Introducción a la metodología de diseño descendente

Josep Vilaplana Pastó
M. Jesús Marco Galindo

PID_00149892

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

En este módulo aprenderemos a desarrollar algoritmos para problemas más complicados que los que hemos visto hasta ahora. No introduciremos ningún elemento más del lenguaje algorítmico. Lo que haremos, en todo caso, es madurar el uso que daremos a algunos de los elementos, como por ejemplo la declaración de acciones, las funciones y la definición de nuevos tipos. Dicho de otro modo, aquí propondremos una metodología por la cual llegaremos, si la seguimos, al diseño de algoritmos correctos e inteligibles que solucionan enunciados más complejos.

Pensad que...

... el seguimiento de la metodología que propondremos servirá para que nuestros diseños se hagan en un tiempo de desarrollo mínimamente razonable, de modo que aumente nuestro grado de eficacia al producir algoritmos correctos. Recordad los objetivos que nos habíamos propuesto en el módulo "Introducción a la programación".

Hasta ahora nos hemos dedicado, por un lado, a aprender la sintaxis y la semántica de los elementos del lenguaje algorítmico. Con la resolución de problemas sencillos hemos desarrollado la destreza al expresar nuestros algoritmos de forma clara y no ambigua. A pesar de ello, para diseñar nuestros primeros algoritmos hemos estudiado unas pautas metodológicas basadas en la aplicación de los esquemas de tratamiento secuencial. El uso de estos esquemas nos ayuda a evitar errores típicos en los que podemos caer, así como a formular algoritmos inteligibles.

Así pues, la metodología que se imparte en este curso no es exclusiva de este módulo, sino que en los módulos anteriores ya se ha propuesto de forma progresiva una metodología que hay que seguir para problemas más sencillos que los que encontraremos a partir de este momento. Sin embargo, todo lo que hemos aprendido gracias a estos problemas más sencillos nos servirá también para los problemas complejos que propondremos más adelante.

Los problemas sencillos que hemos estado resolviendo se caracterizan por tener una solución que utiliza un número razonable de elementos básicos del lenguaje algorítmico.

Los enunciados de los **problemas sencillos** hacen referencia a objetos que están muy cercanos a los tipos elementales de lenguaje o que son directamente representables por éstos.

El tratamiento que hay que aplicar a estos objetos se puede alcanzar con unas pocas asignaciones y con algunas construcciones algorítmicas. El resultado final es un algoritmo comprensible y legible en términos de los elementos básicos del lenguaje algorítmico.

Al desarrollar aplicaciones más reales, nos encontramos con problemas grandes y complejos.

Los enunciados de problemas complejos hacen referencia a objetos que no tienen una correspondencia directa con los tipos elementales de la notación algorítmica.

El tratamiento de estos objetos puede no ser elemental y, asimismo, también puede llevar a un número de sentencias y declaraciones excesivamente elevado, si expresamos todo lo que hay que hacer como una composición secuencial de asignaciones, iteraciones y/o alternativas.

El resultado es un algoritmo de muchas páginas, difícil de seguir y, por lo tanto, también de saber si funciona, y lleno de detalles (algunas veces repetidos) que lo hacen prácticamente ilegible. El número de detalles que hay que tener en cuenta es excesivo, y nuestra capacidad de ser conscientes de todos los detalles es limitada. Pensar estos enunciados en términos de los elementos básicos del lenguaje, sin unas pautas que seguir, nos puede comportar muchos quebraderos de cabeza y bastante dispersión mental, de forma que disminuye nuestra capacidad de comprensión sobre lo que hacemos y, por lo tanto, nuestra seguridad sobre si estamos realizando un algoritmo correcto y legible (mantenible). Con estas circunstancias, llegamos a la conclusión de que la notación algorítmica es tan elemental que hace incómodo y poco práctico el desarrollo de aplicaciones reales, y de que haría falta un lenguaje más cómodo que comprendiese objetos más reales.

¿Os imagináis...

... lo que conseguiríamos si expusiésemos el funcionamiento de un programa en términos del movimiento de electrones que hay por las pistas de cobre de la placa de circuito impreso y en el silicio de vuestro ordenador? Acabaríamos confundidos y distraídos por el exceso de información en la exposición, y perderíamos toda referencia a aquello de lo que estamos hablando.

Si este apreciado lenguaje existiese, con la cantidad y la variedad de problemas que hay por resolver, tendría un número tan grande de acciones y tipos que nunca acabaríamos de conocerlo a fondo. Se pueden encontrar algunos lenguajes específicos para algunas aplicaciones concretas, pero cuando no los encontramos, ¿qué hacemos?

La notación algorítmica tiene los suficientes elementos para construir cualquier algoritmo que deseemos a partir de los tipos elementales, la asignación y las composiciones algorítmicas. El lenguaje algorítmico también permite ampliar el propio lenguaje definiendo y construyendo tipos más complejos y definiendo las acciones y/o funciones que más nos convengan.

Con las características del lenguaje algorítmico, todo lo que nos hace falta es saber qué metodología debemos seguir para llevar a buen fin el diseño de un problema complejo. Esta metodología consistirá en descomponer un proble-

ma complejo en subproblemas más sencillos e independientes entre sí. La solución de estos subproblemas implicará la solución del problema. Por otro lado, la descomposición del problema se hace mediante la abstracción.

> La **abstracción** es el resultado de aislar ciertos aspectos (cualidad, atributos, etc.) de un todo para poder razonar o pensar de forma más cómoda y menos dispersa (en nuestro caso, para la solución de problemas).

En el fondo, se trata de simplificar alguna realidad y reducirla sólo a los aspectos que tengan una especial relevancia o un interés determinado para lo que estamos pensando o haciendo.

Hemos estado utilizando la abstracción prácticamente desde el inicio del curso: el modelo de secuencia, los esquemas (que podríamos considerar como algoritmos abstractos), etc. A partir de este módulo, sin embargo, tendremos que aprender a abstraer de forma más adecuada problemas más complejos que los que hemos trabajado hasta ahora. Deberemos descomponer el problema en subproblemas más sencillos. Este módulo os proporcionará las técnicas necesarias para hacerlo (herramientas de la notación para abstraer datos y abstraer código) y una metodología a seguir (diseño descendente).

Objetivos

Cuando hayáis estudiado este módulo, seréis capaces de:

- Analizar problemas de cierta complejidad y descomponerlos en subproblemas más sencillos, aplicando la técnica de diseño descendente por refinamientos sucesivos.

- Saber que hay descomposiciones de problemas más convenientes y efectivas que otras para el diseño. La elección de abstracciones convenientes al problema y el seguimiento de las pautas indicadas en este módulo os llevarán a descomposiciones efectivas.

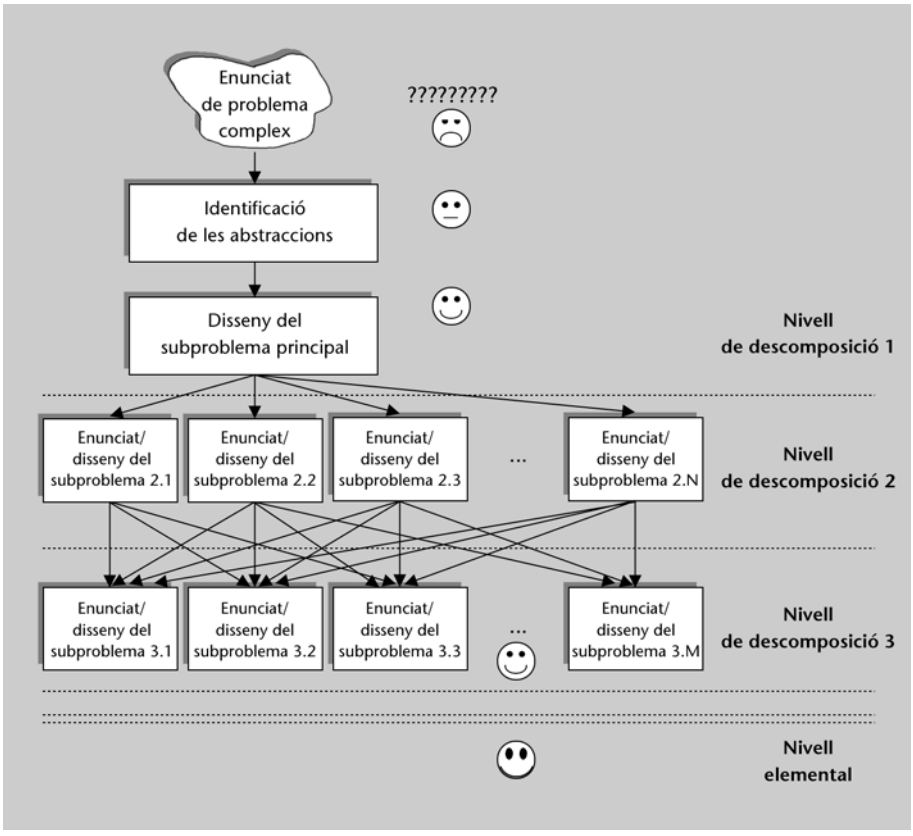
- Aplicar los conceptos estudiados en los módulos anteriores para resolver cada uno de los subproblemas sencillos, utilizando la abstracción de código y datos.

- Desarrollar algoritmos de forma efectiva y en un tiempo razonable para resolver problemas más reales.

1. Diseño descendente

1.1. Análisis problemas complejos

Como ya hemos adelantado en la introducción, la forma de enfrentarse a un problema complejo consiste en descomponerlo en una colección de subproblemas más sencillos e independientes, de forma que la solución de estos subproblemas implique la solución del problema en cuestión. Al mismo tiempo, los subproblemas se pueden descomponer en otros subproblemas si los primeros no son suficientemente sencillos. Estamos hablando de *niveles de descomposición*. Es decir, el primer nivel descompondrá el problema en los subproblemas que sea necesario; el segundo, descompondrá cada uno de los subproblemas resultantes de la descomposición del primero en otros, etc. El número de niveles de descomposición en subproblemas dependerá de la complejidad del problema. En el último nivel encontraremos los subproblemas más sencillos de resolver, es decir, los más concretos y elementales. La solución de cada subproblema contribuye parcialmente a la solución global del problema. Observad la siguiente figura:



Sólo mediante la práctica seremos capaces de realizar buenas descomposiciones. No todas las descomposiciones de un problema en subproblemas son adecuadas para desarrollar un buen diseño. La descomposición se debería basar

en algún criterio guiado por las abstracciones que creyésemos convenientes del problema.

Para ilustrar los conceptos que se dan en este apartado, os proponemos el siguiente problema:

> Una editorial está interesada en analizar los errores de un texto presentado por un autor de materiales. Nos encarga diseñar un algoritmo, que, una vez realizado el análisis de errores de un texto, nos indique el porcentaje de frases, la media de palabras por frase y el máximo de palabras por frase que deben ser corregidos. Si no es necesario hacerlo, no se dice nada en la salida.
>
> En la entrada del algoritmo tendremos un texto como secuencia de caracteres. Cada frase termina con una palabra que finaliza, a su vez, con un punto.
>
> Cada palabra está separada de la siguiente por un único espacio. En el caso de que la palabra acabe en punto, el espacio aparece a continuación del punto. En el inicio del texto no hay espacios. Las palabras que deben ser corregidas contienen uno o más signos de asterisco "*". Si la palabra consiste sólo en el carácter asterisco, se trata también de una corrección que es preciso hacer (una palabra que se tiene que añadir, por ejemplo).
>
> Se ha convenido en que el final del texto original se marcará por una frase que no tiene ninguna palabra; simplemente aparecerá el carácter de punto de final de frase.
>
> Por ejemplo, si en la entrada tenemos:
>
> Hola qu* tal. Como B*a to*o. .
>
> En la salida tendremos:
>
> 100 1.5 2

El diseño puede llegar a ser complicado si pensamos en la aplicación a partir de la secuencia de caracteres. En cambio, si lo planteamos proponiendo un conjunto de abstracciones que son naturales y convenientes para el problema, el desarrollo resultará más sencillo de hacer. Aquí nos resulta útil pensar que el texto es una secuencia de frases y proponer un algoritmo que vaya obteniendo y tratando cada una. Después nos ocuparíamos del siguiente nivel, que consiste en plantear el subproblema de cómo obtener una frase, el subproblema de cómo tratarla, y cómo identificar también que se trata de la última frase. La consideración de una frase como una secuencia de palabras nos será conveniente y dará lugar a la creación de los subproblemas dedicados a las palabras, que se resolverán en el último nivel como problemas de secuencia de caracteres.

El uso de las abstracciones *frase* y *palabra* es propio del problema y natural a éste; su correspondencia en un nivel de descomposición hace que nos podamos concentrar en los detalles más concretos de una abstracción, y que dejemos para más adelante los detalles de las abstracciones que estén pendientes de resolver. Debemos procurar que cada nivel tenga un número de detalles que lo haga fácil de seguir y que asegure la corrección de las soluciones planteadas.

1.2. ¿Análisis ascendente o descendente?

El primer nivel de descomposición es el más abstracto, mientras que los siguientes lo son menos o, dicho de otro modo, son más concretos, hasta llegar

al último, que es el que presenta un mayor nivel de concreción. En el caso del ejemplo, el más concreto es la secuencia elemental de caracteres.

A la hora de plantear un problema complejo, tenemos dos posibles direcciones para su análisis:

- Resolver los problemas más concretos, para resolver después los más abstractos. Es decir, vamos del más concreto al más abstracto. Esta forma de plantearlo recibirá el nombre de **análisis ascendente**.

- Resolver el primer nivel más abstracto, para proseguir después con los niveles que lo son menos, hasta llegar al nivel más concreto. Esta segunda dirección recibirá el nombre de **análisis descendente**. Esta dirección es la sugerida en el ejemplo.

El análisis ascendente está repleto de dificultades y no resulta adecuado para llevar a buen término un diseño en un tiempo razonable de desarrollo. La dificultad principal consiste en que, si no se resuelven en primer lugar los niveles más abstractos, no sabemos realmente qué problemas concretos tenemos que solucionar. De este modo, podemos estar resolviendo problemas concretos que después no se utilizarán o que no están convenientemente definidos para ser usados en los niveles más abstractos. Esto llevaría a rediseñar el nivel más concreto y, en consecuencia, a retroceder en el diseño, para después volver al otro nivel. Imaginad, en el ejemplo del que nos estamos ocupando, que no tenéis demasiado claro el tratamiento que debéis aplicar; por este motivo, os dedicáis a desarrollar una acción que lea una palabra y la guarde en una estructura de datos, para después descubrir en el nivel más abstracto que la utiliza que no es necesario el contenido exacto de la palabra, ya que sólo hay que saber si es preciso corregirla, y contarla en el caso de que así sea. La estructura de datos puede sobrar y servir de muy poco al problema, sobre todo si pone limitaciones al enunciado, como veremos más adelante.

El diseño descendente es un método más eficaz y rápido para desarrollar problemas complejos que el ascendente u otro método que combine ambos.

En el análisis descendente, las cosas van mejor. Al empezar en el nivel más abstracto, identificamos y definimos los subproblemas que hay que resolver en el siguiente nivel menos abstracto (fijamos los objetivos para el siguiente nivel de menor abstracción); en consecuencia, no haremos más trabajo del que realmente hace falta para resolver el problema global, por lo que acortaremos el tiempo de desarrollo y evitaremos retrocesos. El ejemplo en el que nos centraremos seguirá esta línea. El inconveniente de esta estrategia, cuando se tiene poca experiencia práctica en el método y en el diseño de algoritmos, es que no se tiene la seguridad de que se pueda resolver más adelante alguno de los subproblemas propuestos en el nivel abstracto; entonces se prefiere plantear el problema de forma ascendente, para tener la seguridad de que se dispone de una base de problemas concretos resueltos que ayudarán a resolver, a su vez, los niveles más abstractos. Sin embargo, si lo hacemos así, iremos saltando de niveles y llegando a confusiones innecesarias. La única forma de asimilar el método

descendente y superar este inconveniente inicial es su práctica en distintos problemas, ya que el análisis ascendente o la combinación del análisis ascendente y descendente hace que el desarrollo sea un proceso más lento, a menudo confuso, disperso y que presente incoherencias en los resultados.

1.3. Orientaciones de las abstracciones

Disponemos de dos orientaciones para hacer abstracciones de un problema o subproblema y, de este modo, guiar su descomposición en subproblemas:

- **Orientación funcional**: en el enunciado del problema se pide un tratamiento muy amplio de detalles y pasos. Descubrimos que este tratamiento amplio se puede descomponer en varias etapas conceptualizadas para que cada una realice una función parcial que contribuya a la solución global del problema. Una vez identificadas estas etapas, procedemos a expresar su composición algorítmica, para más adelante desarrollar cada etapa por separado.

- **Orientación a objetos**: un objeto real del problema no es directamente representable por los tipos elementales del lenguaje algorítmico. En primer lugar definiremos un tipo conveniente para el objeto; a lo largo de todo el diseño, para hacer cualquier cambio de estado o acceso de este objeto, será necesario llevar a cabo unas acciones y/o funciones diseñadas expresamente por el tipo de objeto definido. Es decir, asociamos unas acciones y/o funciones al tipo, y sólo estas acciones y/o funciones serán las que modificarán o consultarán el estado de un objeto del tipo definido.

Podemos encontrar un ejemplo de abstracción funcional en el siguiente problema:

> Se dispone de las temperaturas medias diarias de un mes en una tabla. Se quiere diseñar un algoritmo que nos devuelva el día del mes en estudio que tiene la temperatura más próxima a la media.
>
> Es necesario recorrer toda la tabla para calcular la temperatura media del mes; una vez calculada, se debe buscar el valor de temperatura diaria que más se acerca a ella. De este modo, hemos dividido el problema en dos subproblemas más concretos según su función, que son:
>
> **1)** Cálculo de la media de temperaturas de un mes.
>
> **2)** Búsqueda del día de temperatura más próxima a la media del mes.
>
> Ambos problemas trabajan sobre el mismo objeto, que es una tabla de temperaturas medias diarias de un mes.

En el ejemplo propuesto para ser trabajado a lo largo del apartado, se han hecho abstracciones orientadas a objetos. Hemos interpretado parte de la información implícita que guarda la secuencia de caracteres como frases y palabras. Notad que podemos considerar un texto de distintas formas; por ejemplo, el texto como una secuencia de capítulos de un libro, el capítulo como una secuencia de apartados, etc. Pasamos por alto el hecho de que el texto que nos

dan en la entrada estuviese organizado por párrafos, ya que la consideración de párrafos no tiene ninguna utilidad en la resolución de nuestro problema. La fuerza de la abstracción reside en esta afirmación: sólo consideramos aquello que es relevante para nuestro problema. En cada nivel definiremos la representación que debe tener cada abstracción elegida, y qué acciones y/o funciones asociaremos a cada una.

1.4. Formas de trabajar y presentar el diseño descendente

Ahora necesitamos expresar de algún modo nuestro desarrollo, siguiendo la metodología del diseño descendente. Una forma de hacerlo podría consistir en proponer en el primer nivel un borrador de algoritmo informal.

Observemos qué sucede en nuestro ejemplo si resolvemos el primer nivel. En primer lugar, debemos establecer los objetivos del diseño mediante la especificación, manteniendo el nivel de abstracción que nos hayamos propuesto. En nuestro caso, consiste en imaginarnos el texto como una secuencia de frases.

Dicho de otro modo,...

... planteamos el nivel como si dispusiésemos de un procesador virtual (o un lenguaje virtual) que sabe lo que es una secuencia de frases y cómo obtenerlas y tratarlas.

De este modo, una vez examinado el enunciado, escribimos la siguiente especificación:

{ Pre: en la entrada tenemos una secuencia de frases, *T*, que puede estar vacía. Cada frase de T contiene información para saber si hace falta corregirla o no }
AnalisisCorrecciones
{ Post: en el caso de que no sea necesario corregir *T*, no se escribe nada. En el caso de que *T* sea un texto que haya que corregir, se generará en la salida una secuencia *R*, formada por tres elementos:
R1 Representa la media de palabras que hay que corregir por frase de *T*.
R2 Representa el porcentaje de frases que hay que corregir en *T*.
R3 Representa el máximo de palabras que hay que corregir por frase de *T* }

Se pide un análisis del texto, para elaborar una estadística de las correcciones que se deben llevar a cabo. Hay que hacerlo mediante la aplicación del esquema de recorrido, ya que los datos del análisis del texto dependerán de los datos parciales que aportará cada una de sus frases. Las acciones del análisis, así como la obtención de la secuencia, irían a cargo de nuestro procesador virtual o nuestro lenguaje virtual inteligente. De este modo, si aplicamos el esquema de recorrido, obtenemos informalmente:

algoritmo AnalisisCorrecciones
  var ... **fvar**
  "Inicializar los resultados del análisis del texto."
  "Obtener la primera frase."
  mientras "la frase obtenida no sea la última frase" **hacer**
    "Actualizar los resultados del análisis del texto a partir de la frase actual."
    "Obtener la siguiente frase."
  fmientras
  "Escribir los resultados del análisis del texto."
falgoritmo

En el siguiente nivel, refinaríamos cada una de las acciones o funciones expresadas informalmente con posibles nuevos nombres informales. Seguiríamos

de este modo hasta completar un algoritmo completamente formal y concreto. Podéis notar que esta forma de expresar la resolución tiene el inconveniente de que en cada nivel debemos reescribir el algoritmo y al final del desarrollo, si tenemos muchos niveles, podemos encontrarnos con un algoritmo muy largo y muy difícil de comprender a causa de los numerosos detalles que abarca.

Esta técnica de refinamiento...

... es la que hemos utilizado para aplicar los esquemas.

Una forma más adecuada de expresar nuestro diseño se basa en las herramientas que nos proporciona la notación algorítmica. Podemos encapsular mediante acciones y/o funciones parametrizadas cada subproblema a resolver. Por otro lado, podemos encapsular cada objeto complejo mediante los constructores de tipos, o declarando nuevos tipos (enumerativos, etc.). De este modo, en cada nivel escribimos ya un algoritmo y una acción y/o función con total precisión y formalidad, y no reescribimos las partes ya resueltas del diseño. El uso de parámetros nos permitirá alcanzar la máxima independencia entre niveles y entre los subproblemas planteados, pues nos concentraremos en una parcela limitada del problema global.

La especificación de acciones y/o funciones tendrá una doble utilidad: por un lado, servirá de puente entre niveles para verificar la corrección de las soluciones planteadas, asumiendo que las acciones y/o funciones utilizadas en el nivel de estudio cumplen la especificación dada, y por otro lado, la especificación de una acción y/o función es en sí el enunciado formalizado del subproblema que hay que resolver en el nivel que corresponda. Si en el diseño de la acción y/o función se respeta la especificación, el diseño global funcionará como se espera.

Reflejaremos el análisis descendente del problema mediante los mecanismos de encapsulado que nos ofrece la notación algorítmica (definición de tipos, constructores de tipos y acciones y/o funciones).

Siguiendo nuestro ejemplo de trabajo, el resultado del primer nivel queda de la siguiente forma:

```
algoritmo AnalisisCorrecciones

    var
        ResultadosAnalisisTexto: tDatosAnalisis;
        Frase: tFrase;
    fvar

    { Pre: en la entrada tenemos una secuencia de frases, T, que puede estar vacía. Cada
      frase de T contiene información para saber si es necesario corregirla o no }

    InicioResultadosAnalisis(ResultadosAnalisisTexto);
    ObtenerFrase(frase);

    mientras no UltimaFrase(Frase) hacer
        ActualizaResultadosAnalisis(frase, ResultadosAnalisisTexto);
        ObtenerFrase(frase)
    fmientras
```

En esta solución...

... sólo aparecen los comentarios de la precondición y la postcondición. La elección de unos nombres apropiados a las acciones, funciones y variables hace que no sean necesarios más comentarios, si después especificamos cada una de las acciones y/o funciones. Esto lo veremos más adelante, y completará el nivel actual de trabajo.

```
    escribirResultadosAnalisis(ResultadosAnalisisTexto);
    { Post: en el caso de que no sea necesario corregir T no se escribe nada. En el caso de
      que T sea un texto por corregir, se generará en la salida una secuencia R formada
      por tres elementos:
    R1 representa la media de palabras que hay que corregir por frase de T.
    R2 representa el porcentaje de frases que hay que corregir en T.
    R3 representa el máximo de palabras que hay que corregir por frase de T }
falgoritmo
```

Observad que la última solución presentada por el primer nivel tiene la particularidad de ser sintácticamente correcta. Sin embargo, no es completa porque falta por definir el tipo *tFrase* y un conjunto de acciones y/o funciones (*obtenerFrase*, y *ultimaFrase*), y el tipo *tDatosAnalisis* con sus acciones (*inicioResultadosAnalisis*, *actualizaResultadosAnalisis*, y *escribirResultadosAnalisis*), que se deberán definir en un segundo nivel.

A pesar de que no es completa, hay otra ventaja que podemos aprovechar: podemos razonar la corrección de la solución hasta ahora formulada sin necesidad de conocer los detalles de los tipos aún no definidos, ni de cómo las acciones y/o funciones realizarán el efecto que se espera de ellas, que todavía están por desarrollar. La especificación de cada una de las acciones o funciones que hay que desarrollar nos indica cuál es su efecto; a partir de aquí podemos razonar la corrección de la solución formulada. Cuando se desarrolle el siguiente nivel, si las acciones o funciones desarrolladas respetan sus precondiciones y postcondiciones, se mantendrá la corrección del nivel actual. La especificación juega aquí un papel importante de puente entre niveles, ya que ayuda a separar el qué del cómo.

La corrección de la solución nos viene asegurada por el hecho de que hemos aplicado el esquema de recorrido a una secuencia de frases. En la solución propuesta hemos considerado una última frase que actúa como marca de final de secuencia. Es preciso averiguar si esta frase es sólo una marca o si es al mismo tiempo una frase que se debe tratar y una marca.

Uno de los puntos...

... fuertes de los esquemas reside en que conocemos todos los detalles de su funcionamiento o comportamiento, que nos sirven de base para razonar cómodamente sobre la corrección del algoritmo o fragmento de algoritmo resultante de la aplicación de los esquemas.

Si examinamos el enunciado, observamos que al final del texto tendremos dos caracteres ".". El primer punto pertenece a la última frase "real" del texto. El segundo punto puede verse como un punto que cierra una frase vacía (no tiene ninguna palabra) y sólo sirve para marcar el final del texto. La frase vacía indicará el fin de la secuencia de frases; por lo tanto, no es necesario tratarla (analizarla, en este caso). Sólo hay que analizar cada frase leída no vacía, acumulando los resultados de su análisis en el objeto *ResultadosAnalisisTexto*. Cuando salgamos de la iteración **mientras**, habremos analizado todas las frases "reales" del texto (es decir, con caracteres y, por lo tanto, con contenido), y *ResultadosAnalisisTexto* tendrá el resultado del análisis global del texto.

Además, el hecho de que el algoritmo no entre en detalles del tipo no definido, pues la responsabilidad de estos detalles pertenece a las acciones y/o funciones que tratan el tipo, hace que la definición concreta de *tFrase* sea irrelevante

La orientación a objetos nos facilita una separación de los problemas muy efectiva para descomponerlos en subproblemas independientes.

para razonar la corrección de esta etapa de diseño. En el segundo nivel deberemos definir los tipos *tFrase*, para trabajar con los detalles que creamos convenientes. Podremos estudiar la mejor alternativa, que no tendrá ningún efecto sobre el primer nivel, siempre que se respete el significado, implícito en estos momentos, de las acciones y/o funciones que utiliza el algoritmo del primer nivel, y que se desarrollarán en el segundo. Podremos comprobar que cada función o acción haga lo que tiene que hacer, sin tener que retroceder a etapas anteriores del diseño.

Antes de proseguir con el problema que nos ilustra este apartado, vamos a hacer otras observaciones. La notación algorítmica nos permite definir acciones y/o funciones y encapsular subproblemas, tal y como hemos hecho en la última solución del ejemplo. Los constructores de tipos y la declaración de nuevos tipos nos permiten encapsular las abstracciones de datos. De este modo, la notación nos da todos los elementos necesarios para poder reflejar en el diseño las abstracciones que hemos realizado y la metodología seguida en las descomposiciones. Si seguimos las pautas, también podremos evitar reescribir o modificar cada una de las etapas previas.

1.5. Herramientas de la notación para abstraer datos

Comencemos por hablar sobre cómo encapsular nuestras abstracciones de datos. Por ejemplo, intentemos definir la abstracción de frase de nuestro ejemplo. Podemos ver la frase como una secuencia de palabras o vocablos, y también como una secuencia de caracteres. Imaginemos que en la aplicación que estamos desarrollando nos piden detectar frases capicúas. Cuando llegásemos al nivel que trata la abstracción de frase, nos daríamos cuenta de que nos hace falta guardar toda la frase para comparar el último carácter con el primero, el penúltimo con el segundo, etc., aplicando un esquema de búsqueda dentro de una función que señalaría si una frase es capicúa o no. Para definir los tipos, podemos utilizar los constructores de tipos que nos ofrece el lenguaje: tablas y tuplas.

Por ejemplo, la frase "Dábale arroz a la zorra el abad" es capicúa.

En nuestro caso, puesto que la frase consiste en datos homogéneos (caracteres), utilizaríamos las tablas. Por ejemplo:

```
tipo
    tFrase = tabla [maxCarFrase + 1] de caracter;
ftipo
```

donde *maxCarFrase* es el máximo de caracteres que preveíamos que tendría cada frase. El elemento de más es para el centinela, ya que *a priori* no sabemos

el número de caracteres. Dado que nos interesa comparar el último con el primero, etc., siempre nos puede ser más útil tener la longitud de la frase en lugar del centinela. Sin embargo, si declaramos lo siguiente:

```
var
  ncaracteresFrase: entero;
  Frase: tFrase;
fvar
```

estamos dispersando la abstracción de frase en dos conceptos: su longitud y la frase en sí. Esta dispersión puede parecer ridícula, pero si después nos vemos obligados a guardar más de una frase, deberemos declarar más contadores de caracteres para las diferentes tablas. Además, tendríamos que cambiar los parámetros de las acciones y/o funciones en las que no lo hayamos previsto, para considerar los nuevos contadores. Es decir, vamos de un nivel a otro reescribiendo el código ya resuelto, y esto no nos interesa. Lo mejor es encapsular completamente todo lo que haga referencia a la abstracción que estamos implementando: 

```
tipo
    tFrase = tupla
                nCaracteres: entero;
                c: tabla [maxCarFrase] de caracter;
             ftupla
ftipo
```

De este modo, tenemos dentro del tipo declarado los detalles que nos interesan de la abstracción que trabajamos; en el nivel más abstracto no es necesario hacer ningún cambio, ya que los detalles serán trabajados en el nivel de la abstracción de frase.

Imaginemos ahora que lo que nos piden es averiguar si una frase del texto tiene las mismas palabras que una frase dada, pero con distinto orden. De forma más breve, si podemos decir que una frase es permutación de otra. Sin entrar en detalles del problema, parece claro que es necesario ver la frase como una secuencia de palabras; por lo tanto, la anterior definición de *tFrase* no nos serviría, ya que necesitamos estructurar la frase en una colección de palabras para compararlas entre sí. Una estructura que no resulta nada útil es la siguiente:

Por ejemplo, la frase "La mar estaba serena" es una permutación de la frase "serena estaba la mar".

```
tipo
    tFrase = tupla
                nPalabras: entero;
                c: tabla [maxPalabrasFrase, maxCarPalabra + 1] de caracter;
             ftupla
ftipo
```

donde *maxPalabrasFrase* es el número máximo de palabras que puede tener una frase, *maxCarPalabra* es el máximo de caracteres que puede tener una palabra, y cada una de las filas de la matriz de caracteres guarda un centinela para cada palabra. A pesar de que la información esté estructurada, es críptico tratar esta estructura porque no refleja del todo la abstracción de palabra que nos conviene para resolver de forma más cómoda el nuevo problema, sin dispersión de abstracciones. En cambio, la siguiente definición es mucho más apropiada y menos problemática a la hora de tratarla:

```
tipo
    tFrase = tupla
                nPalabras: entero;
                m: tabla [maxPalabrasFrase] de tPalabra;
             ftupla
ftipo
```

Donde *tPalabra* se definirá en el nivel correspondiente a las acciones y funciones asociadas que le hagan falta.

Pensad la solución antes de continuar.

De este modo, podemos observar que la abstracción de datos se puede hacer mediante los constructores de tipos, pero no es necesario olvidar que en la abstracción reunimos aquello que nos interesa y que nos conviene porque es relevante para el problema que estamos resolviendo. ¿Habéis encontrado ya la solución que debe aplicarse al tipo *tFrase* de nuestro problema inicial?

Pues aquí la tenéis:

```
tipo
    tFrase = tupla
                nPalabrasC: entero;
                final: booleano;
             ftupla
ftipo
```

Para nuestro ejercicio inicial, todo lo que nos interesa de una frase es saber el número de palabras que tiene por corregir, además de saber también si se trata de la frase final. De este modo, la tupla definida cumple los objetivos que nos hemos propuesto para el problema.

Tal vez penséis que alguno de los tipos en los que se guarda el contenido de la frase también sirve para nuestro objetivo, ya que después podemos desarrollar las acciones y/o funciones que contabilizan el número de caracteres y palabras a partir del cual está guardado. Sin embargo, si pensamos así nos olvidamos de los objetivos que perseguimos al llevar a cabo un diseño. En primer lugar, tenemos una razón de sentido común: ¿por qué debemos guardar cosas que después no utilizaremos? La abstracción de datos consiste en simplificar los detalles del objeto sometido a estudio para poner aquellos datos que son relevantes para el

problema, descartando todo lo que no resulta útil. Por ejemplo, si hiciésemos un estudio consistente en una estadística de estaturas de un grupo de individuos, ¿nos serviría de algo el color de sus ojos para la finalidad del estudio?

También hay razones técnicas que ayudan a los objetivos de diseño que perseguimos. En primer lugar, es necesario llevar a cabo un algoritmo que cumpla los objetivos propuestos (postcondición) por el enunciado, a partir de las condiciones iniciales que establezca (precondición). En la entrada tenemos un texto que es una secuencia de caracteres. ¿Cuántos caracteres y cuántas palabras puede haber en una secuencia? En principio, no lo sabemos. Si declaramos tablas, debemos definir máximos y, en consecuencia, restringir la entrada (texto) del enunciado a condiciones que, si se satisfacen, harán que el algoritmo funcione correctamente. Si no se satisfacen, debemos modificar las constantes del algoritmo. Todo este embrollo sólo sirve para formular un algoritmo que no será del todo correcto, pues no admite cualquier frase, como pide el enunciado. Además, para contar los vocablos de una frase, no nos hace falta tener previamente la frase guardada y, por lo tanto, es un capricho poner restricciones en el enunciado que, en el fondo, no son necesarias. Debemos ajustarnos al enunciado, y no al revés.

Otra razón es la cuestión de eficiencia: guardar una frase para nuestro problema ocupa un espacio de memoria innecesario, así que abusamos sin justificación de dicho recurso.

En los problemas sugeridos anteriormente de la frase capicúa y la permutación de frases tiene sentido que la frase se guarde, ya que para resolverlos nos hace falta el acceso directo que nos proporcionan las tablas. En estos problemas es correcto exigir límites al enunciado donde sea necesario, y el espacio que ocupan los datos está plenamente justificado por su necesidad.

1.6. Herramientas de la notación para abstraer código

Una vez resuelta la abstracción de datos en el nivel correspondiente, es necesario diseñar las acciones y/o funciones que se han propuesto en el nivel anterior al hacer una abstracción de código. De hecho, el primer nivel todavía se debe completar dando significado a cada una de las acciones y/o funciones que se han propuesto. El medio es la especificación, y de este modo establecemos un puente entre niveles que asegura la coherencia entre sí. Para la frase tenemos:

accion obtenerFrase(**sal** f: tFrase)
{ Pre: en la entrada tenemos un texto *T* representado como una secuencia de frases *t*. En la parte derecha de *t*, *DT*, hay una secuencia de frases o la frase vacía }
{ Post: en *f* tenemos representada la primera frase de *DT*. La frase representada está en la parte izquierda de *t* }
funcion ultimaFrase(f: tFrase): **booleano**
{ Pre: *f* = *F* }
{ Post: ultimaFrase(*F*) es **cierto** si *F* es una frase vacía. Retornará **falso** en caso contrario }

Para el análisis del texto, a partir del objeto de tipo *tAnalisisDatos*, debemos centrarnos en la postcondición del problema. Recordad que debemos obtener el porcentaje de frases que hay que corregir, y la media de palabras y el máximo de palabras a corregir por frase. Los dos primeros datos se obtienen a partir de los elementos siguientes:

- El número de frases del texto (*nfrases*).
- El número de frases que hay que corregir (*nfrasesC*) del texto.
- El número de palabras que hay que corregir del texto (*npalabrasC*).

Hemos puesto entre paréntesis unos nombres que nos servirán para abreviar la especificación. Abreviaremos también el máximo de palabras por frase mediante el nombre *mx*. Si seguimos estos nombres, la especificación nos queda de la siguiente forma:

accion InicioResultadosAnalisis(**sal** a: tDatosAnalisis)
{ Pre: }
{ Post: en *a* tenemos *nfrases*, *nfrasesC*, *npalabrasC* y *mx* de un texto vacío }

accion actualizaResultadosAnalisis(**ent** f: tFrase; **entsal** a: tDatosAnalisis)
{ Pre: *a* = *A* y *f* = *F* }
{ Post: en *a* tenemos *nfrases*, *nfrasesC*, *npalabrasC* y *mx* del texto que resulta de añadir el análisis de la frase *F* al texto ya analizado en *A* }

accion escribirResultadosAnalisis(**ent** A: tDatosAnalisis)
{ Pre: a = A }
{ Post: si los datos de *A* son de un texto a corregir, se escribe por el canal de salida una secuencia *R* formada por tres elementos:
R1 representa la media de palabras a corregir por frase presente en el texto analizado en *A*.
R2 representa el porcentaje de frases a corregir en el texto analizado en *A*.
R3 representa el máximo de palabras a corregir por frase existente en el texto analizado en *A* }
En el caso de que *A* tenga los datos de un texto vacío, no se escribe nada }

Notad que mediante la especificación establecemos la separación entre niveles:

1) Con la especificación en la mano podemos comprobar si el primer nivel es correcto, sin tener que saber los detalles internos de cada acción o función.

2) Cada tripleta (encabezamiento acción o función)-(precondición)-(postcondición) es el enunciado formalizado de un subproblema que hay que resolver en el segundo nivel, y marca los objetivos que se tienen que trabajar.

Notad que...

... la especificación también nos permite dividir entre un grupo de personas el trabajo de la aplicación que hay que desarrollar, de forma que cada persona no dependa de la otra en el momento de llevar a cabo el desarrollo.

De este modo, las acciones y funciones de la notación algorítmica nos permiten encapsular los subproblemas que necesitamos, sin que tengamos que reescribir el algoritmo, ya que una vez que se hayan diseñado las acciones y funciones que precisamos, el diseño estará completo. Como herramientas de abstracción de

código, permiten separar el qué del cómo en el planteamiento de un nivel. Es decir, permiten escribir en el lugar propicio del texto del algoritmo qué queremos mediante el nombre de una acción o función con los parámetros adecuados, sin pensar en cómo la acción o función llegará a cumplir su propósito. De esta forma podemos mantener las abstracciones que consideremos convenientes en el nivel del que nos ocupamos.

Notad también que la parametrización de acciones y funciones es la responsable de que los entornos de cada problema o subproblema estén convenientemente separados. De hecho, los parámetros nos ayudan a formalizar las entradas y salidas del subproblema en cuestión. Con los parámetros establecemos el contacto con el exterior de la acción o función. Mediante los parámetros actuales usamos la acción en el entorno que hemos decidido que se debe modificar y/o consultar. Con los parámetros formales diseñamos la acción o función sin pensar quién la utilizará, y simplemente decidimos qué parámetros nos son necesarios para llegar al objetivo propuesto por el subproblema.

Con los parámetros...

... las acciones o funciones se pueden ver como cajas negras que interactúan con el exterior a partir de una entrada de datos, y que producen datos en la salida. Lo que hay dentro de la caja no importa cuando la usamos. Además,la podemos mover a cualquier lugar, porque aparte de sus propios parámetros no depende de ningún otro elemento externo para su funcionamiento interno.

1.7. Trabajando con profundidad el ejemplo

Pasemos a resolver el siguiente nivel de descomposición del problema, en el que nos enfrentaremos a dos abstracciones: las frases y los resultados del análisis del texto que hemos conceptualizado como objetos del problema resuelto en el primer nivel.

Empezamos por los resultados del análisis del texto. Si queremos saber el porcentaje de frases a corregir, nos es necesario saber también cuántas frases hay en el texto, y cuántas deben ser corregidas. Nos hará falta, por lo tanto, dos contadores (*nfrases*, *nFrasesCor* respectivamente) para calcular el porcentaje cuando hayamos acabado de procesar el texto.

Si queremos saber la media de palabras por frase corregida, necesitaremos un contador de las palabras a corregir, que, junto con el número de frases a corregir, nos proporcionará la media. También necesitaremos registrar el máximo de palabras que hay que corregir por frase. Encapsulemos los datos:

```
tDatosAnalisis = tupla
                     nFrases: entero;
                     nFrasesCor: entero;
                     nPalabrasPorCorregir: entero;
                     maximoPalabrasPorCorregir: entero;
                 ftupla
```

El diseño de las acciones, de acuerdo con las especificaciones que perseguimos, es el siguiente:

```
accion inicioResultadosAnalisis(sal A: tDatosAnalisis)
    A.nFrases := 0;
    A.nFrasesCor := 0;
    A.nPalabrasPorCorregir := 0;
    A.maximoPalabrasPorCorregir := 0
faccion
accion actualizaResultadosAnalisis(ent F: tFrase; entsal A: tDatosAnalisis)
    var
        nPalabrasPorCorFrase: entero;
    fvar
        A.nFrases := A.nFrases + 1;
        nPalabrasPorCorFrase := palabrasPorCorregir(F);
        si nPalabrasPorCorFrase > 0 entonces
            A.nFrasesCor := A.nFrasesCor + 1;
            A.nPalabrasPorCorregir := A.nPalabrasPorCorregir + nPalabrasPorCorFrase
        fsi
        si nPalabrasPorCorFrase > A.maximoPalabrasPorCorregir entonces
           A.maximoPalabrasPorCorregir := nPalabrasPorCorFrase
        fsi
faccion
accion escribirResultadosAnalisis(ent A: tDatosAnalisis)
    si A.nFrasesCor > 0 entonces
         escribirReal(enteroAReal(A.nFrasesCor) / enteroAReal(A.nFrases) * 100.0);
         escribirReal(enteroAReal(A.nPalabrasPorCorregir) / enteroAReal(A.nFrasesCor));
         escribirEntero(A.maximoPalabrasPorCorregir)
    fsi
faccion
```

Notad que debemos prever el error de dividir por 0 en la última acción. El enunciado pide que no se genere salida si no hay errores.

En la acción *actualizaResultadosAnalisis* se utiliza la función *palabrasPorCorregir* que añadiremos al grupo de acciones y/o funciones del tipo *tFrase*. A pesar de que, por razones de exposición, ya habíamos adelantado el tipo *tFrase*, la función nos resulta útil para mantener la independencia entre las dos abstracciones: para resolver los resultados del análisis del texto no nos hace falta conocer los detalles de cómo se representa una frase; de esta forma, cuando resolvemos la abstracción de frase, llevaremos a cabo esta acción independientemente de lo que se ha hecho en la abstracción de datos del análisis. Si hacemos modificaciones en *tFrase* (con los respectivos cambios que hay que realizar en su conjunto de acciones y funciones), no tendremos que examinar el código referente a la abstracción de datos de análisis. La especificación de la función será la siguiente:

```
funcion palabrasPorCorregir(f: tFrase): entero;
    { Pre: f = F }
    { Post: palabrasPorCorregir(F) nos da el número de palabras por corregir en la frase F }
```

Vamos a resolver el segundo nivel de abstracción: la abstracción de frases. Empezaremos por el subproblema de obtener una frase. La mejor forma de plantear su obtención es considerar que, en la entrada, la frase que se debe obtener es una secuencia de palabras:

$$m_1 \ m_2 \ m_3 \ m_4 \ ...$$

Decidimos hacer la abstracción de palabra para resolver este nivel. Por lo tanto, expresaremos todas las soluciones de este nivel en términos de palabras, si hace falta, y más adelante ya trabajaremos los detalles de los palabras en otra etapa.

Al reconocer una secuencia del problema, sea ésta real o abstracta, es necesario identificar el primer elemento, cómo se obtiene el siguiente y qué elemento es el último o marca el final de la secuencia. Al repasar el enunciado, observamos que no hay ninguna diferencia entre obtener la primera palabra y obtener la siguiente; también observamos que la propiedad que identifica la última palabra es que finaliza con un punto. En nuestro caso, el mantenimiento de la abstracción de palabra convierte la identificación de las operaciones que hay que efectuar en la secuencia en abstracciones de código, como veremos a continuación.

Reconocemos la obtención de una frase como un problema de recorrido, consistente en contar el número de palabras por corregir; esto nos lleva al siguiente borrador que hay que completar:

```
accion obtenerFrase(sal f: tFrase)
   var m: tPalabra;
   fvar
   "Iniciar  el tratamiento"
   obtenerPrimeraPalabra(m);
   mientras no ultimaPalabra(m) hacer
     "Tratar el elemento" { tratamiento de la palabra m }
     obtenerPalabra(m)
   fmientras
   "Tratamiento final"
faccion
```

Observamos también que la última palabra no sólo marca el final de la frase, sino que resulta válida para tratar, al igual que las demás. Esta posibilidad no está contemplada en el esquema de recorrido que deberemos modificar convenientemente para que sea tenida en cuenta.

Debemos prever la posibilidad de que la frase esté vacía. Es decir, que sólo contenga el punto. En este caso, tendremos que contar con la palabra vacía, que no tiene ningún carácter y acaba en punto.

El tratamiento que hay que aplicar para cada palabra consiste en saber si hay que corregir o no aquélla a la que se ha accedido, y en el caso de que así sea, contabilizarla para saber cuántas palabras debemos corregir en la frase. Por otro lado, contamos el número de palabras que hay en la frase para saber si ésta está vacía o no. Debemos recordar que la frase vacía (y, por lo tanto, que contiene una palabra vacía) representa el final del texto y no forma parte del mismo. Obviamente, la palabra vacía no es una palabra real, y por ello no hay

que contarla. Tendremos que distinguir si la última palabra está vacía o no, para hacerle, en consecuencia, un tratamiento distinto. La solución queda de la siguiente forma:

```
accion obtenerFrase(sal f: tFrase)
    var m: tPalabra; nPalabras: entero;
    fvar
    { Pre: en la entrada tenemos un texto T representado como una secuencia de frases t. En
      la parte derecha de t, DT, hay una secuencia de frases o la frase vacía }
    nPalabras := 0;
    f.nPalabrasC := 0;
    obtenerPalabra(m);
    mientras  no ultimaPalabra(m) hacer
        si hayQueCorregirPalabra(m) entonces f.nPalabrasC := f.nPalabrasC + 1 fsi
        nPalabras := nPalabras + 1;
        ObtenerPalabra(m)
    fmientras
    si no palabraVacia(m) entonces
        si hayQueCorregirPalabra(m) entonces f.nPalabrasC := f.nPalabrasC + 1 fsi
        nPalabras:= NPalabras + 1
    fsi
    f: final := nPalabras = 0
    { Post: en f tenemos representado la primera frase de DT. La frase representada está en la
      parte izquierda de t }
faccion
```

Fijaros que...

... en la última estructura condicional (**si** … **fsi**) estamos tratando la palabra que actúa como marca.

La función *hayQueCorregirPalabra* nos dirá si la palabra *m* debe corregirse o no. Por el momento, la tarea que hay que realizar en el siguiente nivel es:

```
accion obtenerPalabra(sal m: tPalabra)
    { Pre: en la entrada tenemos la secuencia de palabras, t, del texto T, de la que sólo po-
      demos obtener su parte derecha DT, y ésta no ha llegado al final del texto }
    { Post: m representa la primera palabra que hay en DT. La palabra obtenida estará en
      la parte izquierda de la secuencia t }

funcion ultimaPalabra(m: tPalabra): booleano
    { Pre: m = M }
    { Post: ultimaPalabra(M) es cierto si m representa la última palabra de una frase }

funcion palabraVacia(m: tPalabra): booleano
    { Pre: m = M }
    { Post: palabraVacia(M) es cierto si m representa una palabra vacía }

funcion hayQueCorregirPalabra(m: tPalabra): booleano
    { Pre: m = M }
    { Post: hayQueCorregirPalabra(M) es cierto si la palabra M debe ser corregida. hayQue-
      CorregirPalabra(M) es falso si no hace falta corregir M }
```

El resto de los subproblemas de la abstracción frase son evidentes; se muestran a continuación las respectivas soluciones:

```
funcion ultimaFrase(f: tFrase): booleano
retorna f.final
ffuncion
    { Pre: f = F , y F es una frase válida }
    { Post: ultimaFrase(F) es cierto si F es una frase vacía. Retornará falso en caso contrario }

funcion palabrasPorCorregir(f: tFrase): entero
retorna f.nPalabrasC
ffuncion
    { Pre: f = F }
    { Post: palabrasPorCorregir(F) nos da el número de palabras que hay que corregir para la
      frase F }
```

Os pueden parecer exageradas estas funciones excesivamente breves, pero la ventaja es clara: puesto que el primer nivel no se preocupa de los detalles de *tFrase*, no se deberá modificar nunca si en el segundo nivel elegimos otra representación para *tFrase* y diseñamos las acciones y funciones de acuerdo con la nueva representación. Sólo hemos cambiado el nivel donde concentramos la abstracción, sin cambiar el resto de los niveles, donde están las otras abstracciones sin concretar. Por lo tanto, las posibles modificaciones que hay que realizar en el algoritmo sólo tienen lugar en las abstracciones o la abstracción cuya representación nos hemos propuesto cambiar.

Pasemos ahora al tercer nivel correspondiente a la abstracción de palabras. Podemos considerar una palabra como una secuencia de caracteres acabada en blanco, a excepción de la última de la frase, que acaba en punto antes de llegar al espacio. En este nivel, entonces, la expresaremos en términos de caracteres, que afortunadamente son entendidos por nuestro lenguaje algorítmico; se trata, por lo tanto, del último nivel de abstracción, ya que el diseño estará ya suficientemente concretado.

Notad que...

... expresar las soluciones que tratan una palabra en términos de caracteres es conveniente para el problema que estamos trabajando, pero no en general. Por ejemplo, si tuviésemos que contar las sílabas de una palabra o un vocablo, tal vez sería conveniente considerar esta palabra como una secuencia de sílabas y no de caracteres.

Lo que nos interesa de la palabra aquí es saber si sería necesario corregirla o no. Por lo tanto, tenemos que detectar la presencia del carácter "*" una vez para saber si la palabra se debe corregir o no. Si la palabra consiste sólo en el carácter "*", se interpretará que hace falta añadir una palabra para corregir la frase. Por otro lado, también nos interesa conocer si se trata de una palabra final válida o es vacía. Por lo tanto:

```
tipo
   tPalabra = tupla
                  final: booleano;
                  vacio:  booleano;
                  hayQueCorregir: booleano;
              ftupla
ftipo

{ Pre: en la entrada tenemos la secuencia de palabras, t, del texto T, de la que sólo podemos obtener su parte derecha DT, y ésta no ha llegado al final del texto }

accion obtenerPalabra(sal m: tPalabra)
    var
       c: caracter;
        caracteresLeidos: entero;
    fvar
    caracteresLeido := 0;
    c := leerCaracter();
    mientras c ≠ '*' y  c ≠ ' ' y  c ≠ '.' hacer { buscamos asterisco }
          caracteresLeidos := caracteresLeidos + 1;
          c := leerCaracter()
    fmientras
    m.hayQueCorregir := c = '*';
    mientras c ≠ ' ' y  c ≠ '.' hacer { pasamos resto palabra }
          caracteresLeidos := caracteresLeidos + 1;
          c := leerCaracter()
    fmientras
    m.vacio := caracteresLeidos = 0;
    m.final := c = '.';
faccion

{ Post: m representa la primera palabra que hay en DT. La palabra obtenida estará en la parte izquierda de la secuencia t }
```

El resto de funciones son obvias:

```
funcion ultimaPalabra(m: tPalabra): booleano
    retorna m.final o m.vacio
ffuncion

funcion palabraVacia(m: tPalabra): booleano
    retorna m.vacia
ffuncion

funcion hayQueCorregirPalabra(m: tPalabra): booleano
    retorna m.hayQueCorregir
ffuncion
```

Con esto llegamos al final de la solución del ejercicio. Habréis notado la cantidad de detalles que han sido necesarios para llegar a la solución. El análisis ha sido complicado a causa de los detalles que se debían tener en cuenta, sobre todo en la etapa en que trabajábamos con frases como secuencias de palabras. Sin embargo, la separación de los problemas nos ha ayudado a concentrarnos de forma más cómoda en estos detalles, algo que no hubiese ocurrido si hubiéramos tenido en cuenta todos los detalles del problema al mismo tiempo.

1.8. Valoración final de la metodología

¿Realmente es tan útil el método propuesto? Si lo hubiésemos pensado sólo en caracteres, sin utilizar ninguna abstracción ni separación de problemas, podríamos haber llegado a una solución como ésta:

```
algoritmo churro

    var
      c: caracter;
      N1, N2, N3, N4, Mx: entero;
      FF, A, DP: booleano;
    fvar
    N1 := 0;
    Mx := 0;
    N2 := 0;
    N3 := 0;
    N4 := 0;
    A := falso;
    FF := cierto;
    c := leerCaracter();
    mientras (no FF) o (c ≠ '.') hacer
    si (c ≠ ' ') entonces
      FF := falso;
      si no A entonces
           A := c = '*'
      fsi
      c := leerCaracter();
      si (c = ' ') entonces
           si A entonces
                N2 := N2 + 1;
                A := falso
           fsi
      DP := c = '.';
      c := LeerCaracter()
      fsi
      sino
        DP := c = '.';
        c := LeerCaracter()
      fsi
```

```
        si DP entonces
            FF := cierto;
            N3 := N3 + 1;
            si N2 > 0 entonces
               N4 := N4 + 1
            fsi
            N1:= N1 + N2;
            si N2 > Mx entonces
               Mx := N2
            fsi
            N2 := 0
        fsi
        fmientras
        si N3 > 0 entonces
           escribirReal(enteroAReal(N4) / enteroAReal(N3) * 100.0);
           escribirReal(enteroAReal(N1) / enteroAReal(N4));
           escribirEntero(Mx);
        fsi
falgoritmo
```

Ahora la discusión consiste en saber si vale la pena llegar a este tipo de solución.

Es conveniente que os preguntéis:

- ¿Os es fácil tener la certeza de que esta solución funciona sin probarla en el ordenador, e incluso probándola?
- ¿Creéis que, al ser un texto más corto, debe ser más eficiente cuando sea procesado?
- ¿Creéis que, al ser un texto más corto, harán falta menos horas de desarrollo?
- ¿Creéis que con muchos comentarios sería más inteligible que la solución anterior, basada en el método que hemos propuesto?
- ¿Creéis que es fácil de modificar, de mantener o de entender por alguien o por vosotros mismos después de un tiempo?

Esta serie de preguntas, en el caso de que consideréis lo que hemos ido proponiendo a lo largo del curso, tiene una respuesta única: no.

Los numerosos detalles que aparecen en el algoritmo lo hacen muy poco práctico para que nos convenzamos de que funciona. Debemos estar muy concentrados y atentos para estar seguros: ¿avanza bien la secuencia?, ¿acabará bien? ¿están bien las inicializaciones?, etc. Si alguien propone un algoritmo de este tipo posiblemente lo haya ido probando en el ordenador, modificándolo continuamente desde una versión inicial, hasta que algunas pruebas le han dado la confianza de que es correcto, pero no la absoluta certeza. "¿Qué prueba le falta?", es su duda. Se habrán hecho múltiples relecturas (entre otras cosas, podría

Pensad...

... además, que es más fácil realizar pruebas si el problema está descompuesto en acciones y/o funciones para que podamos comprobar qué tarea realiza cada función.

ser más explícito con el nombre de las variables, por ejemplo), intentando verlo parcialmente a partir de algunas abstracciones que no están expresadas.

A pesar de todo, la confusión puede existir por el hecho de que las abstracciones no están separadas. Cuando más adelante (una semana o más) haya que revisarlo por el motivo que sea, se deberá hacer de nuevo este ejercicio olímpico de concentración. Si se ha comentado ampliamente, seguro que se encontrará que algunos comentarios despistan, porque no se ha seguido un orden concreto, así como tampoco se han separado ni mantenido las abstracciones que en su momento se han realizado para entender el algoritmo. Además, con comentarios, el algoritmo "churro" podría ser tan largo como el diseño hecho metodológicamente. La mayoría de las veces, un desarrollo de este tipo es muy lento, a pesar de que el texto final del algoritmo sea relativamente corto.

En ocasiones, la intuición nos juega malas pasadas. La longitud del texto en el que se expresa un algoritmo no tiene correlación con su posible eficiencia en tiempo al ser procesado. El coste de introducir encapsulado (acciones y/o funciones, constructores, etc.) es mínimo y despreciable. Recordad que otro de los objetivos de los algoritmos es que puedan ser leídos fácilmente, ya que el coste de mantenimiento y mejoras se puede disparar en una empresa si otros programadores se ven obligados a perder el tiempo leyendo algoritmos confusos de programadores no metódicos. Por ejemplo, la editorial nos acaba de avisar que se ha introducido un nuevo carácter corrector, el "/", que sirve para indicar que las palabras que lo tengan no se cuenten como palabras por corregir, a pesar de que vayan acompañadas de asterisco. ¿Dónde haréis los cambios de forma más cómoda, en el algoritmo "churro" o en el que hemos desarrollado siguiendo el método de diseño descendente?

A lo largo de todo el apartado nos hemos ocupado de un ejemplo. Una aplicación real presenta una complejidad mucho mayor, y todas las observaciones hechas son todavía más evidentes; por ello se hace más necesario trabajar con la metodología que aquí se propone.

1.9. Últimas observaciones

Para asimilar la metodología es necesario practicar mucho y vencer los miedos iniciales. En cada ejercicio que hagáis, tenéis que potenciar vuestra capacidad intelectual para encontrar las abstracciones más convenientes al problema, así como también potenciar vuestra capacidad de expresión (la elección de unos buenos nombres en las definiciones forma también parte del juego). El resto es verificar que la composición de sentencias y expresiones del algoritmo y de las acciones y/o funciones que diseñáis sigue la lógica correcta y prevista, de modo que ampliéis las técnicas más trabajadas a lo largo de la asignatura.

Resumen

En este último módulo hemos introducido la metodología de diseño descendente para refinamientos sucesivos, que nos permite diseñar algoritmos que resuelven problemas más complejos y cercanos a la realidad.

Como hemos observado, la base de aplicación de esta metodología consiste en descomponer el problema complejo, mediante la abstracción, en subproblemas independientes y más sencillos de resolver.

Ahora, por lo tanto, podemos construir adecuadamente los algoritmos que resuelvan problemas complejos. Ya disponemos de todos los recursos necesarios: por un lado, tenemos las herramientas –el lenguaje algorítmico–, y por otro, la metodología que nos enseña cómo utilizarlas.

En cuanto consigamos la suficiente destreza en la aplicación de la metodología y el uso de las herramientas del lenguaje algorítmico, habremos alcanzado uno de los objetivos básicos que proponemos al principio de la asignatura: aprender a diseñar e implementar algoritmos correctos, inteligibles, eficientes y generales que resuelvan los problemas planteados. O, dicho de otro modo, aprender a programar de forma profesional; es decir, con eficiencia (en un tiempo razonable) y eficacia (que el resultado responda a los objetivos pedidos). A medida que vayamos resolviendo problemas, iremos adquiriendo esta destreza.

Notad que...

... dado el carácter incremental de la materia, estas pautas son al mismo tiempo una síntesis de todo lo que hemos aprendido mediante los módulos y, por lo tanto, un buen resumen final.

Cada vez que nos enfrentemos al diseño de un algoritmo que debe resolver un problema, será necesario seguir las pautas que os hemos recomendado y que recordamos ahora: entender el enunciado, plantear la solución, formularla y, finalmente, evaluarla.

1) Entender el enunciado

Cuando leamos por primera vez el enunciado, sabremos si corresponde a un problema al que es necesario aplicar la metodología de diseño descendente, cuando los objetos del problema o los objetivos que el enunciado propone alcanzar no se pueden expresar fácilmente en términos de tipos, acciones y funciones elementales de la notación algorítmica, posiblemente a causa de una cantidad excesiva de detalles que hay que tener en cuenta. Si éste es el caso, deberemos encontrar las abstracciones que hacen más abordable un enunciado tan lleno de detalles, y a partir de aquí lo releeremos teniendo presente las abstracciones elegidas, sin considerar los detalles que están por debajo de estas abstracciones y de los que ya nos ocuparemos en posteriores niveles de descomposición.

Como ya sabemos, lo primero que hay que hacer siempre es saber qué queremos conseguir: si no tenemos claro cuál es el problema que se debe resolver, no encontraremos su solución. Debemos interpretar correctamente el enunciado y definir de forma clara y sin ambigüedades qué es necesario hacer. Por ello especificaremos el algoritmo que hay que diseñar. Si es necesario usar abstracciones, la especificación se expresará en términos de las abstracciones convenidas. En otros niveles de descomposición más concretos ya se especificarán los detalles que ahora ocultan las abstracciones.

Recordad que...

… cuando especificamos, sólo pensamos en *qué* es necesario hacer, y no en *cómo* hay que hacerlo.

La especificación nos indica cuál es el problema que se debe resolver; por lo tanto, debemos especificar la precondición y la postcondición de cada algoritmo, acción y función que queramos diseñar.

A pesar de que la especificación se redacta en lenguaje natural, es necesario que procuremos hacerlo con precisión.

La **precondición** describe el estado inicial en el que se encuentra el entorno del que partimos.

La **postcondición** describe el estado final del entorno al que debemos llegar después de que el algoritmo resuelva el problema.

2) Plantear el problema

La etapa de planteamiento empieza una vez establecidas la precondición y la postcondición de un algoritmo, una acción o función. La especificación corresponderá al problema en sí o a una parte del problema (subproblema). Ahora hay que decidir cómo se debe resolver el problema, y por lo tanto, cómo podemos seguir y aplicar gran parte de la metodología que conocemos.

En primer lugar, debemos ver si podemos resolver el problema directamente con facilidad. Es decir, si para resolverlo podemos aplicar directamente las estructuras y los esquemas que conocemos sobre los datos reales del problema. En este caso nos encontramos ante un problema concreto y relativamente sencillo de resolver. A pesar de ello, para realizar con unas ciertas garantías de corrección un problema que se corresponde a un seguimiento de una secuencia, deberemos utilizar las pautas que os hemos ido proponiendo y que sintetizamos ahora:

- Reconocer la secuencia que se debe tratar. En algunos casos será sencillo, porque trabajaremos directamente con la secuencia explícita que sigue los datos del enunciado. En otros casos, tendremos que reconocer que para resolver el problema nos hace falta generar una secuencia. La identificación del primer elemento, del siguiente y del que determina el final de la secuencia concretará la secuencia que se debe tratar.

Recordad que...

… la secuencia puede venir dada por el enunciado del problema, o bien, puede ocurrir que los propios objetivos del problema nos pidan generarla.

- Decidir el esquema que se debe aplicar. Es necesario averiguar cuál de los esquemas que conocemos (búsqueda y recorrido) nos ayudará a resolver el problema.

- Aplicar el esquema elegido al problema concreto, llevando a cabo las adaptaciones convenientes en cada caso.

Es preciso ir con pies de plomo a la hora de aplicar el esquema elegido. Para estar seguros, es conveniente reflexionar sobre las siguientes cuestiones:

- Cómo obtenemos los diferentes elementos de la secuencia.
  - Cómo obtener el primer elemento.
  - Cómo obtener los siguientes.
  - Cómo detectar el final de la secuencia.

- Cómo hacemos el tratamiento.
  - Identificar el tratamiento que corresponde para cada elemento.
  - Definir las variables necesarias.

    Con frecuencia debemos definir variables para acumular los resultados que vamos obteniendo a medida que vamos tratando los elementos; estas variables se deben inicializar convenientemente en el tratamiento inicial.

  - Decidir si es necesario trabajar con tablas.

    Si podemos hacer el tratamiento secuencialmente, no definiremos tablas para almacenar (de forma parcial o total) los datos de la secuencia, ya que esto nos limitaría los datos que debemos tratar, y ocuparíamos espacio innecesario. Sin embargo, en caso de que tengamos que hacer algún acceso directo sobre (algunos o todos) los datos, deberemos definirlos convenientemente.

  - Determinar si hay que tratar el último elemento.

    A veces, el último elemento es un elemento no válido que sólo sirve de marca para detectar el final de la secuencia. No obstante, en otros casos, además de ser la marca final, también es un elemento válido más de la secuencia y, por lo tanto, hay que prestar atención y aplicarle el mismo tratamiento que al resto. Entonces el esquema debe contener este tratamiento haciendo los cambios adecuados.

  - Decidir si es necesario un tratamiento final.

    Observar si todavía se debe llevar a cabo un tratamiento final para llegar a la postcondición, una vez ya tratados todos los elementos de la secuencia.

En el caso de que el problema no siga ninguna secuencia, no podemos aplicar los esquemas de recorrido y búsqueda que conocemos; además, dado que no tenemos ningún esquema previo que aplicar, plantear la resolución del problema consistirá en descubrir cuáles son las acciones y funciones necesarias

que se deben diseñar con el fin de que, combinadas de forma conveniente, alcancen el objetivo final. En estos casos, cada acción o función (elemental o no) resuelve un subobjetivo, y compuestas adecuadamente resolverán el objetivo final del algoritmo.

Las pautas expuestas hasta ahora son también aplicables a problemas complejos que no pueden ser expresados directamente con los elementos de la notación algorítmica. Ahora ampliaremos las indicaciones que nos servirán para plantear estos tipos de problemas más complejos.

Recordad que...

... los esquemas algorítmicos no imponen ninguna restricción respecto del tipo de los elementos de la secuencia y, por lo tanto, se pueden aplicar a secuencias de tipos no elementales definidos mediante tuplas y tablas, para resolver problemas más complejos.

Cuando nos hallamos ante un problema complejo, debemos encontrar abstracciones convenientes que simplifiquen los detalles que hay que tener en cuenta en el planteamiento del problema, y que por ello nos permitan resolverlo con comodidad y con suficiente claridad (por ejemplo, que facilite la aplicación de los esquemas). Una vez que hemos elegido estas abstracciones, debemos mantenerlas a lo largo del planteamiento del problema; es decir, no debemos entrar en detalles que corresponderán a otro nivel de diseño descendente.

Estamos haciendo una abstracción de datos.

Al aplicar la metodología de diseño descendente, identificaremos la secuencia de elementos abstractos; cuando apliquemos el esquema elegido, deberemos definir acciones y/o funciones correspondientes a la identificación del primer elemento, el siguiente y la marca de elemento. De este modo, en lugar de concretar las operaciones anteriores con sentencias y tipos elementales del lenguaje, las encapsulamos mediante acciones y/o funciones con tipos no elementales que esconden los detalles concretos de la abstracción o las abstracciones realizadas. Suponer la existencia de un tipo no elemental con un conjunto de acciones y/o funciones que no existen no nos debe comportar ningún problema: siempre los podremos diseñar más adelante.

En el momento de formular un esquema sobre los datos abstractos, deberemos introducir tantas acciones y funciones y tipos nuevos como sea necesario. De este modo, la aplicación de la metodología del diseño descendente comporta la creación de nuevos enunciados de subproblemas (tipos, acciones y funciones), que se deberán plantear en su nivel correspondiente (y, por lo tanto, por cada subproblema deberemos entender el enunciado, plantearlo, formularlo y evaluarlo; si el subproblema no es suficientemente concreto, dará lugar a otros subproblemas que habrá que resolver). Iremos siguiendo este proceso de forma sucesiva, hasta que sea posible aplicar los esquemas directamente sobre los datos reales del problema.

Estamos haciendo una abstracción de código.

Por lo tanto, en cada nivel más concreto del diseño descendente tendremos que hacer primero un planteamiento del nivel, y a continuación el planteamiento, la formulación y la evaluación de cada uno de los subproblemas para resolver en el nivel de trabajo.

- Para cada abstracción correspondiente al nivel de descomposición es necesario representar los tipos que hay que trabajar en detalle y que todavía no se han definido.

- Para cada acción y/o función ya especificada (subproblema) que se debe desarrollar en este nivel, hay que:
  - Reconocer, si hace falta, la secuencia que hay que tratar. En estos casos es más complicado, ya que los datos abstractos adecuados que forman la secuencia deben ser identificados.
  - Decidir el esquema que se debe aplicar sobre la secuencia abstracta. Como siempre, es necesario observar cuál de los esquemas que conocemos (búsqueda y recorrido) nos ayudará a resolver el problema.
  - Aplicar el esquema elegido al problema concreto, efectuando las variaciones que convengan en cada caso. La aplicación se debe hacer respetando las abstracciones que correspondan a otro nivel más concreto, en el caso de que existan.
  - Especificar las acciones y funciones auxiliares que hayan aparecido en el paso anterior.

El punto clave en la abstracción de código consiste en determinar cuáles son los niveles de abstracción adecuados. Normalmente, los objetos que trata el problema son el criterio que seguimos en el momento de determinar los niveles y descomponer el problema.

Notad que...

… normalmente hacemos una descomposición orientada a objetos.

En cada nivel de descomposición expresamos los algoritmos en función de unos tipos, acciones y funciones que desarrollaremos en el siguiente nivel. La abstracción que hemos hecho en cada nivel nos permite concentrarnos sólo en la resolución de un subproblema concreto; no tenemos por qué preocuparnos de cómo se diseñan los tipos, acciones y funciones que necesitamos; simplemente los utilizamos sabiendo qué hacen y, más adelante, en otros niveles, ya los iremos diseñando.

La descomposición en niveles...

... se puede entender como si tuviésemos un procesador distinto para cada uno que fuese capaz de entender todos los tipos, las acciones y las funciones que aún esperan a ser desarrollados.

Recordad que para cada nivel de abstracción de diseño iría bien releer el enunciado, con el fin de captar los detalles que son relevantes. 

Notad que, a pesar de toda la exposición que acabamos de hacer sobre la tipología de problemas que podemos resolver en este curso, todavía nos faltaría añadir un caso. Nos podemos encontrar ante un problema en el que no reconozcamos o identifiquemos ninguna secuencia que nos convenga al plantearlo. En este caso, deberemos descubrir el orden de las acciones que pueden alcanzar el objetivo que que pretende el algoritmo.

En ocasiones el número de acciones elementales que se deben proponer para conseguir un objetivo puede ser excesivo; en este caso conviene reconocer las

diferentes etapas que alcanzan subobjetivos parciales que, al componerse, proporcionan el objetivo final. Cada una de estas etapas se encapsula mediante acciones y funciones. De este modo obtenemos legibilidad e inteligibilidad en las soluciones formuladas. La orientación de la descomposición es funcional. Notad que esto que acabamos de comentar es aplicable a subproblemas provenientes de una descomposición orientada a objetos (por ejemplo, el tratamiento de un elemento abstracto de una secuencia).

3) Formular la solución

Una vez que ya sabemos cómo queremos resolver el problema, sólo nos hará falta formular la solución en la notación algorítmica que hemos definido. De este modo, podremos expresar con precisión y rigor el algoritmo que resuelve el problema.

Recordad que...

... cuando formuléis vuestros algoritmos, debéis seguir estrictamente la notación algorítmica que hemos definido para evitar ambigüedades.

Ya conocemos la sintaxis y la semántica de cada uno de los elementos y, en el siguiente cuadro, se sintetizan las construcciones del lenguaje algorítmico.

Resum d'elements i construccions del llenguatge algorísmic

- **Objectes**
  - Nom
  - Valor
    - (Constants) **const** ... **fconst**
    - (Variables) **var** ... **fvar**
  - Tipus
    - Predefinits
      - **enter**
      - **real**
      - **boolea**
      - **caracter**
    - Definició de nous: **tipus** ... **ftipus**
      enumeratius: *nom* = {$valor_1$, ... , $valor_n$}
    - Constructors
      - **taula**
      - **tupla** ... **ftupla**
- **Estructures algorísmiques**
  - Assignació: *nomvariable* := *expressió*
  - Alternativa
    - **si** *condició* **llavors** ... **fsi**
    - **si** *condició* **llavors** ... **sino** ... **fsi**
  - Iterativa
    - **mentre** *condició* **fer** ... **fmentre**
    - **per** ... **fins** ... **fer** ... **fper**
- **Accions i funcions**
  - Predefinides
    - de conversió de tipus: realAEnter(...), enterAReal(...), caracterACodi(...), codiACaracter(...)
    - d'entrada/sortida: llegirEnter(), llegirCaracter(), llegirReal(), escriureEnter(...), escriureReal(...), escriureCaracter(...)
  - Definició de noves
    - **accio** nom(*paràmetres*) ... **faccio**
    - **funcio** nom(*paràmetres*): **tipus** ...
    - **retorna** ... **ffuncio**

4) Evaluar la solución

De momento no hemos introducido ningún método formal para verificar la corrección de un algoritmo. Es decir, debemos hacerlo informalmente, mediante el razonamiento sobre la semántica de los elementos y las construcciones del lenguaje algorítmico que hayamos utilizado, y mediante la comprobación, paso a paso, de que los hemos ido componiendo de modo adecuado para conseguir finalmente la postcondición deseada.

Los comentarios que hayamos insertado en el momento de hacer el diseño, como ya sabéis, nos ayudarán a llevar a cabo esta reflexión.

De todas formas, el uso de la metodología de diseño y aplicación adecuada de los esquemas de recorrido y búsqueda nos asegura la corrección de los tratamientos secuenciales que hayamos hecho. Recordad que un esquema es una especie de plantilla cuyo funcionamiento conocemos muy bien, y que utilizamos para solucionar un tipo concreto de problemas; por lo tanto, si lo aplicamos correctamente, nos conducirá a resultados también correctos.

El uso combinado del diseño descendente y de los mecanismos de abstracción no sólo beneficia al diseño, sino que también favorece otras etapas del desarrollo de programas:

Como vimos en el módulo "Introducción a la programación" las etapas de desarrollo de un programa son el análisis de requerimientos, el diseño, la implementación, las pruebas y el mantenimiento.

- El proceso de desarrollo: gracias al hecho de parcelar la complejidad de los problemas y de descomponerlos sucesivamente en fragmentos menores, nos ocupamos en cada momento de la resolución de un único problema de dimensiones más manejables.

- La puesta a punto: los fragmentos que componen los programas se pueden probar por separado (individualmente cada acción y función), y poco a poco.

- El mantenimiento: las modificaciones posteriores que será necesario realizar durante la vida útil de un programa, normalmente, tienen un impacto localizado sólo en unas pocas acciones; el resto del programa no se ve afectado y, por lo tanto, el hecho de tenerlo descompuesto facilita su modificación. Por otro lado, si los programas tienen una estructura más clara, resulta más fácil y seguro modificarlos.

- La reutilización del código: si se desarrollan programas que tratan con objetos similares, seguramente podremos aprovechar estos diseños de tipos ya hechos por otros programas, así como las funciones y las acciones que traten.

- La legibilidad: los programas que nos resultan después de aplicar las técnicas de diseño descendente y los esquemas están estructurados, por lo que son más fáciles de leer y analizar.

Ejercicios de autoevaluación

1. Diseñar un algoritmo que descubra cuántas veces aparece "LA" en una frase escrita en mayúsculas y terminada con un punto en el canal de entrada. Por ejemplo, en la entrada tenemos la frase siguiente:

BLANCA REGALA A LAIA UNA MANTA.

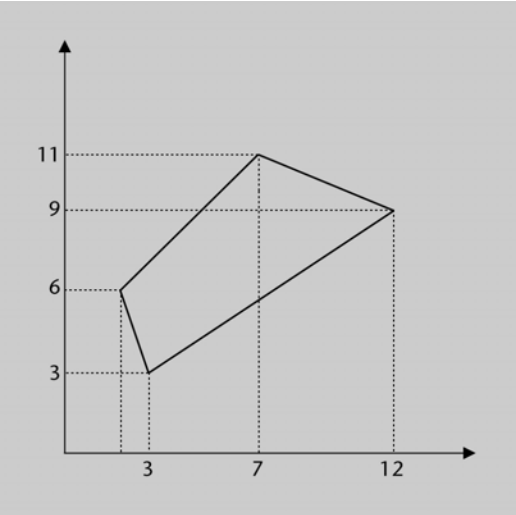
El programa deberá dar como resultado: 3, ya que encontramos 'LA' en los siguientes casos:

BLA[1]NCA REGALA[2] A LA[3]IA UNA MANTA.

2. Dado un polígono especificado por una secuencia de vértices en el orden de la serie de su contorno, diseñad un algoritmo que encuentre su área. La secuencia termina con el primer vértice con el que ha empezado. Como mínimo, en la entrada están los datos de un triángulo. Cada vértice se especifica por dos coordenadas reales. Por ejemplo, si la secuencia es:

3.0 3.0 2.0 6.0 7.0 11.0 12.0 9.0 3.0 3.0

El polígono que representa es:



Un amigo a quien le gusta la geometría nos dice que el cálculo del área de un polígono, descrito por sus vértices y el contorno que delimitan, se puede calcular mediante la fórmula:

$$\sum_{i=1}^{n} \frac{(x_{i+1} - x_i)\ (y_{i+1} + y_i)}{2}$$

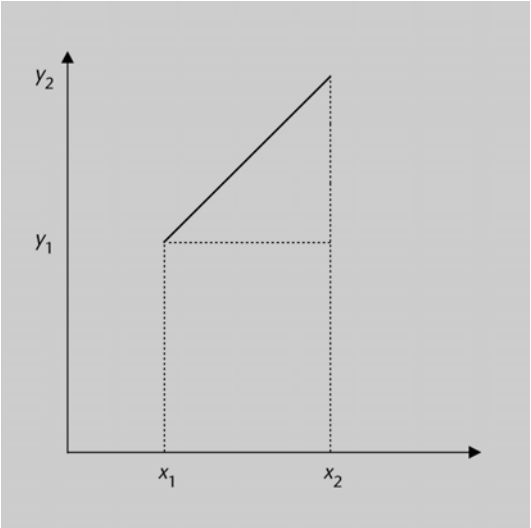
donde n es el número de aristas y cuando $i = n$, $i + 1$ es en realidad 1 (utilizar el módulo es la manera correcta de hacerlo, pero no lo ponemos para que la fórmula tenga una mayor claridad). En nuestro ejemplo, el área A del polígono sería:

A = 1/2 ((2 – 3) (6 + 3) + (7 – 2) (11 + 6) + (12 – 7)(9 + 11) + (3 – 12)(3 + 9))

A = 1/2 (–9 + 85 + 100 – 108) = 68/2 = 34

Podría darse el caso de que el resultado fuese negativo, pero si le cambiásemos el signo el valor del área continuaría siendo correcto. Disponemos de la función *abs*(*r*), que da el valor absoluto

de un número real *r*. De hecho, cada término del sumatorio calcula el área del trapecio que forma una arista del polígono sobre el eje *x*, tal como se aprecia en la siguiente figura:



Para quienes tengáis curiosidad por el tema (no es necesario que leáis esto para diseñar el algoritmo, ya que el enunciado dice muy claramente qué se debe resolver), podéis ver en la figura que el trapecio está compuesto por un triángulo y un rectángulo; entonces, el área A_t será:

$$\mathrm{A}_t = \underbrace{y_i\,(x_2 - x_1)}_{\text{rectángulo}} + \underbrace{\frac{(x_2 - x_1)\,(y_2 - y_1)}{2}}_{\text{triángulo}}$$

$$\mathrm{A}_t = \frac{2y_i\,(x_2 - x_1) + (x_2 - x_1)\,(y_2 - y_1)}{2}$$

$$\mathrm{A}_t = \frac{(x_2 - x_1)\,(y_2 + y_1)}{2}$$

Observemos que el área tiene signo positivo o negativo dependiendo de si $x_2 > x_1$ o no. Si los vértices de las aristas de un polígono se dan respetando el sentido del contorno (horario o antihorario), las áreas de los trapecios se podrán sumar de forma algebraica, de modo que obtendremos como resultado el área neta del polígono, una vez que hayamos recorrido todas las aristas.

De este modo, el área del polígono se representará de la forma siguiente:

$$\sum_{i=1}^{n} \frac{(x_{i+1} - x_i)\,(y_{i+1} + y_i)}{2}$$

Donde *n* es el número de aristas, y cuando *i* es igual a $n + 1$, $n + 1$ es 1.

3. Dada una frase acabada con un punto, contad el número de palabras capicúa. Entre palabras hay uno o más caracteres separadores como espacios, comas o puntos y coma. Cada palabra de la frase tiene como mucho 15 caracteres.

4. Una empresa de publicidad tiene un servidor de Internet y quiere facturar a sus clientes las páginas de HTML instaladas. Con este propósito, la empresa quiere que diseñemos un programa que indique lo que cada cliente tiene que pagar a partir del contenido de las páginas instaladas.

En la entrada del programa habrá una secuencia de caracteres organizada de la forma siguiente:

$$m_{11} \cdot m_{12} \cdot ... \cdot m_{1n_1} \cdot \mathit{FinTextoHTML} \cdot m_{21} \cdot m_{22} \cdot m_{2n_2} \cdot \mathit{FinTextoHTML} \cdot ... \cdot \\ \cdot ... \cdot m_{n1} \cdot m_{n2} \cdot m_{nn_n} \cdot \mathit{FinTextoHTML} \cdot \mathit{FinPaginas}$$

Donde m_{ij} son cadenas de caracteres sin separadores que pueden tener cualquier número de caracteres. *FinTextoHTML* es la palabra "FinTextoHTML" e indica el final de la página HTML. *FinPaginas* indica el final de la secuencia y es la palabra "FinPaginas" acabada con un blanco. Se considera que un carácter es separador si éste es un espacio (" "), un "=", o se trata de un control de línea (retorno de carro o salto de línea).

De entre las palabras que hay en cada página, interesa reconocer la palabra "propiedad", ya que a continuación de ésta aparecerá el nombre del cliente que ha puesto la página y, por lo tanto, a quien se le debe facturar. Para facturar la página a la empresa propietaria, el programa deberá tener en cuenta lo siguiente:

- Las palabras "propiedad", "FinTextoHTML" y "FinPaginas" son obviamente gratuitas, ya que sólo sirven para controlar la secuencia.
- Toda palabra en la cual el primer carácter sea "<" y el último ">" no contará, ya que se trata de controles del lenguaje HTML que la empresa considera gratuitos (sólo palabras, las frases entre los caracteres "<" y ">" no serán gratuitas).
- Si se encuentra la palabra "<IMG", se facturarán 50 euros.

Cualquier otra palabra no incluida en los puntos anteriores y que tenga más de un carácter se facturará a 5 euros.

Observaciones:

- No habrá más de 50 empresas.
- Una empresa puede tener un número cualquiera de páginas.
- La secuencia de entrada estará bien formada. En particular:
  – Puede haber páginas vacías (S ≡ 'FinTextoHTML FinTextoHTML ... ').
  – La secuencia puede ser vacía (S ≡ 'FinPaginas').
  – Puede no existir propietario. Dado este caso, la página se facturará a nombre de una hipotética empresa denominada *NoIdentificados*.
- Nos interesa tratar sólo los primeros 40 caracteres de una palabra, como mucho.
- Las palabras de control delimitadas por "<" y ">" no tendrán más de 40 caracteres.

Un ejemplo de secuencia podría ser el siguiente:

```
<! propiedad VagonesPopulares>
<HEAD>
<TITLE> La última innovación </TITLE>
</HEAD>
<BODY>
<meta name="description" value=" Especificación">
<meta name="keywords" value=" diseño">
<meta name="resource-type" value="documento">
<meta name="distribution" value="global">
<P>
<BR> <HR><A NAME=tex2htm1176 HREF="node16.html">
<IMG ALIGN=BOTTOM ALT="next"
SRC= "http://asdww.cern.ch/icons/next_motif.gif">
</A> <A NAME=tex2htm1174 HREF= "no
...
FinTextoHTML
<HEAD>
<TITLE> Cursos de posgrado</TITLE>
</HEAD>
<BODY>
<! propiedad UOC>
...
<BGSOUND LaMarsellesa>
<FRAME Pals>
<SCRIPT javascript>
puedes encontrar la página en
<A NAME=hola HREF = http://www.uoc.es/presentacio>
FinTextoHTML

FinPaginas
```

En la salida tendremos:

```
VagonesPopulares 7.2 UOC 2.1
```

Hay 42 palabras contabilizadas a 5 euros. Para el caso de VagonesPopulares:

```
<!  VagonesPopulares> La última innovación</TITLE> <meta name “descrip-
tion” value “ Especificación”> <meta name “keywords” value “diseño”> <me-
ta name “resource-type” value “documento”> <meta name “distribution”
value “global”> <HR><A NAME tex2htm1176 HREF “node16.html”> <ALIGN BOT-
TOM ALT “next” SRC “http://asdwww.cern.ch/icons/next_motif. <A NAME
tex2htm1174 HREF “no ... .
```

Y aún hay que añadir 50 euros al importe porque hay un “<IMG”. Y las de la UOC son 21:

```
Cursos de posgrado</TITLE> <! UOC... <BGSOUND LaMarsellesa> <FRAME Pals>
<SCRIPT javascript> puedes encontrar la página en <A NAME hola HREF
http://www.uoc.es/presentacio>.
```

Solucionario

1. Tenemos que diseñar un algoritmo que calcule cuantas veces aparece "LA" en una frase escrita en mayúsculas y acabada en punto y final.

Especificación

La frase se puede ver como una secuencia de caracteres acabada con un punto; necesitaremos un contador que denominaremos *nuDeLA* para indicar el número de ''LA" que aparecen en la secuencia. De este modo, entonces,

```
var
   nuDeLA: entero
fvar
   { Pre: f = F, f es una frase acabada con un punto y f sólo contiene mayúsculas }
   contarLA
   { Post: nuDeLA indica el número de 'LA' que hay en F }
```

Planteamiento general 1

Al principio puede parecer que sea posible pensar el problema directamente a partir de la secuencia de caracteres aplicando un esquema de recorrido. Pero, tener que trabajar carácter por carácter dificulta nuestra perspectiva del problema, ya que necesitaremos ver dos caracteres al mismo tiempo para decidir si es una pareja "LA" o no. Si se piensa carácter a carácter, nos podemos encontrar con los siguientes casos:

1. El carácter leído es diferente de 'L' → Pasamos a consultar el siguiente carácter.

2. El carácter leído es igual a 'L' → Debemos verificar que el carácter siguiente sea una 'A'. Para el carácter siguiente, nos encontraremos con los siguientes casos:

(a) Es una 'A'→ Incrementamos el contador de 'LA'.

(b) Es una 'L'→ Debemos continuar verificando el carácter que sigue al actual, ya que se podría dar el caso de, por ejemplo, 'ELLA'.

(c) Es diferente de 'L' y 'A' → No tenemos en cuenta el carácter actual, ya que no dará *'LA'*.

Solución 1

```
algoritmo contarLA
var
   c: caracter;
   nuDeLA: entero;
fvar
   { f = F , f es una frase acabada en punto y f sólo contiene mayúsculas }
   nuDeLA := 0;
   c := leerCaracter();
mientras c ≠ '.' hacer
      si c = 'L' entonces {Carácter actual es una 'L'}
      c := leerCaracter();
      si c = 'A' entonces { el carácter anterior fue una 'L' y el actual, una 'A' }
         nuDeLA := nuDeLA + 1;
         c := leerCaracter()
      fsi
      {En caso de que
            c = 'A'→ ya habremos avanzado y contado.
            c = '.' → mejor no avanzar.
            c = 'L'→ aprovecharemos el primer condicional y, por lo tanto, no avanzamos.
            c ≠ 'L'→ ya avanzaremos con el sino del primer condicional.}
   sino {c ≠ '.'}
   c:= leerCaracter()
```

```
fsi
fmientras
   {nuDeLA indica el número de 'LA' que hay en F}
   escribirEntero(nuDeLA)
falgoritmo
```

Evaluación 1

La corrección de la solución presentada es muy difícil de evaluar. Por ejemplo, debemos resolver cuestiones del siguiente tipo:
1. ¿Tiene en cuenta la secuencia vacía formada sólo por el punto?
2. ¿Tiene en cuenta casos como 'ELLA'?
3. ¿Avanza siempre la secuencia, o se queda estancada en algún punto?

Si seguís bien el algoritmo, os daréis cuenta de que todas las cuestiones están bien cubiertas. A pesar de ello, es muy difícil verlo.

Planteamiento general 2

El hecho de pensar este problema en términos de caracteres ha creado un cierto desorden. En cambio, si convenimos en la abstracción de pareja de caracteres, veremos que el problema es más fácil de analizar y desarrollar. Lo primero que debemos hacer es imaginarnos que tenemos una secuencia de parejas de caracteres. Esto nos conduce a particularizar el esquema de recorrido teniendo en cuenta la abstracción convenida:

1) Primer nivel

Solución

```
algoritmo contarLA
   var
      nuDeLA: entero;
      Pareja: tPareja;
   fvar
   { f = F, f es una frase  acabada en punto y f sólo contiene mayúsculas}
   nuDeLA := 0;
   obtenerPrimeraPareja(pareja);
   mientras no ultimaPareja(pareja) hacer
      si esLA(pareja) entonces
         nuDeLA := nuDeLA + 1
      fsi
      obtenerPareja(pareja)
   fmientras
   { nuDeLA indica el número de 'LA'que hay en F }
   escribirEntero(nuDeLA)
falgoritmo
```

Ahora tenemos una parte de la solución expresada, y el nivel se completará con la especificación de las acciones y funciones que quedan por desarrollar:

```
accion obtenerPrimeraPareja (sal p: tPareja);
    { Pre: s = S, s es una frase acabada en punto y la parte izquierda de s está vacía }
    { Post: p contendrá la primera pareja de la secuencia de parejas y la parte izquierda de s
      contendrá un carácter }

accion obtenerPareja (entsal p: tPareja);
    { Pre: s = S, s es una frase acabada en punto y la parte izquierda de S no está vacía, la parte
      derecha de s tampoco está vacía y p = P y P es la anterior pareja obtenida }
    { Post: p contendrá la pareja siguiente de la secuencia de parejas, y la parte izquierda de s
      contendrá un carácter más }

funcion ultimaPareja(p: tPareja): booleano
   { Pre: p = P }
   { Post: ultimaPareja(p) es cierto si P es la última pareja de la secuencia }
```

funcion EsLA(P: tPareja): **booleano**;
{ Pre: p = P }
{ Post: *esLA(P)* es **cierto** si *P* es la pareja *LA* }

Notad que para verificar esta parte no necesitamos conocer la definición del tipo *tPareja*, y dado que disponemos de la especificación de acciones y funciones que hay que desarrollar, podemos comprobar si es correcta la solución formulada hasta ahora.

2) Segundo nivel

Planteamiento

Para continuar el diseño se necesita definir la representación que tendrá una pareja de caracteres. Una posibilidad consiste en definir la tupla, y otra en definir una tabla. Elegiremos una tabla (elegir una tupla también habría sido una buena solución para este problema).

```
tipo
   tPareja = tabla [2] de caracter;
ftipo
```

Ahora debemos plantear el problema de cómo hacer que nuestra secuencia de caracteres se pueda ver como la secuencia de parejas que nos conviene. Veamos la correspondencia que existe: la secuencia f está formada por $c_1c_2c_3c_4c_5...c_f$, donde c_f es el carácter final '.', y queremos ver una secuencia de parejas $p_1p_2...p_f$ tal que $p_1 = c_1c_2$, $p_2 = c_2c_3$, y p_f tenga como segundo carácter el punto.

Observamos que se trata de parejas solapadas de caracteres: cada pareja que no sea la primera comparte un carácter con la anterior.

La primera pareja entendida como los dos primeros caracteres de la secuencia (c_1c_2), obligaría a que la secuencia de caracteres tuviese como mínimo un carácter diferente al punto (c_1 *'.'*). Para ajustarnos a la precondición (como mínimo, la secuencia tiene el punto final) deberemos hacer que el primer carácter de la primera pareja sea un carácter arbitrario que no forme parte de la secuencia y que el segundo carácter sea el primero de la secuencia. Elegiremos el carácter arbitrario de modo que no nos estropee el tratamiento (que no sea una 'L'). Por ejemplo, el carácter espacio. De este modo, la secuencia que nos conviene es:

$$p_1 = \text{' '}\ c_1,\ p_2 = c_1c_2, \text{ y } p_f$$

que tenga como segundo carácter el punto.

Por lo tanto:

Solución

```
accion obtenerPrimeraPareja(sal p: tPareja)
   p[1]:= ' ' ;
   p[2]:= leerCaracter()
faccion

accion obtenerPareja(entsal p: tPareja)
   p[1]:= p[2];
   p[2]:= leerCaracter()
faccion

funcion ultimaPareja(p: tPareja): booleano
    retorna p[2] = '.'
ffuncion

funcion esLA(p: tPareja): booleano
    retorna p[1] = 'L' y p[2] = 'A'
ffuncion
```

2. Tenemos que diseñar un algoritmo que calcule al área de un polígono a partir de las coordenadas de sus vértices.

Especificación

Pasemos a especificarlo. Para hacerlo, abreviaremos las especificaciones denominando la secuencia de entrada de datos *pol*.

var area: **real fvar***;*
{ Pre: *pol* = *POL* y *POL* es una secuencia de reales que representa como mínimo un triángulo }

areaPoligono
{ Post: en *area* se encuentra el área del polígono representado en la secuencia *POL* }

1) Primer nivel

Planteamiento

Para afrontar bien el problema es necesario identificar los objetos que pueden ser convenientes para nuestro análisis. Éstos son:

- aristas
- vértices
- reales

En la primera etapa del diseño, es conveniente hacer la formulación del algoritmo en términos de arista, a pesar de que en la entrada disponemos de una secuencia de reales. Imaginemos, entonces, el tipo arista y que en la entrada tenemos una secuencia de arista. En este caso, las cuestiones clásicas de cómo se obtiene el primero, el siguiente y el último elemento estarán encapsuladas mediante las acciones y las funciones que encontraréis más adelante.

Repasando el enunciado, vemos que cada término del sumatorio es el área del trapecio correspondiente a cada arista. El tratamiento que hay que aplicar consistiría, de este modo, en ir acumulando estas áreas para tener al final el sumatorio. Obviamente, esto corresponde al esquema de recorrido sobre una secuencia de aristas y también deberemos reflexionar sobre el esquema. Por ejemplo, es necesario tratar la última arista que hace de marca de fin de secuencia; por lo tanto, deberemos modificar ligeramente el esquema para conseguir este efecto.

Otra observación es que del sumatorio podemos extraer factor común de la división por dos, y posponerla para el tratamiento final. El tratamiento final consistirá en dividir por dos las áreas acumuladas y encontrar su valor absoluto. Así, para cada arista calcularemos el área doble, y de este modo ahorraremos algunos cálculos. Notad, en la solución que tenéis más adelante, que la última asignación es el tratamiento final, pero la expresión en la parte derecha de la asignación incluye el tratamiento de la última arista.

Tenemos, pues, todos los elementos para pasar a expresar una solución centrada en las aristas del polígono y las áreas de los trapecios correspondientes; así, no entraremos en detalles más concretos.

algoritmo areaPoligono
var
area: **real**;
a: **arista**;
fvar
{ Pre: *pol* = *POL* y *POL* es una secuencia de reales que representa como mínimo un triangulo }
obtenerPrimeraArista(a);
area := 0;
mientras no ultimaArista(a) **hacer**
area := area + areaTrapecio(a);
obtener Arista(a)
fmientras
area := abs ((area + areaTrapecio(a)) / 2.0)
{ Post : en *area* tenemos el area del polígolno representado en la secuencia POL }
falgoritmo

accion obtenerPrimeraArista(**sal** a: arista)
{ Pre: en la entrada tenemos una secuencia de aristas *pol* y *pol* = *POL* y *pol* tiene como mínimo tres aristas en la parte derecha, mientras que la parte izquierda está vacía }
{ Post: *a* será la primera arista de *POL* y en la parte izquierda de *pol* hay una arista }

accion obtenerArista(**entsal** a: arista)
{ Pre: *pol* = *POL* y la parte derecha de *pol*, *PDPol,* no está vacía y tiene como mínimo una arista; la parte izquierda de *pol* tiene como mínimo una arista, *a* = *A* y *A* es la última arista obtenida }
{ Post: *a* tiene la primera arista de *PDPol*, y el valor de *a* es la última arista de la parte izquierda de *pol* }

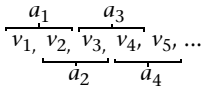
funcion ultimaArista(a: arista): **booleano**
{ Pre: *a* = *A* }
{ Post: *ultimaArista* (*A*) es **cierto** si *A* es la última arista de *POL* }

funcion areaTrapecio (a: arista): **real***;*
{ Pre: *a* = *A* }
{ Post: *areaTrapecio*(*A*) es el área doble del trapecio formado por la arista *A* y el eje *x* }

2) Segundo nivel

Convencidos del primer nivel, pasemos al siguiente, que consiste en definir las acciones y las funciones y tipos que han quedado pendientes. En este nivel es conveniente pensar en términos de vértices, y considerar nuestra secuencia como una secuencia de vértices.

Para el planteamiento de la secuencia de vértices, es necesario observar que la abstracción de arista está formada por dos parejas de vértices solapadas en la secuencia:



Así pues, las ideas que hay que aplicar al planteamiento (primer elemento, siguiente y último) son las mismas que nos encontramos cuando solucionábamos el problema de contar 'LA', y por ello no las representaremos aquí. La última arista merece una atención especial. Si examinamos el enunciado, vemos que la última arista es aquélla cuyo extremo final, v_f, es idéntico al vértice inicial de la primera arista (v_1), que es además el primer elemento de la secuencia. Con estas observaciones pasamos directamente a la solución.

Cuando definamos el tipo *arista* deberemos tener en cuenta que necesitaremos sus dos vértices para poder aplicar la fórmula del área del trapecio, y también saber cuál es el último vértice de la secuencia:

tipo
arista = **tupla** vi, vf, dv: vertice;
ftupla
ftipo

{ Pre: en la entrada tenemos una secuencia de aristas *pol* y *pol* = *POL* y *pol,* como mínimo tiene tres aristas en la parte derecha y la parte izquierda está vacía }
accion obtenerPrimeraArista(**sal** A: arista)
obtenerPrimerVertice(A.vi); { por la precondición sabemos que como mínimo hay cuatro vértices }
obtenerVertice(A.vf); { contando el que marca el final }
A.dv := A.vi
faccion
{ Post: A tiene la primera arista de *pdPol* y el valora de *A* es la última arista de la parte izquierda de Pol }

{ Pre: *pol* = *POL* y la parte derecha de *pol*, *pDPol*, no está vacía y tiene como mínimo una arista; la parte izquierda de *pol* tiene como mínimo una arista; *a* = *A* y *A* es la última arista obtenida }
accion obtenerArista(**entsal** a: arista)
a.vi:= a.vf;
obtenerVertice(a.vf)
faccion
{ Post: *A* tiene la primera arista de *pDPol* y el valor de *A* es la última arista de la parte izquierda de *pol* }

{ Pre: *a* = *A* }
funcion ultimaArista(a: arista): **booleano**
    retorna verticesIguales(a.vf, a.dv)
ffuncion
{ Post: ultimaArista (A) es cierto si A es la última arista de POL }

{ Pre: *a* = *A* }
funcion areaTrapecio(a: arista): **real**
    retorna areaTrapecioVertices (a.vi,a.vf)
ffuncion
{ Post: *areaTrapecio*(*A*) es el área doble del trapecio formado por la arista *A* y el eje *x* }

3) Tercer nivel

Finalizado el segundo nivel, es necesario definir la secuencia de vértices a partir de la secuencia de reales. En este caso, podemos observar que los vértices son parejas de reales disyuntas:

$$\overbrace{x_1,\ y_1,}^{v_1}\quad \overbrace{x_2,\ y_2,}^{v_2}\ \ldots$$

tipo
   vertice = **tupla** x, y: real;
             ftupla
ftipo

{ Pre: *v*1 = *V*1 y *v*2 = *V*2 }
funcion areaTrapecioVertices (v1, v2: vertice): **real**
   retorna (v2.x – v1.x) ∗ (v2.y + v1.y)
ffuncion
{ Post: *areaTrapecioVertices*(*V*1,*V*2) es el área doble del trapecio formado por la arista que va de V1 a V2 y el eje *x* }

{ Pre: *pol* = *POL* y *pol* tiene como mínimo dos reales (un vértice) en su parte derecha }
accion obtenerPrimerVertice(**sal** v: vertice)
   v.x:= leerReal();
   v.y:= leerReal()
faccion
{ Post: la parte izquierda de *pol* tiene un vértice más y v tiene el vértice que antes estaba en la parte derecha }

{ Pre: *pol* = *POL* y *pol* como mínimo tiene dos reales (un vértice) en su parte derecha }
accion obtenerVertice(**sal** v: vertice)
   v.x:= leerReal();
   v.y:= leerReal()
faccion
{ Post: la parte izquierda de *pol* tiene un vértice más y *v* tiene el vértice que antes estaba en la derecha }

{ Pre: *v*1 = *V*1 y *v*2 = *V*2 }
funcion verticesIguales(v1, v2: vertice): **booleano**
    retorna (v1.x = v2.x) **y** (v1.y = v2.y)
ffuncion
{ verticesIguales(*v*1, *v*2) es **cierto** si V1 y V2 son el mismo vértice }

3.

Especificación

var nPalabrasC: **entero;**
fvar
{ en la entrada hay la secuencia de palabras *f* de la frase *F*. *F* no está vacía }
cuentaCapicuasFrase
{ *nPalabrasC* tiene el número de palabras capicúa de *F* }

1) Primer nivel: frase

Planteamiento

- Abstracciones utilizadas: palabra.
- Identificación de la secuencia: frase vista como secuencia de palabras. La obtención de las palabras primera y siguiente y el reconocimiento de la marca se encapsulan mediante acciones y funciones.
- Identificación del esquema: recorrido. Hay que tratar la palabra que hace de marca de final.

Aplicación directa del algoritmo

```
algoritmo cuentaCapicuasFrase
   var
      palabra: tPalabra;
      nPalabrasC: entero;
   fvar
   nPalabrasC := 0;
   obtenerPrimeraPalabra(palabra);
   mientras no palabraFinal(palabra) hacer
      si palabraCapicua(palabra) entonces nPalabrasC := nPalabrasC + 1;
      fsi
      obtenerPalabra(palabra)
   fmientras
   si palabraCapicua(palabra) entonces nPalabrasC := nPalabrasC + 1
   fsi
   escribirEntero(nPalabrasC)
falgoritmo
```

Acciones pendientes de desarrollar

El algoritmo funcionará si las acciones y funciones que se deben desarrollar cumplen las especificaciones siguientes:

accion obtenerPrimeraPalabra(**sal** m: tPalabra)
{ Pre: en la entrada hay la secuencia de palabras *f* de la frase *F*. La parte izquierda de *f* está vacía. La parte derecha, *DF*, no está vacía }
{ Post: *m* representa la primera palabra de *DF*. La palabra obtenida estará en la parte izquierda de la secuencia *f* }

accion obtenerPalabra(**entsal** m: tPalabra)
{ Pre: en la entrada hay la secuencia de palabras *f* de la frase *F*. La parte derecha, *DF*, no está vacía. La anterior palabra leída está en *m* }
{ Post: *m* representa la primera palabra de *DF*. La palabra obtenida estará en la parte izquierda de la secuencia *f* }

funcion palabraFinal(**ent** m: tPalabra): **booleano**
{ Pre: *m* = *M* }
{ Post: *palabraFinal*(*M*) es **cierto** si *m* es la última palabra de una frase }

funcion palabraCapicua(**ent** m: tPalabra): **booleano**
{ Pre: *m* = *M* }
{ Post: *palabraCapicua*(*M*) es **cierto** sólo si *m* es una palabra capicúa }

2) Segundo nivel: palabra

- Abstracciones utilizadas: ninguna.
- Identificación de la secuencia: palabra vista como una secuencia de caracteres. La obtención de las palabras primera y siguiente se hará mediante la función predefinida *leerCaracter*() y el reconocimiento del carácter que marca el final será el carácter espacio o la coma, o el punto y coma, o el punto.

Definición del tipo

Necesitaremos guardar los elementos siguientes:

- El contenido de la palabra: la tabla *c* y *long*, ya que la tabla puede estar parcialmente llena.
- El último carácter leído de la secuencia: *caracterLeido* será siempre el carácter de la siguiente palabra que se lee en el caso de que no se haya llegado al final de la secuencia.
- Un indicador de final: *final* será **cierto** si se trata de la última palabra.

```
const maxCar: entero = 15;
fconst

tipo
tPalabra = tupla
                long: entero;
                c: tabla[maxCar] de caracter;
                caracterLeido: caracter;
                final: booleano;
           ftupla
ftipo
```

Planteamiento y soluciones de las acciones y funciones del nivel

Identificación esquema: recorrido.

```
accion obtenerPrimeraPalabra(sal m: tPalabra)
   var c: caracter;
   fvar
   m.long := 0;
   c := leerCaracter();
   mientras separador(c) y c ≠ '.' hacer { saltamos los primeros separadores }
       c := leerCaracter()
   fmientras
   m.caracterLeido := c;
   obtenerPalabra(m)
faccion
```

Identificación del esquema: recorrido.

```
accion obtenerPalabra (entsal m: tPalabra)
   var c: caracter;
   fvar
   m.long := 0;
   c := m.caracterLeido;
   mientras no separador(c) y c ≠ '.' hacer { obtenemos la palabra }
       m.long := m.long + 1;
       m.c[m.long] := c;
       c := leerCaracter()
   fmientras
   mientras separador (c) y c ≠ '.' hacer { saltamos los separadores }
       c := leerCaracter()
   fmientras
   m.caracterLeido := c;
   m.final := c = '.'
faccion
```

Aplicación directa del algoritmo

```
funcion palabraFinal(m: tPalabra): booleano
    retorna m.final
ffuncion
```

Identificación del esquema: búsqueda.

```
funcion palabraCapicua(m: tPalabra): booleano
   var
      i, j; entero;
      encontrado: booleano;
   fvar
   i := 1; j := m.long;
   encontrado := falso;
   mientras (no encontrado) y (i < j) hacer
      si m.c[i] ≠ m.c[j] entonces
         encontrado := cierto
      sino i := i + 1; j := j – 1
      fsi
   fmientras
    retorna no encontrado
ffuncion

funcion separador(ent c: caracter): booleano
    retorna (c= ' ') o (c = ',') o (c = ';')
ffuncion
```

4. Del enunciado deducimos que podemos organizar los niveles de descomposición del diseño a partir de los objetos del problema. En la entrada del programa tenemos una secuencia de páginas. Una página es una secuencia de palabras separadas por espacios y controles de línea. Una palabra es una secuencia de caracteres sin espacios ni controles de línea. La secuencia de entrada es de tipo carácter. Podemos distinguir, pues, tres niveles de descomposición del diseño:
a) Secuencia de páginas de web acabada con "FinPaginas".
b) Página: secuencia de palabras acabada con la palabra "FinTextoHTML".
c) Palabra: secuencia de caracteres terminada con separador (espacio, símbolo =, caracteres de control de línea).

1) Primer nivel: secuencia de páginas

Releemos el enunciado y lo adaptamos siguiendo la abstracción de página.

Especificación

El objetivo es confeccionar una lista de las empresas clientes que participan en el servidor y, para cada empresa participante, obtener la cantidad que se les debe facturar. Para cada página, podemos obtener el nombre de la empresa cliente a la cual pertenece y valorar su precio a partir del contenido. En un primer nivel, formulamos el algoritmo en términos de una secuencia de páginas. De este modo, por lo tanto, la entrada se puede ver como una secuencia de páginas:

$$s = S = <P_1 \cdot P_2 \cdot P_n \cdot \text{'FinPaginas'}>$$

'FinPaginas' indica el final de la secuencia y se puede considerar como una página especial y única que sólo tiene la palabra "FinPaginas".

La precondición se puede expresar como:

{ Pre: $s = S = <P_1 \cdot P_2 \cdot \ldots \ P_n \cdot$ 'FinPaginas'> **y** (toda P_i, $1 \leq i \leq n$, es una página) }

La postcondición consistirá en tener una secuencia t de empresas clientes con el importe facturado. La secuencia t tendrá la forma siguiente:

$$t = <E_1 \cdot F_1 \cdot E_2 \cdot F_2 \cdot \ldots \cdot E_m \cdot F_m>$$

Donde E_i es el nombre de una empresa cliente presente en la secuencia S, y F_i es el importe de la facturación que se debe aplicar a la empresa E_i. La especificación del algoritmo será, por lo tanto:

{ Pre: en la entrada se tiene una secuencia $s = S = <P_1 \cdot P_2 \cdot \ldots P_n \cdot$ 'FinPaginas'> **y** (toda P_i, $1 \leq i \leq n$, es una página) }

facturaPaginas
{ Post: crea la secuencia de salida $t = <E_1 \cdot F_1 \cdot E_2 \cdot F_2 \cdot ... \cdot E_m \cdot F_m>$
y (si $E_i \neq$ 'NoIdentificados', E_i es un nombre de empresa cliente que hay en S)
y (F_i es el importe correspondiente a la empresa E_i de todas las palabras no gratuitas que hay en las páginas de S que pertenecen a E_i)
y (todas las E_i presentes en t son diferentes)
y (si no hay ningún E_i que sea igual a "NoIdentificados",
m es el número de empresas diferentes de S) }
y (si existe una E_i igual a "NoIdentificados",
$m - 1$ es el número de empresas identificadas diferentes de S) }

Planteamiento del algoritmo principal

Identificamos el esquema que hay que aplicar, y hacemos abstracciones nuevas.

La solución consistirá en hacer que por cada página que obtenemos de la secuencia, identifiquemos a la empresa cliente, calculemos el importe de su factura y lo acumulemos en una estructura de datos que guarda a las empresas clientes identificadas con los importes facturados hasta el momento. Se trata, pues, de aplicar el esquema de recorrido sobre la secuencia de páginas. El tratamiento para cada página consistirá en actualizar una estructura de datos que mantiene la facturación de todas las empresas clientes que han ido apareciendo o aparecerán más adelante en S a medida que vayamos progresando. Denominaremos a este objeto *Facturas*, y ya veremos sus detalles en el nivel del diseño correspondiente.

Entendemos que la última página consistirá sólo en la palabra gratuita 'FinPaginas' y, por tanto, no será necesario tratarla.

Aplicación directa del algoritmo

Particularizamos la solución del esquema de recorrido para este problema.

Con todo esto, el algoritmo principal resulta de la forma siguiente:

```
algoritmo facturaPaginas;
   var
      facturas: tFacturas;
      pagina: tPagina;
   fvar
   inicializarFacturas(Facturas);
   obtenerPagina(Pagina);
   mientras no ultimaPagina(Pagina) hacer
      facturaPagina(Pagina, Facturas);
      obtenerPagina(Pagina)
   fmientras
   escribirFacturas(Facturas);
falgoritmo
```

Especificación de las acciones

accion obtenerPagina(**sal** p: tPagina)
{ Pre: la parte derecha de S contiene una subsecuencia que es una página, como mínimo, o la página especial "FinPaginas". Denominamos *página E* a esta subsecuencia. }
{ Post: P representa la página E }

funcion ultimaPagina (p: tPagina): **booleano**
{ Pre: $p = P$ }
{ Post: *ultimaPagina* (p) es **cierto** cuando P representa la última página }

accion inicializarFacturas(**sal** F: tFacturas)
{ Pre: }
{ Post: hay 0 empresas propietarias en F }

accion facturaPagina (**sal** p: tPagina; **entsal** f: tFacturas)
{ Pre: $p = P$ y $f = F$ y F contiene el importe total por propietarios de páginas procesadas anteriormente por *facturaPagina* }
{ Post: en f se ha acumulado el importe de P correspondiente al propietario de P. Si P no tiene propietario, el importe se ha acumulado en el propietario especial "NoIdentificados" }

accion escribirFacturas(**sal** f: tFacturas)
{ Pre: $f = F$ }
{ Post: crea una secuencia de salida con la estructura <n_1 i_1 n_2 i_2 ... n_m i_m>, donde n_i es una subsecuencia de caracteres que representa a un propietario de páginas F y acaba en blanco, e i_i es una subsecuencia de caracteres dígitos acabada en blanco que representa el importe total que hay en F para el propietario n_i }

2) Segundo nivel: *Páginas y Facturas*

En este nivel tenemos dos abstracciones: *pagina* y *facturas*.

a) Pasemos primero a trabajar la abstracción *página*:

– Abstracciones utilizadas: palabra.
– Identificación secuencia: página vista como una secuencia de palabras. La obtención de los elementos primero y siguiente se hará por medio de las acciones *leerPrimeraPalabra* y *leerPalabra*, y el reconocimiento de la palabra que marca el final será la palabra "FinTexto-HTML". En el caso especial de estar en la última página, ésta contendrá sólo "FinPáginas".

Definición de los tipos

Lo único que interesa de una página es conocer a su propietario y el importe de la factura. En el caso de la última página, lo único que interesa es saber que se trata de la última. Definiríamos, pues, el tipo *tPagina* de la forma siguiente:

```
tipo
tPagina = tupla
                propietario: tPalabra;
                importe: real;
                final: booleano;
          ftupla
ftipo
```

Planteamientos y soluciones

Identificación del esquema: recorrido con identificación previa de si se trata de la última página.

```
accion obtenerPagina (sal p: tPagina)
   var
      palabra, palabraNoIdentificados, palabraFinPaginas, palabraFinTextoHTML, palabra-
      Propiedad: tPalabra;
   fvar
   inicializaPalabraNoIdentificados(palabraNoIdentificados);
   inicializaPalabraFinPaginas(palabraFinPaginas);
   inicializaPalabraFinTextoHTML(palabraFinTextoHTML);
   inicializaPalabraPropiedad(palabraPropiedad);
   p.Importe := 0.0;
   p. Propietario := palabraNoIdentificadas; { Lo inicializamos en "NoIdentificados" por si
                                               acaso no tiene propietario }
   leerPrimeraPalabra(palabra);
   si palabrasIguales(palabra, palabraFinPaginas) entonces
      p.final := cierto
   sino
      p.final := falso
      mientras no palabrasIguales(palabra, palabraFinTextoHTML) hacer
        si palabrasIguales(palabra,palabraPropiedad) entonces
           leerPalabra(palabra);
           p.Propietario := palabra
        sino
           contabilizaPalabra(palabra, p.Importe);
           leerPalabra(palabra)
        fsi
      fmientras
   fsi
faccion { obtenerPagina }
```

Aunque en el enunciado se ha dejado claro que la secuencia de entrada está bien formada, podéis observar que en la solución dada, en el caso peor en que después de "propietario" no hubiese el nombre del propietario y se llegase al final de la secuencia, el propietario sería "FinTextoHTML".

```
funcion ultimaPagina(P: tPagina): booleano
    retorna p.final
ffuncion { ultimaPagina }
```

b) Consideremos ahora la facturación de empresas. Aquí el objetivo es que el objeto *Facturas* contenga el nombre de las empresas clientes presentes en la secuencia *S*, y para cada empresa cliente, el importe acumulado de las facturaciones de todas las páginas de *S* que pertenecen a la empresa cliente en cuestión.

Definición del tipo

Hay que mantener una tabla donde conste el nombre de cada empresa cliente y el valor del importe de la factura. El enunciado nos asegura que no habrá más de 50 empresas distintas. Debemos tener en cuenta que puede haber páginas sin identificación de propietario que serán para la empresa fantasma "NoIdentificados". Así pues, tenemos 51 empresas diferentes.

Dado el propietario de una página, tendremos que buscar en la tabla si ya está o si se le debe añadir. La búsqueda que utilizaremos para saber si una empresa cliente ya está necesitará una posición más de la tabla, como veremos más adelante. De este modo, la declaración de tipo será la siguiente:

```
const
   maxEmpresas: entero = 51;
fconst

tipo
   tFactura = tupla
                  nombreCliente; tPalabra;
                  importe: real;
              ftupla

   tFacturas =tupla
                  nEmpresas: entero;
                  f: tabla[ maxEmpresas + 1] de tFactura;
              ftupla
ftipo
```

Notad que...

... hemos utilizado un esquema de búsqueda por centinela. Por este motivo, hemos necesitado una posición de más. El centinela es siempre la empresa que buscamos en la tabla.

Planteamiento y soluciones

Identificación del esquema: búsqueda.

```
accion facturaPagina(sal p: tPagina, entsal f: tFacturas)
   var i: entero;
   fvar
   { Buscamos la empresa por centinela }
   f.f[F.nEmpresas + 1].nombreCliente := p.propietario;
   i := 1;
   mientras no palabrasIguales(f.f[i].nombreCliente, p.propietario) hacer
       i := i + 1
   fmientras
   si i = f.nEmpresas + 1 entonces
       { Damos de alta a la empresa e inicializamos el importe }
       f.nEmpresas := f.nEmpresas + 1;
       f.f[i].importe := p.importe
   sino { Añadimos el nuevo importe }
       f.f[i].importe := f.f[i].importe + p.importe;
   fsi
faccion { facturaPagina }
```

Aplicación directa

```
accion inicializarFacturas(sal f:tFacturas)
   f.nEmpresas := 0
faccion
```

Identificación del esquema: recorrido.

```
accion escribirFacturas(ent f: tFacturas)
   var i: entero;
   fvar
   i := 1;
   mientras i ≤ f.nEmpresas hacer
       escribePalabra(f.f[i].nombreCliente);
       escribirReal(f.f[i].importe);
       i := i + 1;
   fmientras
faccion { escribirFacturas }
```

Acciones pendientes de desarrollar

```
accion inicializaPalabraFinPaginas(sal m: tPalabra)
{ Pre: }
{ Post: m es la palabra "FinPaginas" }
```

```
accion inicializaPalabraFinTexto(sal m: tPalabra)
{ Pre: }
{ Post: m es la palabra "FinTexto HTML" }
```

```
accion inicializaPalabraPropiedad(sal m: tPalabra)
{ Pre: }
{ Post: m es la palabra "propiedad"}
```

```
accion inicializaPalabraNoIdentificados(sal m:tPalabra)
{ Pre: }
{ Post: m es la palabra "NoIdentificados" }
```

```
accion leerPrimeraPalabra(sal m: tPalabra)
{ Pre: en la entrada hay la secuencia de palabras s de la frase S. La parte izquierda de s está
vacía. La parte derecha, DF, no está vacía }
{ Post: m representa la primera palabra de DF. La palabra obtenida estará en la parte izquierda
de la secuencia s }
```

```
accion obtenerPalabra (entsal m: tPalabra)
{ Pre: En la entrada tenemos la secuencia de palabras s de la frase S. La parte derecha, DF, no
está vacía. La última palabra leída está en m }
{ Post: m representa la primera palabra de DF. La palabra obtenida estará en la parte izquierda
de la secuencia s }
```

```
accion contabilizaPalabra(ent palabra:tPalabra; entsal importe: real)
{ Pre: palabra = M e importe = I y l es el número de caracteres de M }
{ Post: palabra = M e importe = I + 50 si M es "<IMG",
   o importe = I + 5 si M no está acotada por "<" y ">" y l > 1 y M ≠ '<IMG',
   o importe = I si M está acotada por "<" y ">" o l ≤ 1 y M ≠ '<IMG' }
```

```
funcion palabrasIguales(m1: tPalabra; m2: tPalabra): booleano
{ Pre: m1 = M1 y m2 = M2 }
{ Post: palabrasIguales(m1,m2) es cierto cuando M1 y M2 representan exactamente la misma
palabra}
accion escribirPalabra(ent m:tPalabra)
{ Pre: m = M }
{ Post: se ha escrito la palabra M en el canal estándar de salida }
```

3) Tercer nivel: palabra

– Abstracciones utilizadas: ninguna.
– Identificación de secuencia: palabra vista como una secuencia de caracteres. La obtención de los elementos primero y siguiente se hará por medio de la función predefinida *leerCaracter*() y el reconocimiento del carácter que marca el final será el carácter espacio, o el igual, o el salto de línea.

Definición de los tipos

Podéis ver la solución del ejercicio anterior para los comentarios sobre el tipo.

En este nivel trabajamos con los tipos predefinidos del lenguaje y con el tipo *tPalabra* que definimos a continuación:

```
const
   maxCar: entero = 40;
fconst

tipo
   tPalabra = tupla
                  long: entero;
                  c: tabla[maxCar] de caracter;
                  caracterLeido: caracter;
              ftupla
ftipo
```

Planteamientos y soluciones

```
accion inicializaPalabraFinPaginas(sal m: tPalabra)
   m.long := 10;
   m.c[1] := 'F'; m.c[2] := 'i'; m.c[3] := 'n';
   m.c[4] := 'P'; m.c[5] := 'a'; m.c[6] := 'g'; m.c[7] := 'i'
   m.c[8] := 'n'; m.c[9] := 'e'; m.c[10] := 's'
faccion

accion inicializaPalabraFinTextoHTML(sal m: tPalabra)
   m.long := 12;
   m.c[1] := 'F'; m.c[2] := 'i'; m.c[3] := 'n';
   m.c[4] := 'T'; m.c[5] := 'e'; m.c[6] := 'x'; m.c[7] := 't'; m-c[8] := '0';
   m.c[9] := 'H'; m.c[10] := 'T'; m.c[11] := 'M'; m.c[12] := 'L'
faccion

accion inicializaPalabraPropiedad(sal m: tPalabra)
   m.long := 9;
   m.c[1] := 'p'; m.c[2] := 'r'; m.c[3] := 'o';
   m.c[4] := 'p'; m.c[5] := 'i'; m.c[6] := 'e';
   m.c[7] := 'd'; m.c[8] := 'a'; m.c[9] := 'd'

accion inicializaPalabraNoIdentificados(sal m: tPalabra)
   m.long := 15;
   m.c[1] := 'N'; m.c[2] := 'o'; m.c[3] := 'I'
   m.c[4] := 'd'; m.c[5] := 'e'; m.c[6] := 'n'
   m.c[7] := 't'; m.c[8] := 'i'; m.c[9] := 'f';
   m.c[10] := 'i'; m.c[11] := 'c'; m.c[12] := 'a';
   m.c[13] := 'd'; m.c[14] := 'o'; m.c[15] := 'S'
faccion
```

Identificación del esquema: recorrido.

```
accion leerPrimeraPalabra(sal m: tPalabra)
   var c: caracter;
   fvar
   m.long := 0;
   c := leerCaracter();
```

```
   mientras separador(c) y c ≠ '.' hacer { saltamos los primeros separadores }
      c := leerCaracter()
   fmientras
   m.caracterLeido := c;
   obtenerPalabra(m)
faccion
```

Identificación del esquema: recorrido.

```
accion leerPalabra(sal m: tPalabra)
   var
      i: entero; cNoSeparador: booleano;
      c: caracter;
   fvar
   m.long := 0;
   c := m.caracterLeido;
   mientras no (separador(c) y (m.long < maxCar)) hacer { obtenemos la palabra }
      m.long := m.long + 1;
      m.c[m.long] := c;
      c := leerCaracter()
   fmientras
   mientras no separador(c) hacer { saltamos los caracteres que sobran }
      c := leerCaracter()
   fmientras
   mientras separador(c) hacer { saltamos los separadores }
      c := leerCaracter()
   fmientras
   m.caracterLeido := c
faccion
```

Observad que...

... la acción *leerPalabra* tiene en cuenta la posibilidad de que la palabra leída sea más larga que *maxCar*. La palabra se lee, pero los caracteres de la palabra que están más allá de *maxCar* en la tabla no se guardan. Dado que sólo nos interesa comprobar las palabras que tienen *maxCar* caracteres, como mucho, el truncamiento de la palabra no tiene ningún efecto en el resto del algoritmo y permite aceptar palabras tan largas como se quiera.

Aplicación directa de la alternativa

```
accion contabilizaPalabra(ent palabra: tPalabra; entsal importe: real)
   var palabraImagen: tPalabra;
   fvar
   inicializaPalabraImagen(palabraImagen);
   si palabrasIguales(palabra, palabraImagen) entonces
      importe:= importe + 50
   sino
      si no(palabraEntreMenoresMayores(palabra) y longitudPalabra(palabra) > 1) entonces
            importe := importe + 5;
      fsi
   fsi
faccion
```

Identificación del esquema: búsqueda.

```
funcion palabrasIguales(m1: tPalabra, m2: tPalabra): booleano
   var
      i: entero;
      iguales: booleano;
   fvar
   si m1.long = m2.long entonces
      si m1.l > 0 entonces
         i := 1;
         mientras (m1.c[i] = m2.c[i]) y (i < m1.l) hacer i := i + 1
         fmientras
         iguales := m1.c[i] = m2.c[i]
      sino iguales := cierto
      fsi
   sino iguales := falso
   fsi
   retorna iguales
ffuncion { palabrasIguales }
```

Identificación esquema: recorrido.

```
accion escribePalabra(ent m: tPalabra)
   var i : entero;
   fvar
   i := 1;
   mientras i ≤ m.long hacer
      escribirCaracter(m.c[i]);
      i := i + 1
   fmientras
   escribirCaracter(' ')
faccion { escribePalabra }
```

Acciones pendientes de desarrollar

El diseño de este nivel ha dado pie a unas funciones auxiliares que detallamos a continuación.

accion inicializaPalabraImagen(**sal** m: tPalabra)
{ Pre: }
{ Post: *m* es la palabra <IMG }

funcion palabrasEntreMenoresMayores(m: tPalabra): **booleano**
{ Pre: *m* = *M*}
{ Post: *palabrasEntreMenoresMayores* es **cierto** cuando el primer carácter de la palabra es < y el último es > }

funcion longitudPalabra(m: tPalabra): **entero**
{ Pre: *m* = *M*}
{ Post: *longitudPalabra(m)* da el número de caracteres que tiene *M* }

funcion separador(c: **caracter**): **booleano**
{ Pre: *c* = *C* }
{ Post: *separador*(*c*) es **cierto** si *C* es espacio o *C* es un igual o *C* es algún carácter de retorno o de salto de línea }

```
accion inicializaPalabraImagen(sal m: tPalabra)
   m.long := 4;
   m.c[1] := '<'; m.c[2] := 'I'; m.c[3] := 'M'; m.c[4] := 'G'
faccion

funcion palabraEntreMenoresMayores(ent m:tPalabra): booleano
    retorna m.c[1] = '<' y m.c[m.long] = '>'
ffuncion

funcion longitudPalabra(ent m:tPalabra): entero
    retorna m.long;
ffuncion
{ c = C }

funcion separador(c: caracter): booleano
    retorna c = ' ' o c = '=' o c = codigoACaracter(10) o c = codigoACaracter(13)
ffuncion
```

La función *separador*...

... deberá tener en cuenta los convenios seguidos para los caracteres de control de retorno de línea y de salto de línea. Estos caracteres en ASCII son los códigos 10 y 13.

Glosario

abstracción
Simplificación de una realidad teniendo en cuenta sólo una de sus partes y considerando sus atributos y cualidades relevantes para la tarea que hay que realizar.

complejo
No simple. Dicho de todo aquello que comprende partes o elementos discernibles.

descomposición de un problema
Separación e identificación de subproblemas más elementales que el problema original.

diseño descendente
Metodología que descompone un problema complejo en subproblemas menos complejos que el original. La descomposición es guiada por abstracciones naturales y convenientes al problema, y da como resultado un conjunto de subproblemas independientes y más concretos. Se procede de idéntica forma para cada uno de los subproblemas, hasta llegar a subproblemas lo suficientemente concretos y elementales.

nivel de abstracción
Nivel en el que se concreta una abstracción utilizada para descomponer un problema o subproblema. Nivel de abstracción y nivel de descomposición son dos conceptos distintos y que pueden coincidir en la mayoría de los casos, pero no necesariamente.

nivel de descomposición
Nivel propio de una etapa de la descomposición de subproblemas en subproblemas más concretos. El primer nivel corresponde al desarrollo del problema original. La descomposición de los subproblemas generados en el primer nivel se desarrollará en un segundo nivel, y así sucesivamente.

Bibliografía

Castro, J.; Cucker, F.; Messeguer; Rubio, A.; Solano, L.; Vallés, B. (1992). *Curso de Programación*. Madrid, etc.: McGraw Hill.

Sebastià Vila, M. (1995). *Programació Fonamental. Problemes*. Barcelona: Edicions UPC.

Wirth, N. (1982). *Introducción a la Programación Sistemática*. Buenos Aires: El Ateneo.