copia en memoria, y viceversa.

La opción 4 muestra la tabla del diccionario de datos, es decir, todos los nombres reconocidos y su código numérico correspondiente.

El programa `tester` se invoca con varios parámetros que dependen del uso que se quiera de él. Hay dos parámetros que se necesitan en todos los casos; son los que configuran la cache del diccionario: tamaño de la cache y tamaño de los conjuntos. Si se quiere el menú interactivo basta especificar `consola` como tercer parámetro. Por ejemplo `tester 1000 4 consola`

```
Menu del tester
(0) = fin
(1) = Segmenta on line
(2) = consulta on line
(3) = cambiar a interfaz de copia en memoria.
(4) = diccionario de datos.
Opcion?  :
```

Tabla 5.1: Menú que el módulo de pruebas presenta al usuario.

habilita una cache de 1000 posiciones, con conjuntos de 4, y presenta la interfaz interactiva como la de la tabla 5.1 al usuario.

Si se quiere segmentar un conjunto de palabras agrupadas en un fichero entonces se indica `segmentar` como tercer parámetro; en el cuarto se especifica si se quiere segmentar con el segmentador interno (`int`) que ofrece el diccionario sobre el trie o con otro segmentador que se ofrece, que no aprovecha las ventajas del trie (`ext`); y en el quinto se señala el nombre del fichero con las palabras a segmentar. Por ejemplo `tester 1000 4 segmentar int file1` segmentaría con el segmentador del diccionario todas las entradas del fichero `file1`.

Si se quiere consultar, ver los haces de rasgos, de un conjunto de entradas agrupadas en un fichero entonces se indica `consultar` como tercer parámetro; como cuarto se especifica si queremos trabajar con la interfaz sobre estructuras originales (`disco`) o con la interfaz sobre estructuras copia en memoria (`mem`); en el quinto se señala el nombre del fichero con las entradas a consultar. Por ejemplo `tester 1000 4 consultar disco file2` muestra por pantalla los haces con los rasgos de las entradas en `file2` trabajando sobre las estructuras originales del diccionario.

5.2 PORTABILIDAD

Uno de los objetivos de este proyecto era conseguir que el código del diccionario realizado fuera portable a muchas plataformas distintas para que pudiera utilizarse en distintos tipos de máquina sin problemas. Para ello se deci-

dió escribir todos los módulos y programas fuente en *lenguaje C según el estándar ANSI*, más brevemente, en **ansi C**. Este lenguaje es estándar y por ello la mayoría de los compiladores son capaces de generar código objeto y ejecutable desde él.

El objetivo de la portabilidad se ha conseguido en buena medida, y el sistema diccionario se ha ejecutado sin problemas en máquinas tan distintas como una Sun SparcStation 10, una estación de trabajo Hewlett-Packard Apollo 9000, y un PC 486DX. Conseguir que funcionara en las estaciones de trabajo no ha sido difícil, pero portarlo a un PC con sistema operativo MSDOS ha sido más complicado. La principal dificultad radicaba en la gestión de memoria que realiza MSDOS y el uso extensivo que de ella requiere el diccionario.

Como hemos explicado en el capítulo 4 resulta muy conveniente para la rapidez del sistema trabajar con el trie en memoria, y si es posible también con el conjunto de haces. Para la base léxica fuente de 4'4 MB con la que hemos trabajado se genera un trie de 1'5 MB y un fichero con los haces de 8'5 MB, que como hemos dicho será deseable situar en memoria.

En las estaciones de trabajo, el sistema operativo es Unix y maneja una memoria virtual de 4 GB, que se explota según un modelo lineal. Es decir, los punteros de memoria que entregan las funciones estándar ansi C de asignación de memoria (*malloc, realloc*) son punteros de 32 bits que direccionan linealmente un espacio de 4 GB que es la **memoria virtual**. Por lo tanto los programas de usuario direccionan sobre la memoria virtual. En realidad no se tienen 4 GB de memoria física en la arquitectura, sino mucho menos, por ejemplo 32 MB. El sistema operativo se encarga de situar en esa memoria real las partes de memoria virtual que se necesiten, y de realizar la adecuada traducción entre punteros virtuales y direcciones de memoria física.

En las estaciones de trabajo si nuestro código quiere situar el trie de 1'5 MB en memoria el sistema operativo lo permite, reservándole 1'5 MB de memoria (virtual). Del mismo modo tampoco hay problema si se quieren situar los 8'5 MB de los haces en memoria. Probablemente no estén enteramente en memoria física porque el sistema operativo, mediante mecanismos de paginación y/o segmentación sólo tendrá en memoria real las partes más utilizadas de la memoria virtual. En cualquier caso ésta es una cuestión de eficiencia de la arquitectura que es transparente para los programas de aplicación, los cuales se limitan a manejar los punteros de memoria virtual.

En el PC bajo MSDOS el modelo de memoria es más enrevesado. Por

razones de compatibilidad con procesadores antiguos (8086) la memoria no se explota linealmente, sino a través de bloques de 64 KB que se llaman *segmentos*. Las direcciones son de 32 bits pero se dividen en dos componentes de 16 bits: el primero es el índice del segmento y el segundo es el *desplazamiento*, un índice que puede recorrer linealmente los 64 KB de ese segmento. Combinando ambos campos se obtiene la dirección de la memoria real desde los 32 bits. Por razones de compatibilidad con procesadores que sólo empleaban 20 bits para las direcciones de memoria sólo se pueden direccionar $2^{20} = 1$ MB, y siempre bajo el esquema segmento-desplazamiento. Este es el funcionamiento cuando se trabaja con el procesador en **modo real** (es decir, compatible con 8086).

Siguiendo este complicado manejo de memoria los compiladores tradicionales de C para PC, como por ejemplo el BorlandC++ 3.1, ofrecen a los programas de usuario como máximo 1 MB de memoria dinámica a través de las funciones estándar (*malloc, realloc*). ¡¡ Aunque el PC disponga de 8 MB de memoria RAM real !!. El diccionario realizado puede funcionar con sólo ese megabyte de memoria, pero lo hace trabajando directamente sobre disco, lo que provoca que sea muy lento. Para que el diccionario ofrezca tiempos de respuesta admisibles es necesario que pueda utilizar más memoria. Pero cuando el procesador del PC se usa en *modo real* resulta imposible utilizar más de 1 MB.

La solución adoptada para ejecutar el código del diccionario en una arquitectura PC ha sido utilizar un **DOS extender**. EL *DOS extender* ejecuta programas en el PC utilizando al procesador en **modo protegido** (no compatible con 8086) y aprovechando de modo lineal toda la capacidad de direccionamiento del procesador. Los compiladores usuales como BorlandC++ 3.1 generan código para *modo real*, no para *modo protegido*, que es el que es capaz de correr el *DOS extender*. Ha sido necesario buscar un compilador que generara código para *modo protegido*. El compilador utilizado es el **djgpp** de GNU. Gracias a él y al *DOS extender* las aplicaciones como nuestro diccionario pueden utilizar linealmente más del megabyte que ofrecían los compiladores para modo real, y aprovechan toda la memoria RAM existente en la máquina donde se ejecutan.

Otro compilador que genera código para PC ejecutable en modo protegido y que direcciona más de 1 MB, es el BorlandC++ 4.5 con la utilidad PowerPack instalada. También hemos compilado con él nuestro diccionario y lo hemos probado satisfactoriamente.

La portabilidad se ha conseguido en un grado elevado: el código fuente empleado para estos compiladores y plataformas es el mismo en todos los casos. En las diferentes combinaciones probadas el diccionario funciona adecuadamente, con la lógica diferencia de velocidad debida a trabajar en distintas arquitecturas.

Hay un aspecto en los programas fuente que difiere de utilizarlos en la Sun SparcStation a ejecutarlos en la plataforma PC: el modo de apertura de los ficheros. Como se explicó anteriormente el diccionario maneja ficheros como si fueran espejos de memoria, para lo cual necesita abrir esos ficheros en *modo binario*, no en *modo carácter*. Ocurre que para el compilador *gcc* empleado en la estación Sun, esto se especifica con los modos de apertura "`r`' (para sólo lectura), "`w+`' (para escritura) y "`a+`' (para actualización) como parámetros en la función estándar *fopen*. Sin embargo para el compilador *djgpp* sobre PC, se especifica añadiendo una `b` de binario a los modos anteriores. Es decir con "`rb`', "`wb+`' y "`ab+`' respectivamente. Esta diferencia se ha salvado con una directiva de compilación que define la constante UNIX o la constante PC, y que escoge los parámetros adecuados de *fopen* para manejar los ficheros en modo binario según estemos en una plataforma u otra. Por defecto se abren al estilo gcc. Al incluir esta directiva no hay que retocar para nada el código fuente de una plataforma a otra, sólo habrá que compilarlo con la directiva correspondiente.

5.3 EFECTO DEL BUFFER

Como vimos en el capítulo 4 el buffer entra en funcionamiento en la compilación de la base léxica expandida, con objeto de acelerarla. Para evitar el manejo directamente sobre el disco, que es lento, se crea una zona de memoria como espejo de una parte del fichero que contiene los haces de manera que todas las lecturas y escrituras sobre esa parte de diccionario lógico se realizan rápidamente sobre la memoria de ese buffer. Éste se vuelca periódicamente, cuando se llena, al disco, que es la ubicación final del diccionario objeto. Con el buffer se minimiza el número de accesos reales al disco, que son muy costosos en tiempo y ralentizan la ejecución. El buffer no tiene ningún efecto cara a la consulta del diccionario, sólo influye en el tiempo de compilación.

En esta sección vamos a estudiar el comportamiento del tiempo que tarda en compilarse una base léxica expandida de 4'3 MB en función del tamaño

de ese buffer.

Pruebas en SunSparc10.

Compilando esa base expandida de 4'3 MB en una Sun SparcStation10 se ha conseguido la función de la figura 5.1.

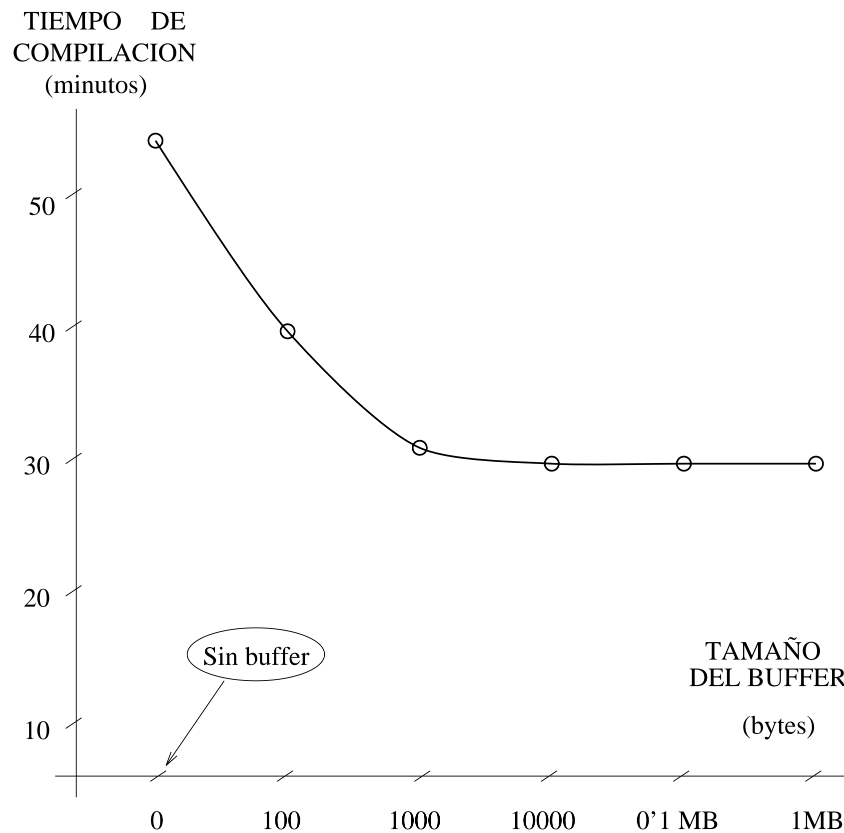

Figura 5.1: Tiempo de compilación en SparcStation en función del tamaño del buffer.

Gráficas totalmente semejantes se obtuvieron compilando bases léxicas más pequeñas, pero con tiempos de compilación menores. En la figura 5.1 se nota claramente una reducción sustanciosa del tiempo de compilación al utilizar buffer. Se pasa de 55 minutos que se tarda en compilar sin ningún

buffer, a 30 minutos que lleva la compilación con uno de 10000 bytes. El empleo del buffer supone una aceleración del 45 %. Es decir, el buffer es realmente beneficioso.

En cuanto al rendimiento de los bytes del buffer se observa claramente una tendencia decreciente, hasta anularse cerca de tamaños de 1000 bytes. Es decir, a medida que el buffer es mayor, iguales incrementos en su tamaño provocan menores descensos en el tiempo de compilación, que prácticamente se estanca a partir de buffers de 1000 bytes. Este es pues el tamaño óptimo recomendado para esta plataforma, porque consigue toda la ganancia en tiempo del empleo del buffer con un coste de memoria, su propio tamaño, mínimo. La figura 5.2 ilustra este hecho. En sus abcisas está la longitud del buffer y en ordenadas el descenso en el tiempo conseguido con el último incremento del buffer. A partir de 1000 bytes los descensos no son significativos.

Este comportamiento es comprensible porque ya con ese tamaño de 1000 bytes se reduce sustancialmente el número de accesos reales a disco; no escribiendo en él para compilar cada rasgo por ejemplo. De ahí a tamaños mayores nos movemos en un número de accesos a disco relativamente pequeño, por lo que no hay grandes diferencias en tiempo. Es decir si con 1 MB sólo accedemos al disco 9 veces, con 10000 bytes accedemos 900, que no es significativo frente a los miles de veces que necesitamos si trabajamos sin buffer. La mayor reducción de accesos se consigue por el mero hecho de activar un buffer y no trabajar sobre disco directamente. Una vez alcanzado un cierto tamaño las reducciones de tiempo que se consiguen incrementando el tamaño del buffer no son significativas.

Conviene recordar que el sistema operativo SunOS 4.1.3 de la estación Sun ya provee mecanismos de buffer para agilizar las operaciones de entrada-salida (E/S). El buffer que hemos realizado con nuestro código trabaja por encima de los buffer E/S del sistema operativo. Todos ellos conjuntamente contribuyen a agilizar los accesos al disco. Es de destacar que cuando no se activa nuestro buffer, los del SunOS 4.1.3 sí actúan, porque lo hacen siempre. Además es resaltable el hecho de que nuestro buffer se reduzca el tiempo de compilación aún más de lo que disminuyen los buffers del sistema operativo. Esto es así porque nuestro buffer está especialmente adaptado, en uso y en tamaño, al empleo que el compilador hace del disco, mientras que los del sistema operativo son más generales, se adaptan peor.

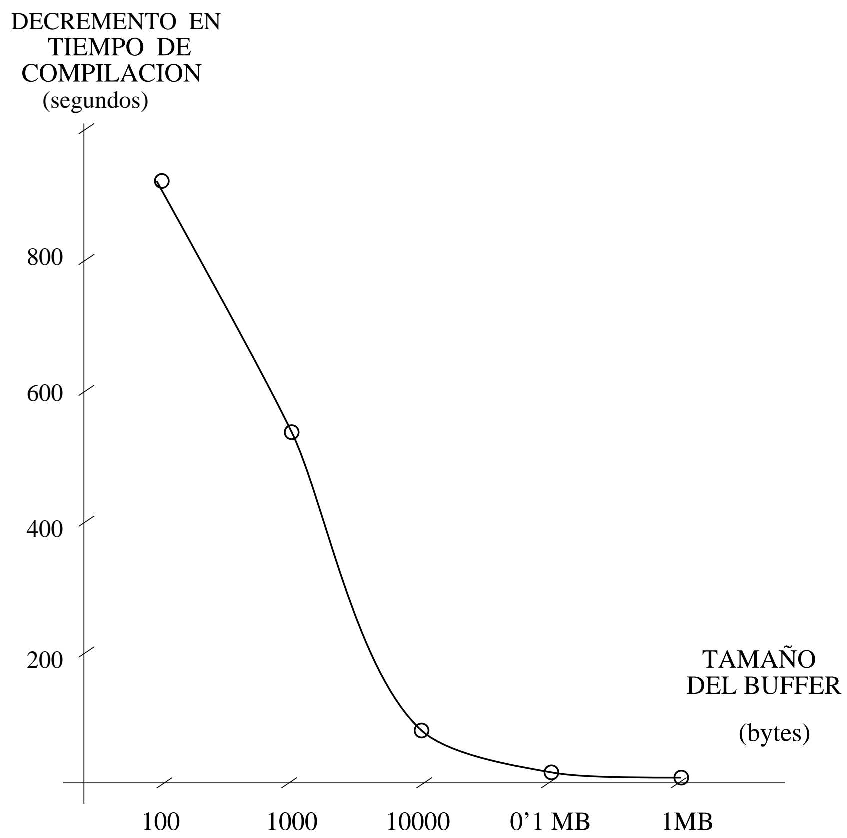

Figura 5.2: Rendimiento del buffer en SparcStation.

Pruebas en PC con djgpp.

Compilando la base expandida de 4'3 MB en un PC con un *DOS extender* y código generado por el compilador *djgpp* se ha conseguido la función de la figura 5.3. Esta función expresa los diferentes tiempos de compilación obtenidos con distintos tamaños de buffer.

Como se observa es totalmente análoga a la gráfica obtenida en la plataforma Sun, pero con valores absolutos más altos y una reducción relativa del tiempo de compilación al usar buffer más significativa en PC. En la Sun, compilar la base de 4'3 MB sin buffer lleva unos 54 minutos, mientras que en PC tarda 211 minutos. Compilarla con un buffer de 100000 bytes lleva 30 minutos en la estación mientras que 68 minutos en PC. La reducción de tiempo conseguida con el buffer en la SparcStation es del 45 %, mientras que en el PC es del 68 %.

Estas diferencias se pueden atribuir a la velocidad del procesador (el 486DX del PC es más lento que el de la estación), y a que MSDOS no provee buffers de E/S que sí proporciona el SunOS 4.1.3. En PC cuando nuestro buffer no existe, las operaciones se realizan directamente sobre disco, y son muy lentas. Por ello cuando se activa el buffer de compilación la reducción de tiempo de compilación es más significativa en el PC que en la estación Sun. En la SparcStation 10 incluso cuando no tenemos nuestro buffer, sí actúan los del sistema operativo, haciendo menos significativa la mejora del buffer de compilación.

En resumen, el coste de incorporar el buffer es nulo (salvo tener un código más complejo) y ofrece una reducción sustanciosa del tiempo de compilación con buffers relativamente pequeños. Se consiguen reducciones del 46 % en la estación Sun y del 67 % en el PC, con buffers de apenas 2 o 3 KB.

Observando los mejores tiempos de compilación conseguidos se refuerza la necesidad de separar los procesos de compilación de la base léxica expandida y su carga. Mientras que cargar el diccionario puede llevar 2 o 3 segundos, incluso menos si sólo se carga el trie, compilar la base expandida lleva más de 30 minutos. Este tiempo imposibilitaría cualquier uso del diccionario *en tiempo real* si hubiera que crear las estructuras del diccionario cada vez que éste se va a utilizar.

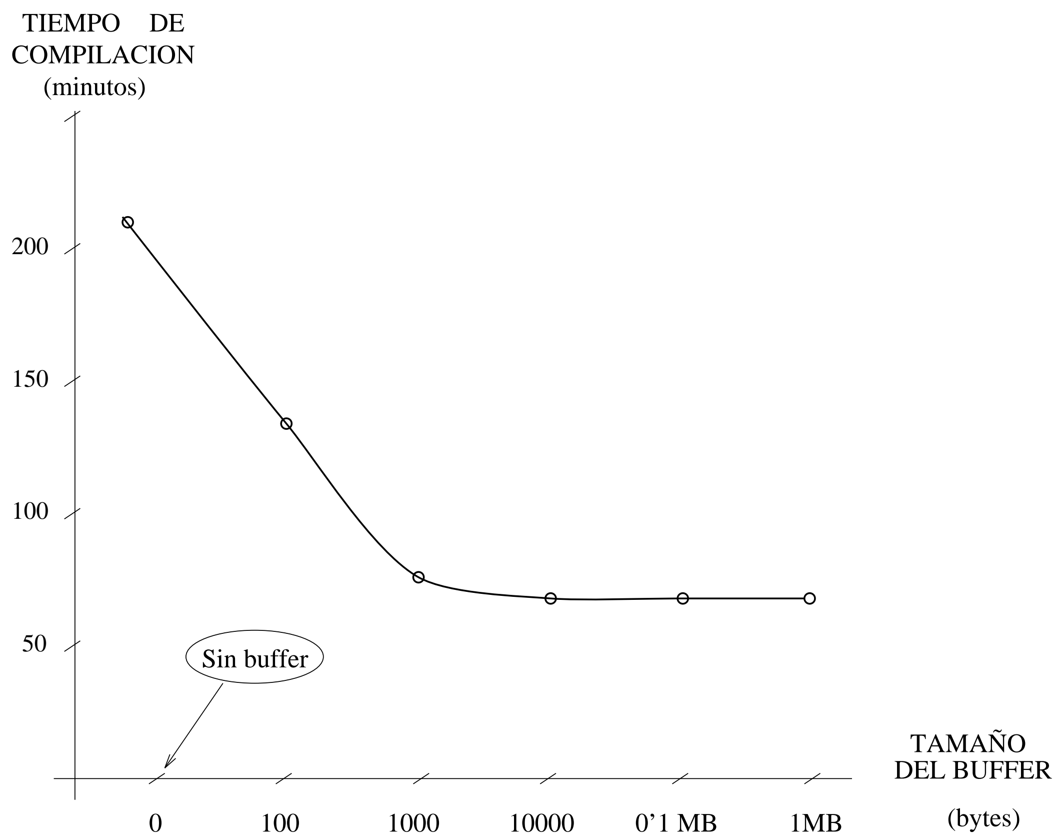

Figura 5.3: Tiempo de compilación en PC en función del tamaño del buffer.

5.4 EFECTO DE LA CACHE DE LECTURA

.

Tal como se describió en el capítulo 4 se ha desarrollado un sistema cache para acelerar las consultas que los programas de aplicación realizan a nuestro diccionario. Básicamente lo que consigue es tener en memoria los haces de rasgos más frecuentemente accedidos, y servir desde ella las consultas que necesitan esos rasgos. Como no se sirven desde disco sino desde memoria los accesos son mucho más rápidos. Puesto que se tienen los rasgos más consultados la ganancia global es mayor que si tuvieramos rasgos escogidos aleatoriamente o con otro criterio que no atendiera a la frecuencia de uso.

En esta sección vamos a estudiar el comportamiento del tiempo de consulta cuando se trabaja con todo el conjunto de haces de rasgos en disco, o todo en memoria o en las distintas alternativas que se ofrecen de empleo de la cache.

El sistema cache que se ha desarrollado acepta dos parámetros que condicionan su funcionamiento y le permiten adaptarse a la cantidad de memoria RAM que la plataforma hardware presente. Uno de los parámetros es el **tamaño de cache**, que es el número de posiciones de la tabla de traducción que emplea la cache. El segundo es el **tamaño de los conjuntos**, que es el número de posiciones de esa tabla por cada conjunto.

Para las gráficas de esta sección se han medido los tiempos que tarda el diccionario en segmentar un conjunto de 2000 palabras representativo del uso del castellano. Este conjunto se ha escogido aleatoriamente de un corpus con decenas de artículos periodísticos reales. En él aparecen por lo tanto las palabras más utilizadas en el habla castellana, y en la proporción adecuada a su frecuencia de uso. Por ello este conjunto es una muestra representativa de las consultas que se le hacen al diccionario cuando se analiza texto real en castellano. Por otro lado la segmentación es la consulta típica que le hace el analizador morfológico al diccionario, el cálculo de las *segmentaciones de existencia* de una palabra. Con esto se ha conseguido medir tiempos en unas condiciones muy cercanas a las típicas en las que se prevee que se utilizará nuestro diccionario. Estas mediciones se toman como representativas del tiempo medio que tarda en diccionario en servir una consulta típica, es decir, de su *tiempo de consulta*.

Pruebas en Sun SparcStation.

En la máquina SparcStation 10 se han obtenido los tiempos de la figura 5.4, donde el tamaño de los conjuntos aparece en el eje de abcisas y el número de posiciones de la cache como parámetro.

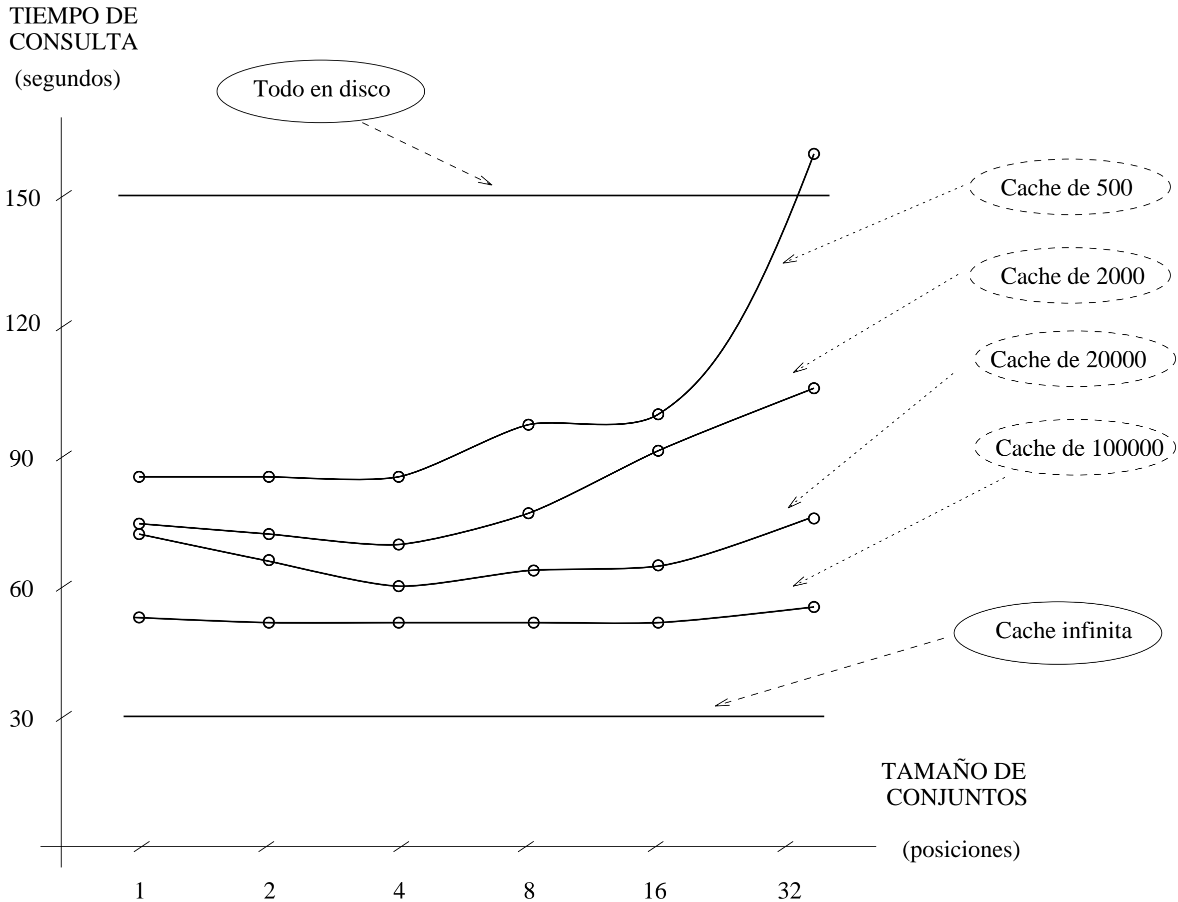

Figura 5.4: Tiempo de consulta en SparcStation frente a tamaño de conjuntos.

Como era de esperar las consultas son más lentas cuando todos los haces de rasgos se tienen en disco, es decir, cuando tengo una **cache nula**, tardando 150 segundos en segmentar las 2000 entradas. Conviene recordar los buffers de E/S que provee el sistema operativo de la estación. Estos provocan que,

aunque nuestro código no se ocupe de ello, partes del fichero con los haces residan en memoria, en esos buffers del sistema operativo, y desde ellos se respondan algunas consultas. Es decir, gracias SunOS 4.1.3 el "acceso al disco' no es tan lento.

De todas las posibles configuraciones el diccionario responde más rápidamente con la opción de **cache infinita**, es decir, cuando todo el fichero con los haces está guardado en memoria. En este caso sólo tarda 28 segundos en segmentar las 2000 entradas. Cinco veces más rápido que sin ninguna cache, con todo en disco.

Conviene recordar que el sistema operativo de la SparcStation incorpora la paginación y segmentación para soportar la memoria virtual con la memoria real disponible. Esto quiere decir que si el fichero con los haces ocupa por ejemplo 8'5 MB, aunque tenga enteramente todos los haces en memoria virtual, no todos estarán en memoria real, que es la que responde velozmente a las consultas. Este hecho y los buffers de E/S hacen que las diferencias entre trabajar con todos los haces desde el disco y trabajar con todos en memoria, no sean mayores.

Entre medias de estas dos opciones extremas se encuentra la **cache finita**, que incorporará sólo algunas partes del conjunto de haces a memoria manteniendo las menos accedidas en disco. Como era de esperar los tiempos que tarda con *cache finita* en segmentar las 2000 palabras se encuentran entre los tiempos extremos de *cache nula* y *cache infinita*. La cache finita tiene todo un abanico de posibilidades de funcionamiento, que se escogen con los dos parámetros que antes comentamos.

En cuanto al comportamiento con el tamaño de los conjuntos se observa en la figura 5.4 que *conjuntos mayores retardan más la consulta*. Tal como se explicó en el capítulo 4 la razón es que las búsquedas dentro de los conjuntos son lineales y exhaustivas; si los conjuntos son grandes las búsquedas son lentas. Como muestra la figura 5.4 el comportamiento permanece aproximandamente constante para conjuntos de 1 a 4 posiciones, deteriorándose cada vez a medida que se toman conjuntos mayores. Esta degradación es más notable para las caches pequeñas, en las caches grandes los mismos tiempos aparecen para conjuntos aún mayores. Cuando la cache es grande hay muchas posiciones y como se segmenta un número relativamente pequeño de palabras, entonces rara vez varios punteros coinciden en el mismo conjunto. Por ello las búsquedas en los conjuntos encuentran su puntero en las primeras posiciones, las únicas ocupadas. Por lo tanto son búsquedas rápidas.

En los tiempos de la figura 5.4 influye el hecho de que se ha segmentado un número finito, pequeño, de palabras (2000), con lo cual las caches grandes tienen un factor de ocupación bajo. Con un mayor número de vocablos las caches grandes se hubieran llenado más y los problemas de usar conjuntos largos en ellas se hubieran manifestado más claramente. Del análisis de esta figura se desprende que conjuntos de 3 y 4 posiciones se muestran como los más beneficiosos.

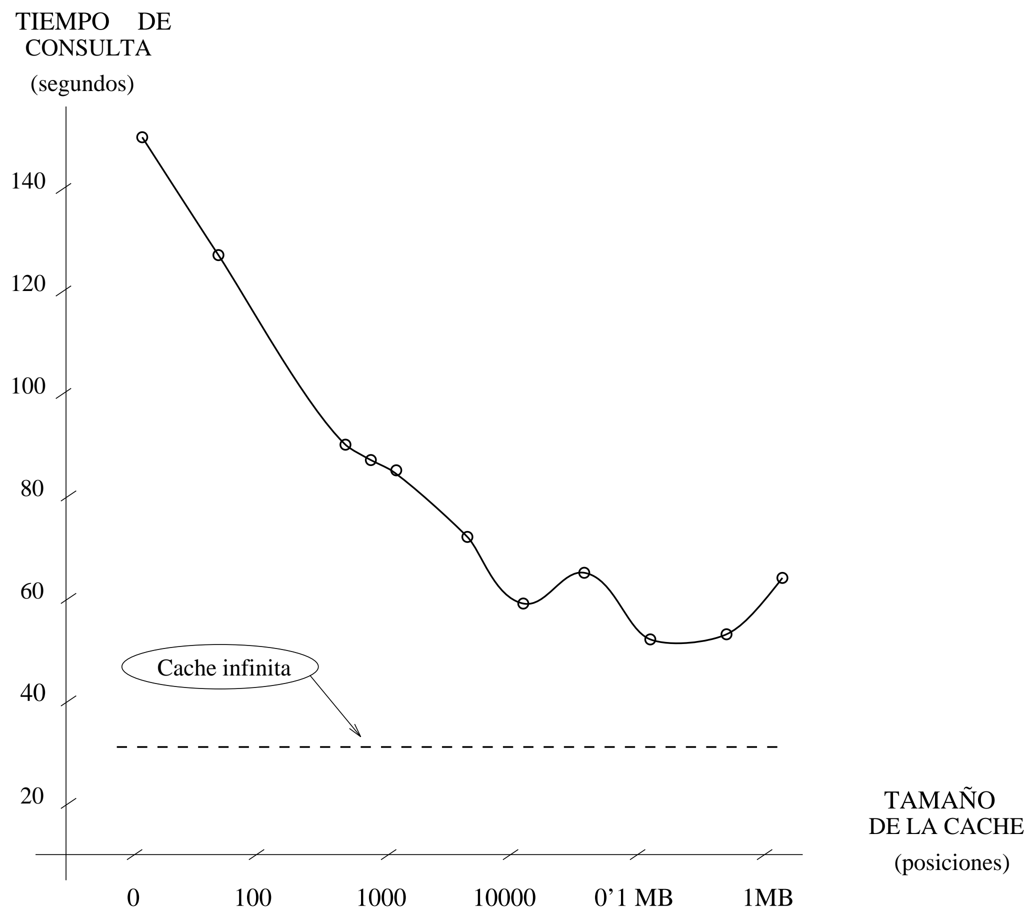

Figura 5.5: Tiempo de consulta en SparcStation en función del tamaño de cache.

La figura 5.5 muestra los tiempos de consulta medidos manteniendo los

conjuntos de 2 posiciones y variando el tamaño de la cache. El comportamiento es el que razonablemente cabía esperar: a tamaños mayores se obtienen menores tiempos de consulta, porque hay más consultas que se sirven desde memoria al estar más partes del diccionario objeto en ella. Por lo tanto cuando se utilice nuestro diccionario será conveniente hacerlo con el mayor tamaño de cache que permita la arquitectura.

En la figura 5.5 también se puede observar el *rendimiento decreciente* del tamaño de la cache. Es decir a medida que la cache es mayor un mismo incremento en su tamaño provoca menor descenso en el tiempo de consulta. Este comportamiento es comprensible porque con caches pequeñas los rasgos que hay en la cache son los más frecuentemente accedidos. A medida que aumenta de tamaño hay sitio para rasgos que se acceden menos frecuentemente y cuya incorporación a la cache provoca por tanto menores descensos del tiempo de consulta. Siguiendo este razonamiento el tiempo de consulta se estabiliza cuando en la cache caben todos los rasgos consultados. En este caso caches mayores no acelerán más la consulta porque igualmente se trabaja con todos los rasgos en la cache. En la figura 5.5 ese estancamiento se observa a partir de 100000 posiciones. Los pequeños altibajos en esa zona de la gráfica se deben al distinto comportamiento de la paginación y segmentación del sistema operativo.

Aun cuando se tienen caches tan grandes en las que caben con holgura todos los rasgos accedidos, como de 0'2 MB o 1 MB para nuestras 2000 palabras, el tiempo de consulta no logra bajar hasta el tiempo con *cache infinita.* Esto se puede explicar porque el direccionamiento en la situación de cache infinita no se sigue el esquema de traducción de punteros con tabla, que se sigue para caches finitas. En su lugar la traducción se consigue aritméticamente, que resulta mucho más rápida. Además con cache infinita el conjunto de haces se vuelca a memoria con un sólo acceso a disco, mientras que con caches finitas hay múltiples accesos al disco, al menos uno por cada unidad que se acceda por primera vez, para incorporarla a la cache.

Pruebas en PC con djgpp

El mismo análisis de los tiempos en segmentar 2000 palabras se ha realizado sobre un PC procesador 486DX, y los resultados son similares salvo que se obtienen tiempos mayores.

En el PC la reducción del tiempo de consulta es mucho más significativa

que en la Sun SparcStation.¡¡ Se pasa de unas 20 horas con todos los haces en disco a unos 10 minutos con una cache bien configurada !!. Básicamente ocurre que el sistema operativo no tiene buffers de E/S y cada lectura o escritura sobre el fichero con los haces es un acceso físico al disco. Por lo tanto el funcionamiento sobre disco es muy lento, y grande la mejora que se consigue con la cache, porque hay muchos accesos a disco para ser evitados.

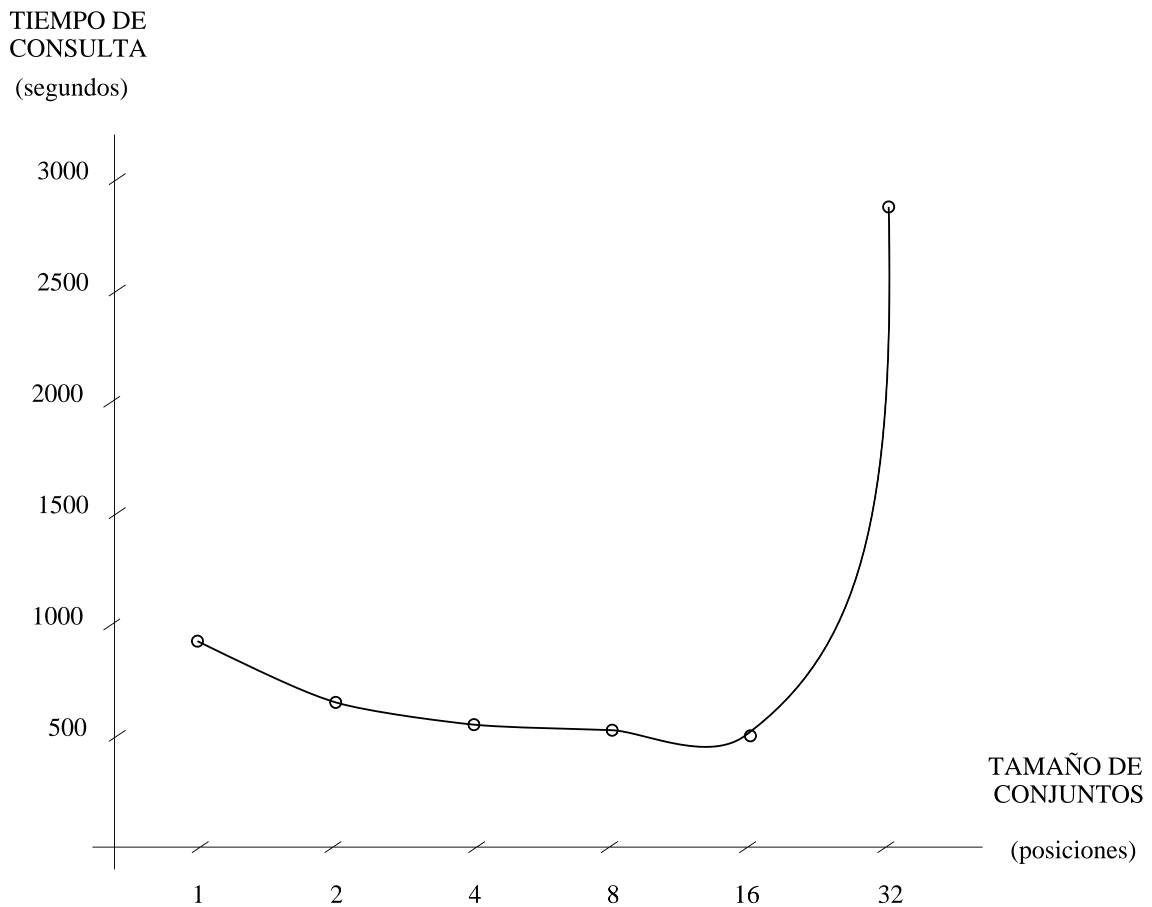

Figura 5.6: Tiempo de consulta en PC con conjuntos de distinto tamaño.

En el comportamiento del tiempo de consulta con el tamaño de los conjuntos en PC se muestra en la figura 5.6, donde el tamaño de la cache se ha mantenido constante a 10000 posiciones. Se observa que no son convenientes

conjuntos muy pequeos, como de 1 o 2 posiciones. Aparte de empeorar el tiempo de consulta con conjuntos muy grandes tal como apuntamos en la gráfica con Sun SparcStation, ahora el tiempo de consulta también empeora con conjuntos demasiado pequeños. La razón es que con tamaños pequeños las colisiones (expulsar un puntero de un conjunto para hacer hueco a otro) son más probables, y al no haber buffers de E/S estas sustituciones son muy costosas en tiempo porque conllevan accesos reales a disco.

De la figuras 5.4 y 5.6 también se desprende que las caches asociativas por conjuntos ofrecen mejores prestaciones que las de correspondencia directa (en las figuras son las de conjuntos de 1 posición) y notablemente mejores que las caches totalmente asociativas, que tienen un único conjunto de tantas posiciones como tenga la cache entera.

Como se observa, el PC viene a ser unas 8 veces más lento en responder a las consultas que la SparcStation de Sun. Además se puede apreciar que en PC el empleo de la cache es imprescindible, porque el disco es muy lento debido a la ausencia de buffers de E/S del sistema operativo. Si no se utiliza adecuadamente configurada los tiempos de consulta se disparan y la ejecución se ralentiza notablemente.

5.5 BASES EXPANDIDAS DE DISTINTOS TAMAÑOS

En esta sección comentaremos cómo evolucionan el tamaño de los diccionarios objeto y el tiempo de compilación en función del volumen de la base léxica expandida.

En la figura 5.7 se observa que tanto el tamaño del fichero con los haces como el volumen del trie *crecen linealmente* con la longitud (y el numero de entradas) de la base léxica expandida, según una recta que pasa por el origen. También se observa que el trie ocupa mucho menos que el fichero con los haces, que es la parte más voluminosa (85 %) del diccionario objeto. Para una base léxica de 4'3 MB se generan 8'5 MB de haces y 1'4 MB con el trie, esto es, 10 MB de diccionario objeto.

Quizá la linealidad del trie sea más sorprendente porque con cada nueva palabra sólo se incorporan como nodos del trie las letras que no coincidan con ninguna de las raíces que tiene el trie hasta ese momento. Esta idea

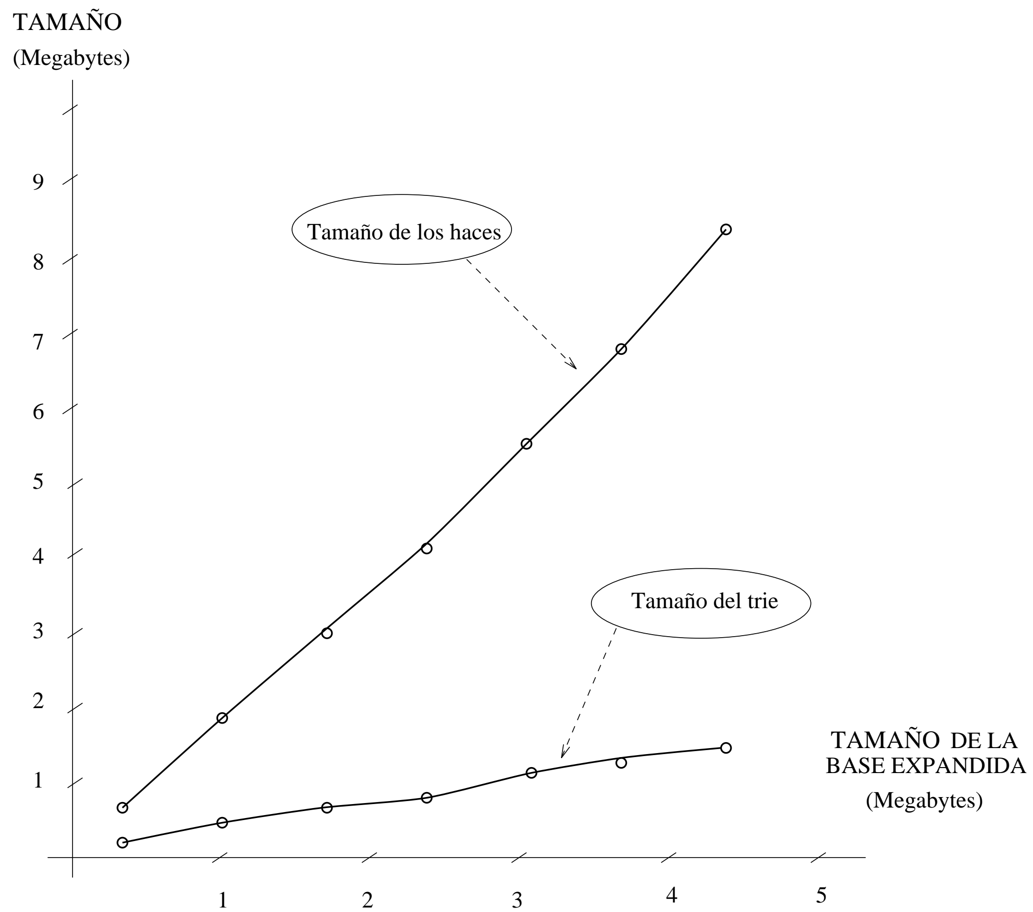

Figura 5.7: Tamaño del trie y del fichero con haces.

sugiere un crecimiento logarítmico, con incrementos en el tamaño del trie cada vez menores a medida que se compilan bases fuentes mayores, porque cada vez hay más raíces en el trie y las entradas nuevas incorporarán menos nodos. Sin embargo las pruebas realizadas muestran claramente crecimiento lineal. En la base léxica fuente de 4'3 MB con la que hemos trabajado la longitud media de las entradas es de 8'2 letras. En media cada una de esas entradas incorpora al trie menos de 4'8, es decir, que las 3'4 primeras letras de cada entrada ya están en el trie cuando se compila una entrada. Dado el crecimiento lineal, ésta proporción se mantiene aproximadamente constante para los distintos tamaños de base fuente probados. Al introducir sólo los alomorfos en el trie no aparece una repetición notoria de los comienzos de entradas, y por lo tanto el crecimiento se mantiene lineal. Si en lugar de incorporar los formantes se hubieran introducido todas las palabras flexionadas entonces probablemente se hubiera notado más la estabilización en el crecimiento del trie a medida que se incorpora mayor número de entradas.

Las pruebas realizadas también mostraron un *incremento lineal* del tiempo de compilación a medida que es mayor la base expandida que se procesa.

5.6 SEGMENTACIÓN.

Como comentamos en el capítulo 3 la consulta principal que realiza el analizador morfológico al diccionario es el cálculo de *segmentaciones de existencia* de las palabras. Estas segmentaciones son la fragmentación de la palabras en segmentos o formantes que tienen existencia como unidades del diccionario.

El algoritmo de segmentación es un algoritmo que prueba todas las particiones posibles de la palabra en subconjuntos de letras. De estas particiones sólo aquellas en las que todos los subconjuntos sean unidades reconocidas por el diccionario serán segmentaciones de existencia. Para no probar todas y cada una de las particiones se consulta al diccionario si contiene distintos segmentos iniciales. Las particiones cuyo primer segmento no pertenezca al diccionario se desechan por lo tanto con una sola consulta. Para los segmentos iniciales con significado se lanza recursivamente la segmentación para el resto de la cadena, y si ésta tiene éxito entonces se ha encontrado una segmentación de existencia de la palabra completa.

A este algoritmo general que prueba con todas las particiones posibles desde los segmentos iniciales le llamaremos **segmentador universal** porque

número de entradas	segmentador universal *(segs)*	segmentador sobre el trie *(segs)*	ganancia relativa
15000	5’4	2’8	48%
240000	84’5	44’5	47%

Tabla 5.2: Tiempos de segmentación en SparcStation.

es independiente de la estructura que almacene las unidades del diccionario. De hecho la única interacción que tiene con el diccionario es la que pregunta si tal grupo de letras está en el diccionario o no.

Si se almacena el conjunto de entradas en un trie, entonces se puede aprovechar esta estructura para optimizar el algoritmo de segmentación. Como se explicó en el capítulo 3 la ventaja del trie es la que ilustra el siguiente ejemplo: si se está en el nodo `i` de `camillero` y éste no tiene ningún hijo con la `l` entonces se sabe que los segmentos iniciales `camil, camill, camille, camiller` y `camillero` no existen en el diccionario. Por lo tanto no se pierde tiempo en preguntar por cada uno de esos segmentos individualmente. Este algoritmo optimizado requiere que las entradas se almacenen en un trie y por ello le llamaremos **segmentador sobre el trie**.

En esta sección compararemos las prestaciones en tiempo de ambos segmentadores, el que aprovecha el trie y el universal, en la estación de trabajo Sun Sparc 10 . A tal efecto se han agrupado palabras en ficheros de distintos tamaños (palabras representativas del uso del castellano) y se ha medido el tiempo que se tardaba en segmentar todas esas palabras. La tabla 5.2 muestra los resultados obtenidos.

Como ilustra la tabla 5.2 la mejora que introduce segmentar aprovechando las ventajas del trie es del 47 %, que es un porcentaje significativo. Esta mejora relativa aumentaría si las palabras fueran más largas porque entonces se descartarían más segmentos iniciales con el trie. Es decir, si segmentando `casa` el segmento `ca` no existe, entonces nos ahorramos consultar si `cas` y `casa` están en el diccionario. Si segmentando `rascacielos` el segmento `ras` no existe, entonces nos ahorramos consultar `rasc`, `rasca`, `rascac`, etc. No obstante, si las palabras son más largas el tiempo absoluto de segmentación es mayor que si son cortas.

Esta mejora en el tiempo de segmentación justifica la decisión de incorporar la función que segmenta dentro del sistema diccionario, para que pueda aprovechar el trie. Con un segmentador universal, independiente de la estructura del diccionario, se consiguen tiempos peores que retardarían el análisis morfológico.

En cuanto al tiempo absoluto de segmentación se ha conseguido, aprovechando el trie, segmentar 3 millones de palabras en 9 minutos y 20 segundos. Esas palabras se escogieron de un corpus con texto real y por ello reflejan por aparición y frecuencia, el uso real del castellano. Por lo tanto nuestro diccionario es capaz de segmentar a un ritmo de más de 5000 palabras por segundo.

Este tiempo es independiente de la configuración de cache que se escoja porque la cache sólo afecta al acceso a los haces, y no al acceso al trie. Si se quiere que cada segmentación incorpore copias de los *polisignificados* de cada segmento, entonces sí influye la cache. Ésto por ejemplo es lo que hace el segmentador cuando trabaja para el procesador morfológico descrito en [Gon95]. En esta situación el tiempo de segmentación incluye el tiempo en replicar los polisignificados. Midiendo tiempos en este caso se observa que los tiempos absolutos de segmentación se multiplican por 10 con respecto a los de la tabla 5.2, por lo que el ritmo medio baja a unas 500 palabras segmentadas por segundo. También se observa que las diferencias entre utilizar el segmentador universal o aprovechar el trie para segmentar se hacen menos significativas. Estas observaciones apuntan a que la mayor parte del tiempo se gasta en hacer esas copias de los polisignificados. Como el tiempo en recorrer el trie (bien con segmentador universal, bien con segmentador interno al diccionario) es menos relevante, la mejora de segmentar aprovechando al trie queda enmascarada. Por ejemplo segmentando 240000 palabras con una cache de 10000 posiciones y conjuntos de 2, la mejora es tan sólo del 5% en el tiempo total de segmentación.

5.7 EFECTO DE LAS COPIAS

5.7.1 Consultas con copia.

Como comentamos en capítulo 3 el diccionario puede usarse en modo destructivo y en modo no destructivo. En el primer caso el usuario tiene previsto

número entradas consultadas	tiempo copia *(segs)*	tiempo consulta *(segs)*	coste relativo
60000	46	69	67%
112000	85	126	67%
175000	134	196	67%
302000	229	335	68 %

Tabla 5.3: Tiempos de consulta y copia en SparcStation con cache infinita.

modificar las estructuras con el significado de la entrada que consulta. Por ello el diccionario no entrega estructuras originales sino que para mantener su integridad entrega una copia en memoria de los significados. Con el uso no destructivo los programas usuarios no pretenden cambiar los significados y sólo accederan a ellos para leerlos. En este último caso el diccionario puede permitir a los usuarios consultar directamente sus estructuras originales. Como vimos anteriormente se han desarrollado interfaces de acceso para ambos usos.

En esta sección vamos a las comentar las diferencias en tiempo que hay en consultar al diccionario a través de una interfaz u otra. Es decir, vamos a medir el retardo que provoca tener que replicar las estructuras originales cuando se usa el diccionario en *modo destructivo*.

Para medir tiempos se han construido unos ficheros de formantes desde los ficheros de palabras que mencionamos en la sección anterior. Para cada una de esas palabras se han calculado todas sus segmentaciones de existencia y cada uno de los formantes de estas particiones se ha incorporado al fichero de formantes. Por lo tanto se tiene un fichero con los formantes en la proporción en que el procesador morfológico los consultará cuando analice texto castellano real.

Consultar a través de la interfaz de uso destructivo consiste básicamente en buscar la estructura original, copiarla en memoria y acceder a la réplica. En los tiempos de la tabla 5.3 se mide el porcentaje que se lleva el copiar estructuras cuando se utiliza la interfaz destructiva del diccionario.

La tabla 5.3 se ha obtenido configurando el diccionario con el trie enteramente en memoria y cache infinita, que es una situación típicamente ópti-

número de entradas consultadas	tiempo de copia *(segs)*	tiempo de consulta *(segs)*	coste relativo
60000	141	169	83%
112000	265	319	83%
175000	410	494	83%
302000	726	878	83%

Tabla 5.4: Tiempos de consulta y copia en SparcStation con cache finita.

ma cuando se ejecuta el código en una estación de trabajo. Si en lugar de una cache infinita se trabaja con caches finitas el tiempo total de consulta aumenta porque es más lento acceder al conjunto de haces, que reside entonces parcialmente en disco. Además el porcentaje gastado en hacer copias aumenta porque para copiar hay que acceder a las estructuras originales a través de la cache, que al ser finita es más lenta. Así lo confirma los resultados obtenidos con una cache de 20000 posiciones y conjuntos de 4, que se muestran en la tabla 5.4. En lugar del 67% empleado con cache finita, se gasta el 83% del tiempo de consulta en replicar en memoria las estructuras originales.

5.7.2 Trie con copia.

En la misma línea que señala que las copias de información ralentizan el funcionamiento del diccionario se encuadra la optimización que se hizo sobre la idea original del trie.

El trie descrito en [Jun et.al92] utiliza el carácter # como terminador de las entradas. Debido a esto cada vez que se busca una entrada debía hacerse una copia local de esa entrada y concatenar el carácter # al final de esa copia, porque esa sería la cadena que estaría o no inserta en el árbol trie. Para evitar esa copia no se podía concatenar alegremente el # al parámetro que pasa el usuario para buscar, porque esa memoria es del programa que utiliza el diccionario y no se puede modificar, ni siquiera temporalmente reestableciendola después de procesar la entrada. La opción más elegante para evitar esa copia es poder utilizar directamente la cadena que pasa el

programa usuario para navegar por el trie. Es decir, considerar el caracter '\0', que se utiliza como carácter terminador de cadenas en lenguaje ansi C, también como terminador de cadenas en el trie.

Gracias a esta optimización no es necesario copiar cada cadena que se busca antes de recorrer el trie. Para constatar la mejora en el tiempo de respuesta del diccionario se han medido tiempos de consulta y tiempos de segmentación para distintos conjuntos de formantes y palabras. Tanto en consulta como en segmentación se observa que al evitar las copias utilizando el carácter '\0' como terminador de cadenas, los tiempos de respuesta del diccionario se reducen en un 15 %.

Capítulo 6

CONCLUSIONES Y PROPUESTAS.

6.1 CONCLUSIONES.

La *base léxica* desarrollada permite almacenar eficientemente gran cantidad de *información lingüística* asociada a un *léxico* muy extenso, y posibilita el rápido acceso por otros programas de aplicación a esa información. Por ello es muy útil en sistemas de procesamiento del lenguaje natural.

Como se describe en capítulo 2, inicialmente toda la información disponible reside en el fichero *base léxica fuente*, en un formato especificado por un *formalismo léxico*. Para poder acceder a esos datos de modo eficiente el sistema diccionario procesa el contenido del fichero base léxica fuente y vuelca su información a unas estructuras que permiten un acceso más rápido. En primer lugar la base léxica fuente se *expande* para dar lugar a la *base léxica expandida*. Este proceso consiste en explicitar toda la información que aparece compactada e implícita en la base fuente. En segundo lugar la base expandida se *compila* para dar lugar al *diccionario objeto*. Este paso consiste en la generación de unas estructuras que almacenan la información lingüística ocupando un espacio mínimo, y que permiten un rápido acceso a ella en tiempo de ejecución. Estas estructuras que conforman el diccionario objeto son los **haces de rasgos** para almacenar los datos asociados a cada entrada, y el **trie** para almacenar el conjunto de entradas. El diccionario objeto reside en un conjunto de ficheros, por lo que no es necesario generarlo cada vez que

se va a usar la base léxica.

Para que un programa de aplicación pueda consultar la información contenida en una base léxica fuente debe haberse generado el diccionario objeto correspondiente y el programa se limita a *cargar* ese diccionario objeto y a acceder a la información a través de las *interfaces de acceso* que el sistema diccionario ofrece.

6.1.1 Eficiencia temporal y espacial.

En el desarrollo de la base léxica el objetivo prioritario ha sido el tiempo de respuesta, sin descuidar que el tamaño del diccionario objeto típico fuera razonable. Este criterio ha guiado la elección de estructuras para almacenar el conjunto de rasgos y el conjunto de entradas.

La base fuente concreta con la que se ha trabajado ocupa 4'3 MB, contiene toda la información morfológica y sintáctica asociada a unas 48000 entradas (fundamentalmente morfemas, raíces y algunas palabras de flexión irregular) y puede considerarse relativamente completa. Esta base fuente genera un diccionario objeto de 10 MB: 8'5 MB para los haces de rasgos y 1'4 MB para el trie. Es decir, el diccionario objeto cabe perfectamente en el disco duro de todas las plataformas hardware más comunes. Además, como vimos en el capítulo 5, el tamaño del diccionario objeto crece linealmente a medida que se tienen bases fuentes mayores. Por lo tanto, en cuanto a la eficiencia espacial, el tamaño de los diccionarios objeto generados es razonable y no son esperables tamaños mucho mayores.

En cuanto a la eficiencia temporal, el diccionario objeto con el que se ha trabajado permite calcular las segmentaciones de existencia de 3 millones de palabras en menos de 10 minutos. Es decir el diccionario permite segmentar más de 5000 palabras por segundo. Si se va a modificar el haz de rasgos asociado a cada segmento entonces hay que utilizar la interfaz de copia en memoria. En este caso el diccionario tarda más en segmentar porque pierde tiempo en realizar las copias de los haces de rasgos y la media baja hasta 500 palabras segmentadas por segundo.

Estos tiempos de respuesta conseguidos permiten el uso del diccionario en un sistema de lenguaje natural *de tiempo real*, es decir, que pueda interactuar con un usuario humano al ritmo que éste suele comunicarse con otras personas. Los tiempos conseguidos de segmentación dejan todavía mucho margen a los análisis morfológico, sintáctico y semántico para que el sistema

global de lenguaje natural responda en tiempo real al usuario humano.

Es destacable que el diccionario es significativamente más rápido (unas 10 veces más veloz) si se utiliza no destructivamente, es decir, a través de la interfaz sobre las estructuras originales.

6.1.2 Diccionario objeto.

La explotación de la base léxica fuente se ha dividido en dos pasos básicos: generación de diccionario objeto (expansión y compilación) y acceso a sus estructuras (carga y acceso a través de interfaces). Esta separación es muy ventajosa porque desvincula la utilización de las estructuras con toda la información, de su generación, que suele ser costosa en tiempo. Es decir, que no es necesario generarlas cada vez que se vayan a utilizar. Gracias a esta segregación, los haces de rasgos y el trie se generan una sola vez para cada base fuente, almacenándose en los ficheros del diccionario objeto. Posteriormente cada vez que se necesite la información de la base léxica se carga el diccionario objeto, no es necesario procesar la base léxica fuente. En compilar la base léxica expandida concreta con que se ha trabajado se tarda 30 minutos como mínimo, mientras que en cargar el diccionario objeto correspondiente se suele tardar unos 4 segundos. Si hubiera que compilar cada vez que se utiliza la base léxica sería imposible cualquier aplicación en tiempo real, como por ejemplo un interfaz interactivo de lenguaje natural, porque el usuario tendría que esperar media hora para que el sistema empezara a responderle.

Para que el diccionario objeto se pueda utilizar en distintas sesiones es necesario que las estructuras que soporta no hagan ninguna referencia a zonas de memoria, es decir, no contengan punteros de memoria. El trie se ha implementado con un *doble array* según la idea de [Jun et.al92]. Esta implementación ya evita el manejo de punteros de memoria para enlazar los distintos nodos del árbol trie; en su lugar se utilizan índices de arrays. Con respecto a los haces de rasgos se ha creado un espacio de direccionamiento virtual para enlazar los nodos de los árboles de rasgos, que se unen por lo tanto con *punteros virtuales*.

Además de permitir la reutilización del mismo diccionario objeto en distintas sesiones, esta indirección permite flexibilidad de ubicación del conjunto de haces. Los punteros virtuales se pueden traducir a posiciones concretas de memoria o a posiciones dentro de ficheros en disco. Por lo tanto los haces se

pueden ubicar en disco, en memoria o parcialmente en ambos soportes, según convenga en cada caso, y traducirse los punteros virtuales como corresponda.

6.1.3 Uso de la memoria de la plataforma.

Como hemos visto tanto el trie como los haces de rasgos no hacen referencia a posiciones de memoria; los distintos enlaces que se utilizan en sus estructuras son índices de arrays o punteros virtuales. Ello permite que la ubicación de estas estructuras pueda ser tanto ficheros de disco como la memoria principal de la plataforma donde se ejecuta el código.

En general si se trabaja con las estructuras residiendo en disco, su acceso y consulta será mucho más lento que si se localizan en memoria porque los accesos a memoria principal son mucho más rápidos que los accesos a memoria secundaria, esto es, dispositivos de almacenamiento como el disco (vease la tabla 4.1). Por lo tanto será conveniente, dentro de las limitaciones de la plataforma hardware, ubicar el diccionario objeto en memoria. De la ubicación del diccionario objeto depende notablemente la rapidez de nuestro diccionario.

En cuanto al trie los índices se pueden interpretar directamente como índices dentro de ficheros, si el trie reside sólo en los archivos de disco, o como incrementos de posiciones en memoria si el trie se ha llevado a memoria. Por lo tanto el sistema diccionario puede funcionar situando al trie completamente en memoria o enteramente en disco. Esto se decide con una directiva de compilación del código fuente, que se elije según los recursos de la plataforma concreta donde se vaya a ejecutar. Como los accesos al trie son muy frecuentes cuando se consulta al diccionario, es muy recomendable que se trabaje con el trie enteramente en memoria. Además dado su tamaño esto no representa un consumo muy grande.

En cuanto al conjunto de los haces los punteros virtuales se pueden interpretar como posiciones dentro de un archivo en el disco o se pueden traducir a direcciones de memoria. El diccionario puede funcionar con todos los haces de rasgos completamente en el disco, y entonces los punteros virtuales se interpretan como posiciones dentro de ficheros.

El diccionario también puede funcionar incorporando todo el conjunto de haces enteramente a memoria. Para este caso los punteros virtuales se traducen aritméticamente, de manera muy rápida, a punteros de memoria. Este modo sólo se podrá habilitar cuando el conjunto de haces se pueda llevar

completamente a la memoria del sistema.

Como vimos en el capítulo 5, el conjunto de haces es la parte más voluminosa del diccionario objeto, ocupando aproximadamente el 85 % de su tamaño. Por ejemplo para la base léxica con la que se ha trabajado, relativamente completa, se genera un conjunto de haces de unos 8'5 MB. Sabiendo que con este tamaño determinadas plataformas hardware no podrían situarlo completamente en memoria, se han desarrollado mecanismos que permiten una incorporación parcial en memoria. Para ello se ha realizado un esquema de tabla cache que lleva a memoria sólo los haces más frecuentemente accedidos del conjunto, manteniendo el resto en disco. La tabla cache traduce cada puntero virtual a su correspondiente localización.

Las tres opciones presentadas se corresponden con distintos grados de utilización de memoria del sistema para acelerar el funcionamiento del diccionario. La primera opción con todos los haces en disco (también llamada cache nula) es la más lenta y sólo se empleará en plataformas muy pobres en memoria. La seguna opción, con todos los haces en memoria (también llamada cache infinita) es la más rápida, pero la más exigente con la plataforma; es la que más memoria necesita. Será la opción típicamente elegida en las estaciones de trabajo. La tercera opción (también llamada cache finita) se sitúa entre las dos anteriores y ofrece prestaciones intermedias. Se utilizará por ejemplo en arquitecturas PC en las que no hay memoria para tener todos los haces en ella y sin embargo si se dispone de cierta cantidad que se quiere aprovechar.

La ubicación final del conjunto de haces se determina en tiempo de ejecución con unos parámetros de configuración que seleccionan el funcionamiento totalmente en disco, con cache normal o con cache infinita. Como hemos visto, pese a que nuestro sistema diccionario puede funcionar sin utilizar memoria para los haces, la elección adecuada de estos parámetros de configuración le permite aprovechar la existente en la plataforma concreta para acelerar su funcionamiento.

6.1.4 Portabilidad

La base léxica desarrollada se ha escrito completamente en lenguaje estándar *ansi C*, lo que garantiza una gran portabilidad del código a distintas máquinas. Al ser lenguaje estándar la mayoría de los compiladores son capaces de generar código ejecutable desde él.

Esta portabilidad se ha contrastado ejecutando el código en distintas arquitecturas como una Sun SparcStation 10 (sistema operativo SunOs4.1.3), una estación de trabajo Hewlett-Packard Apollo 9000 (sistema operativo HP-UX A.09.05) y un PC 486DX (sistema operativo DOS 6.2). Se ha utilizado el compilador gcc sobre cada una de las estaciones de trabajo y el compilador djgpp sobre el PC. En todas ellas se ha ejecutado con éxito y con las lógicas diferencias en velocidad debidas a las distintas eficiencias de las arquitecturas.

Aparte de la velocidad otra de las diferencias entre arquitecturas es la debida a la distinta memoria disponible en cada una de ellas. En las estaciones de trabajo se puede ejecutar la opción cache infinita para los haces de rasgos porque ambas tienen memoria virtual suficiente. Sin embargo, en una plataforma PC con 8 MB de memoria RAM no es posible ejecutar la opción cache infinita con la base léxica concreta que hemos utilizado. El sistema operativo no concede los 8'5 MB de memoria dinámica para situar en ellos el conjunto de los haces.

Otra diferencia entre arquitecturas aparece en el tiempo de compilación del código. Como vimos en el capítulo 5, para que funcione adecuadamente es necesario compilar los programas fuente del sistema diccionario con una directiva de compilación adecuada (PC o UNIX) que permite elegir la manera conveniente de abrir los ficheros binarios en cada arquitectura. En cualquier caso el código fuente no hay que alterarlo, funciona correctamente en todas las arquitecturas probadas.

6.2 PROPUESTAS.

1. Como hemos comentado, existe una diferencia muy notable entre utilizar la interfaz sobre estructuras originales y usar la interfaz sobre copias de memoria. En el segundo caso el tiempo de respuesta del diccionario se multiplica por 10. Este hecho señala que el tiempo que se emplea en duplicar la información asociada a las entradas es significativo para la eficiencia temporal. Por lo tanto apunta claramente a la necesidad de emplear modelos lingüísticos que no requieran la modificación de la información asociada al léxico del diccionario, y procesadores que minimicen el número de copias de información necesarias.

   Así por ejemplo un analizador morfológico que no necesitara modificar

los haces de rasgos de los formantes en el diccionario podría ser mucho más rápido que el analizador equivalente que sí modifica los rasgos. Para poder aplicar esta idea sería necesario que el modelo computacional pudiera determinar si una descomposición es morfológicamente correcta o no, solamente leyendo los haces de sus segmentos. En este caso sólo se realizaría la copia de rasgos, para construir el haz asociado a la palabra en la descomposición válida. Si por el contrario se necesita modificar los haces de los formantes para determinar la corrección morfológica entonces hay que copiar los rasgos de los formantes para todos los segmentos de todas las segmentaciones de existencia de la palabra, incluídas las numerosas que no sean morfológicamente válidas. En el primer caso nos estaríamos ahorrando el tiempo de copia de los rasgos de los todas las segmentaciones de existencia incorrectas morfológicamente.

Esta mejora se encuadra dentro de la idea de minimizar copias, que ya comentamos en el capítulo 5.

2. Se ha observado que muchas entradas de la base léxica expandida tienen muchos rasgos iguales. En concreto todos iguales menos el rasgo *lex*. Esta repetitividad se podría aprovechar para reducir el tamaño del fichero con el conjunto de haces. Se podrían utilizar macros que dieran nombre a esos conjuntos de rasgos que se repiten con mucha frecuencia. En vez de guardar explícitamente todos los rasgos de cada entrada se guardarían sólo los rasgos genuinos, exclusivamente suyos, mientras que los rasgos que se repiten con mucha frecuencia no se almacenarían explícitamente en cada entrada, sino que sólo se guardaría una referencia (que ocupa menos espacio) a la macro que los incluye.

   Habría que estudiar si la resolución de esa referencia en tiempo de ejecución retardaría excesivamente el tiempo de respuesta del diccionario. La reducción del tamaño del conjunto de haces permitiría por ejemplo trabajar con todo él en memoria. Como hemos visto, esto era imposible en algunas plataformas porque no tenían suficiente memoria para albergar al conjunto entero, que era demasiado grande. Quizá el poder situar todos los haces en memoria compense el tiempo en resolver la invocación de macros.

3. Otra de las líneas en las que se puede continuar la labor iniciada en

este proyecto es el estudio de analizadores morfológicos integrados en el diccionario, es decir, diccionarios que contenen todas las palabras ya flexionadas y asociada su información gramatical.

La ventaja que hace atractiva esta línea es que el tiempo de análisis morfológico sería nulo, ya que se encontraría el resultado del análisis morfológico de la palabra a la vez que se busca en el trie.

Tal como se mencionó en el capítulo 1, esta idea se desechó porque generarían diccionarios demasiado grandes. Sin embargo a la luz de los tamaños que arroja la base léxica expandida con la que se ha trabajado y de algunas otras consideraciones que ahora veremos, la idea no queda tan nítidamente dentro del terreno de lo imposible como a priori parecía.

En primer lugar el número de entradas se multiplicaría por un cierto factor con respecto al que tiene la base léxica de formantes. Grosso modo cada raíz nominal originaría cuatro entradas: masculino plural y singular, y los mismos para el femenino. Cada raíz verbal originaría unas 50 palabras flexionadas, atendiendo a las diferentes combinaciones de tiempo, modo, etc. Las preposiciones, adverbios y pronombres no generarían entradas adicionales. Habría que ver empíricamente el factor medio real de multiplicación del número de entradas, que determinaría el tamaño del nuevo trie y del nuevo conjunto de haces. El tiempo de respuesta es independiente del número de entradas que el trie almacene, por lo que no se vería afectado por este aumento.

En cuanto al tamaño del trie es de esperar que el crecimiento fuera menor que lineal con el número de entradas porque ahora sí que habría claramente una repetición de raíces. Es decir `chico, chicos, chica` y `chicas` comparten el mismo comienzo `chic-`, del mismo modo todas las formas verbales de `respirar`; por ello en media bajaría el número de letras nuevas que cada entrada aporta al trie, y el trie podría tener un tamaño manejable.

En cuanto al tamaño del conjunto de los haces, si bien aumenta porque hay más entradas, por otro lado puede compensarse con que cada entrada almacena ahora menos información. Por ejemplo ahora toda los rasgos con información exclusivamente morfológica son innecesarios. Además se podría combinar este enfoque con el uso de macros en tiem-

po de ejecución para dar conjuntos de haces más pequeños.

La utilidad de este enfoque la marca el compromiso entre el mínimo tiempo de análisis morfológico que ofrece y el tamaño de diccionario objeto que necesitaría. A la luz de estas nuevas consideraciones quizá mereciera la pena revisar experimentalmente que sigue siendo una opción no utilizable en la práctica.

4. En pruebas comparativas que se hicieron integrando nuestro diccionario con el procesador morfológico [Gon95] se comprobó que en una situación típica de estación de trabajo, el tiempo gastado en consultar la base léxica representaba el 15 % del tiempo global de análisis. Esto empleando la interfaz de copia en memoria, que como hemos señalado es 10 veces más lenta que la interfaz sobre original. Es decir, que comparativamente el tiempo dedicado a análisis es mucho mayor que el empleado en consultar el diccionario.

   Siguiendo la ley de Steandhal este hecho señala que para disminuir el tiempo global de análisis morfolo'gico es más rentable concentrar esfuerzos en acelerar el analizador que en optimizar el diccionario.

   Una posible mejora del sistema global podría ser integrar el análisis morfológico en el trie, para aprovechar la rapidez de éste último. En esta integración, distinta a la sugerida en la propuesta anterior, las entradas aparecerían agrupadas en distintos tries según su categoría morfológica (raíz, morfema de género, desinencia verbal, etc.). Para analizar morfológicamente se busca primeramente si tiene raíz conocida, recorriendo el trie con las raíces. Por ejemplo al analizar `camas` se encontraría que la raíz `cama` sí tiene sentido, y se intentaría encontrar una segmentación para el resto de la cadena `s`. Para ello no se volvería a explorar el trie con las raíces sino que el nodo terminal de `cama` en el trie de las raíces podría indicar los tries en los que pueden estar las distintas terminaciones morfológicamente válidas que esa raíz admite. Para este caso concreto la raíz `cama` puede admitir concatenación de morfema plural o bien ninguna porque ella misma ya puede ser palabra. Como esta segunda opción no completa la palabra que se busca, `camas`, se desecha. La otra opción busca el resto de la palabra en el trie de los morfemas de número. Se encontraría `s`, que es morfológicamente concatenable y que además completa la palabra que se buscaba.

Por lo tanto tras recorrer esos dos tries se ha encontrado un análisis morfológico válido para la palabra `camas`.

Si por ejemplo completando una palabra se llega a un nodo terminal en el trie de raíces y ese nodo indica que no necesita ninguna concatenación para formar una palabra entonces la palabra existe como tal y su información gramatical vendrá apuntada también por ese nodo terminal.

Puesto que una misma palabra puede admitir varias interpretaciones morfológicas la búsqueda en los tries debe recorrer todas las posibilidades que aparezcan, no conformándose con la primera que se encuentre. Por ejemplo en la palabra `amo` la raíz `am` tiene sentido como raíz verbal y como raíz nominal. Se deben seguir esas dos posibilidades, continuando en el trie de las terminaciones verbales y en el de los morfemas de género respectivamente, de manera que al final del análisis se obtienen las dos descomposiciones morfológicas válidas: `sustantivo masculino singular` y `primera persona presente de indicativo del verbo amar`.

Una ventaja de este enfoque es que al utilizar las reglas morfológicas para navegar entre los tries no se pierde el tiempo en calcular segmentaciones de existencia que no son morfológicamente correctas. Se buscan directamente las descomposiciones válidas; la segmentación se realiza al mismo tiempo que el análisis. Otra ventaja importante es que el análisis se reduce a recorrer varios tries, que como muestran los tiempos medidos en capítulo 5 es una tarea realmente rápida. No se generan las estructuras de análisis (los charts) que utiliza por ejemplo el procesador morfológico de [Gon95].

En este enfoque tanto las raíces como los morfemas nominales y terminaciones verbales se almacenan sólo una vez, lo que da la economía en espacio del diccionario de formantes. Es decir sólo se tiene una terminación `-aba` y no una por cada verbo que la utiliza: `amaba, respiraba, miraba,` etc. Las distintas posibilidades de combinación se plasman en distintos recorridos entre y dentro de los tries. No necesita modificar los constructos inferiores para construir el haz de rasgos de la palabra, por lo que no se pierde tiempo en hacer copias de información.

Las posibles continuaciones dependen por supuesto de las reglas morfo-

lógicas de la palabra, que de este modo se integran en el trie. Este enfoque se basa en la sencillez de las reglas morfológicas, que expresan la palabra básicamente como concatenación de una raíz y uno o dos morfemas. No son necesarias para morfología gramáticas más potentes y complejas como las de contexto libre, cuya incorporación al trie sería más complicada si no imposible.

Bibliografía

[KerRit88] Kernighan, Brian y Ritchie, Dennis. *El lenguaje de programación C*, Prentice Hall, 1988.

[CarHay87] Carbonell, Jaime and Hayes, Philip. *Natural-Language Processing.* In *Encyclopedia of Artificial Intelligence*, Stuart Shapiro (Ed.), vol.1, pp. 660-677. John Wiley & Sons, 1987.

[GoñGon95] Goñi, José Miguel and González, José Carlos. *A framework for lexical representation.* In AI95: Fifteenth International Conference. Language Engineering, Montpellier, France, June 1995.

[Gon et.al95] González, Angel Luis; Goñi, José Miguel y González, José Carlos. *Un analizador morfológico para el castellano basado en chart.* En Actas de la VI Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA'95), Alicante, Noviembre 1995.

[MorGoñ95] Moreno, Antonio and Goñi, José Miguel. *A morphological processor for Spanish implemented in prolog.* Proceedings of the Joint Conference on Declarative Programming (GULP-PRODE'95), Marina di Vietri, Italy, September 1995.

[Goñ et.al95a] Goñi, José Miguel; González, José Carlos and Moreno, Antonio. *A lexical platform for Spanish.* Proceedings of Workshop on the Computational Lexicon (ESSLLI'95), Barcelona, August 1995.

[Goñ et.al95b] Goñi, José Miguel; González, José Carlos and Nieto, Amalio Francisco. *ARIES: a ready for use platform for engineering Spanish-processing tools.* Digest of the Second Language Engineering Convention, pp. 219-226, London, October 1995.

[GazMel89] Gazdar, Gerald and Mellish, Chris. *Natural Language Processing in Prolog. An Introduction to Computational Linguistics.* Addison Wesley, 1989.

[Gon95] González, Angel Luis. *Realización de un procesador morfológico para el castellano.* Departamento de Matemática Aplicada a las Tecnologías de la Información, E.T.S.I.Telecomunicación, Universidad Politécnica de Madrid, 1995.

[Jun et.al92] Jun Ichi, Aoe; Morimoto, Katsushi and Sato, Takashi. *An efficient implementation of trie structures.* Software: practice and experience, vol.22, September 1992.

[Goñ et.al92] Goñi, José Miguel; González, José Carlos y López, Jesús. *Especificación del formalismo léxico para ARIES.* Departamento de Matemática Aplicada a las Tecnologías de la Información y Departamento de Ingeniería de Sistemas Telemáticos, E.T.S.I. Telecomunicación. Universidad Politécnica de Madrid, 1993.

[Mor92] Moreno, Antonio. *Un modelo computacional basado en la unificación para el análisis y generación de la morfología del español.* Tesis doctoral, Depto. de Lingüística, Lenguas Modernas, Lógica y Filosofía de la Ciencia, Facultad de Filosofía y Letras. Universidad Autónoma de Madrid, 1992.

[Aho et.al83] Aho, Alfred; Hopcroft, John and Ullman, Jeffrey. *Data Structures and Algorithms*, Addison Wesley, 1983.

[Sil et.al91] Silberschatz, Abraham; Peterson, James and Galvin, Peter. *Operating System Concepts*, Addison Wesley, 1991.

[Goñ et.al92] Goñi, José Miguel; González, José Carlos y López, Jesús. Curso de procesamiento de lenguaje natural. Departamento de Matemática Aplicada a las Tecnologías de la Información y Departamento de Ingeniería de Sistemas Telemáticos, E.T.S.I. Telecomunicación. Universidad Politécnica de Madrid, 1992.

[Yll et.al83] Yllera, Alicia y otros. Introducción a la Lingüística. Editorial Alhambra Universidad, Madrid, 1983.

La documentación, los listados y los ficheros que forman parte de este Proyecto pueden solicitarse en la siguiente dirección:

Departamento de Matemática Aplicada
a las Tecnologías de la Información.
E.T.S.Ingenieros de Telecomunicación.
Universidad Politécnica de Madrid.
28040 MADRID.