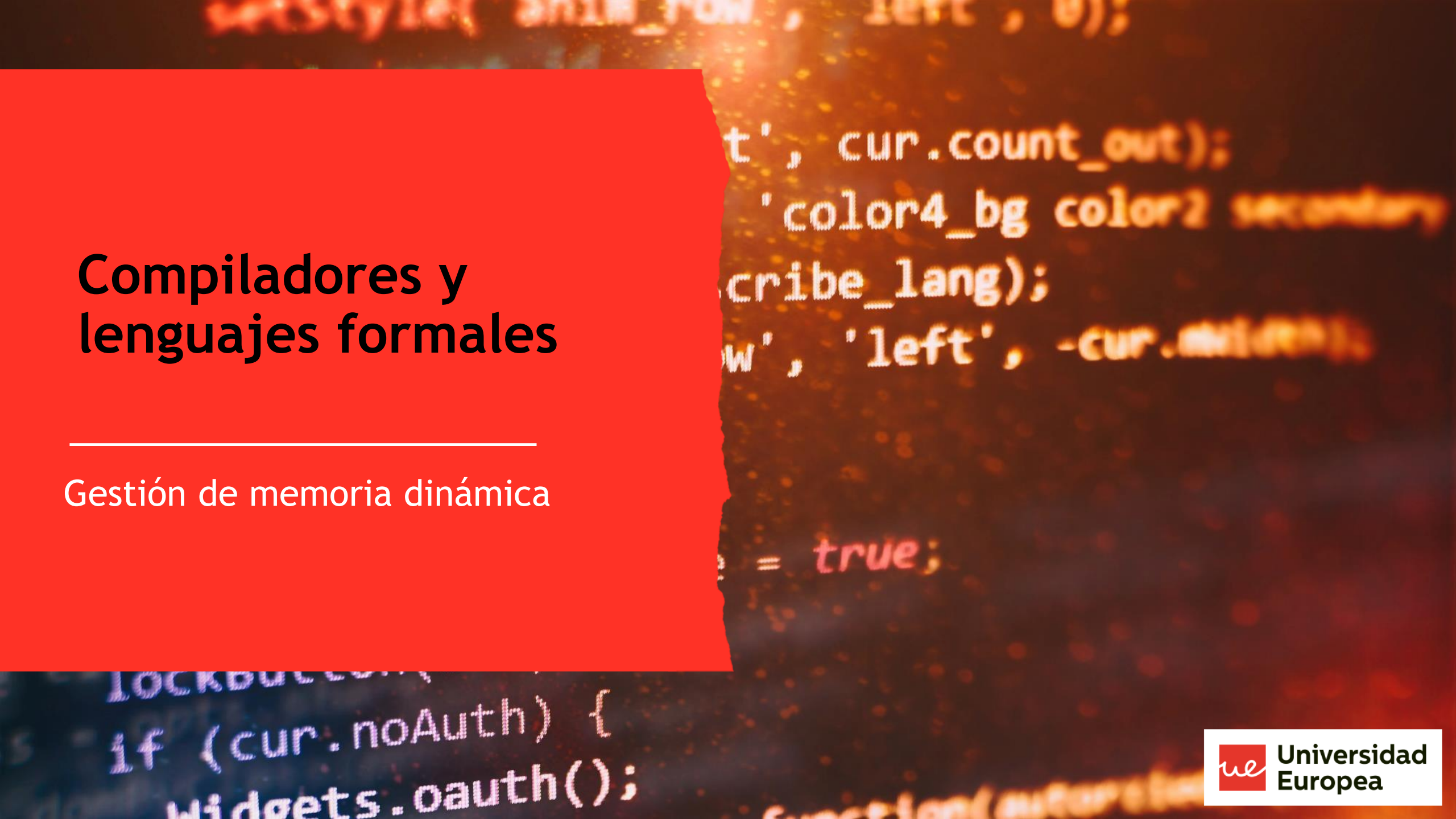
Compiladores y lenguajes formales

Gestión de memoria dinámica



Universidad Europea

Contenidos

©Copyright Universidad Europea Todos los derechos reservados.

1. Objetivos

Los objetivos que se pretenden alcanzar en este recurso son los siguientes:

- Entender la manera en la que un programa en ejecución organiza y utiliza la memoria lógica.
- Comprender el funcionamiento de la memoria de pila.
- Estudiar los distintos algoritmos que se pueden aplicar para gestionar la memoria de heap, prestando especial atención a las diferencias entre los procesos liberación de memoria por parte de un programador y la liberación implícita que realizan los recolectores de basura.

©Copyright Universidad Europea Todos los derechos reservados.

2. Introducción

Para que un programa se pueda ejecutar, el compilador debe cooperar con el sistema operativo para crear y gestionar un entorno de ejecución que debe lidiar con multitud de aspectos básicos de cualquier lenguaje de programación, entre los cuales podemos encontrar ejemplos como los siguientes:

- Reserva de espacio en memoria para variables y objetos nombrados en el programa.
- Mecanismos de acceso a las variables.
- Llamadas entre procedimientos y funciones.
- Paso de parámetros y devolución de resultados.
- Llamadas al sistema operativo.

©Copyright Universidad Europea Todos los derechos reservados.

3. Tipos de memoria

De forma general, un programa en ejecución determina y gestiona direcciones de memoria lógicas, que son posteriormente convertidas por el sistema operativo en direcciones físicas. En lenguajes imperativos el espacio para la memoria lógica se subdivide de la siguiente manera:

1. Memoria ocupada por el código del programa.
2. Memoria ocupada por los datos estáticos, que puede ser establecida en tiempo de compilación considerando únicamente el código del programa, como es el caso de las variables globales.
3. Memoria dinámica, que se va reservando y liberando cuando el programa se encuentra en ejecución. Generalmente, se subdivide en *Stack* y *Heap*, aunque comparten el espacio:
    a. Memoria de pila (*Stack*), que da soporte por ejemplo al paso y devolución de parámetros.
    b. Memoria de montón (*Heap*), que permite al programador la reserva y liberación de espacio en memoria.

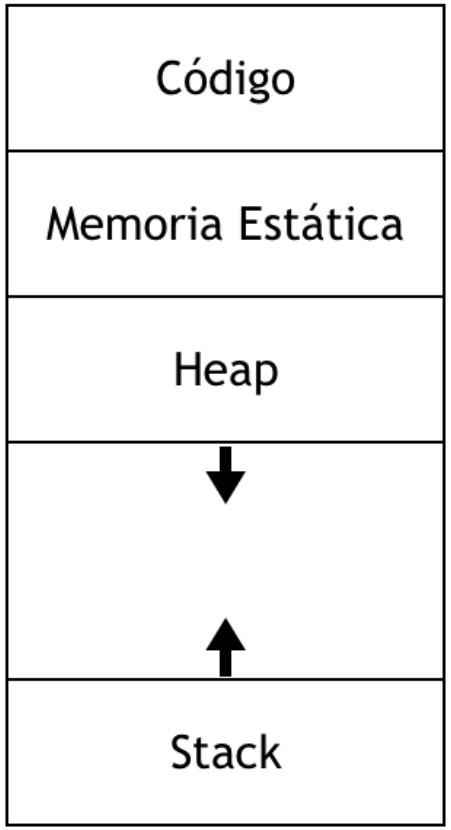


Figura 1.

©Copyright Universidad Europea Todos los derechos reservados.

4. Memoria estática

Cuando un programa entra en ejecución, el compilador deber reservar una serie de bloques en memoria que garanticen el almacenamiento del código del programa y de las variables declaradas dentro de él. Estos bloques no deben ser modificados ni accedidos por otros programas ni por el propio sistema.

Definiremos entonces *variables estáticas* como aquellas creadas en la sección declarativa de un programa, cuyo espacio de memoria es reservado al comienzo de la ejecución y que existen hasta su conclusión.

Generalmente este espacio se define de forma secuencial, de manera que el compilador solamente necesita conocer la primera posición de la memoria estática e ir colocando las variables a partir de ella, tanto para leerlas como para modificarlas.

El único cambio que se puede producir en una variable estática es, por tanto, su contenido, y solamente a través de sentencias de asignación. Esto implica que, por ejemplo, en el caso de los *arrays*, éstos no puedan cambiar su tamaño en tiempo de ejecución y, por tanto, solo puedan almacenar el número de elementos y el tipo definidos cuando fueron declarados.

©Copyright Universidad Europea Todos los derechos reservados.

5. Memoria de pila (*Stack*)

En general, los compiladores para lenguajes que utilizan procedimientos, funciones o métodos como unidades lógicas de agrupación de sentencias gestionan una parte de su memoria de ejecución mediante una pila.

Cada vez que se invoca a una de estas subrutinas se reserva espacio para sus variables locales en la pila, y cuando la subrutina termina, ese espacio se libera de la pila. A este proceso de ejecución de una subrutina lo denominamos *activación*.

Debido a la naturaleza jerárquica con la que se suelen disponer los procedimientos y funciones, la ejecución de una subrutina en muchos casos implicará la llamada a otras subrutinas, quedando la primera en suspenso hasta que se termina la ejecución de la segunda y generando una suerte de "árbol de activaciones" que proporciona una foto de todo el proceso de ejecución del programa

©Copyright Universidad Europea Todos los derechos reservados.

5. Memoria de pila (*Stack*): ejemplo de árbol de activación

Supongamos el siguiente programa que calcula el número de Fibonacci para una posición *n* dada.

```
int fibonacci(int n) {
    int f = 0;

    if (n <= 1)
        f = 0
    else
        f = fibonacci(n-1) + fibonacci(n-2);
    return f;
}
main() {
    fibonacci(2)
}
```


©Copyright Universidad Europea Todos los derechos reservados.

5. Memoria de pila (*Stack*): ejemplo de árbol de activación

Las secuencia de invocaciones a las funciones se corresponde con el recorrido en preorden del árbol de activación, mientras la secuencia de *returns* se puede obtener a través de su recorrido en postorden.

En un momento determinado de la ejecución, las activaciones que están abiertas son aquéllas que se corresponden con el nodo actual y sus ancestros.

```
main()
     fibonacci(3)
          fibonacci(2)
               fibonacci(1)
               fibonacci(0)

          fibonacci(1)
```

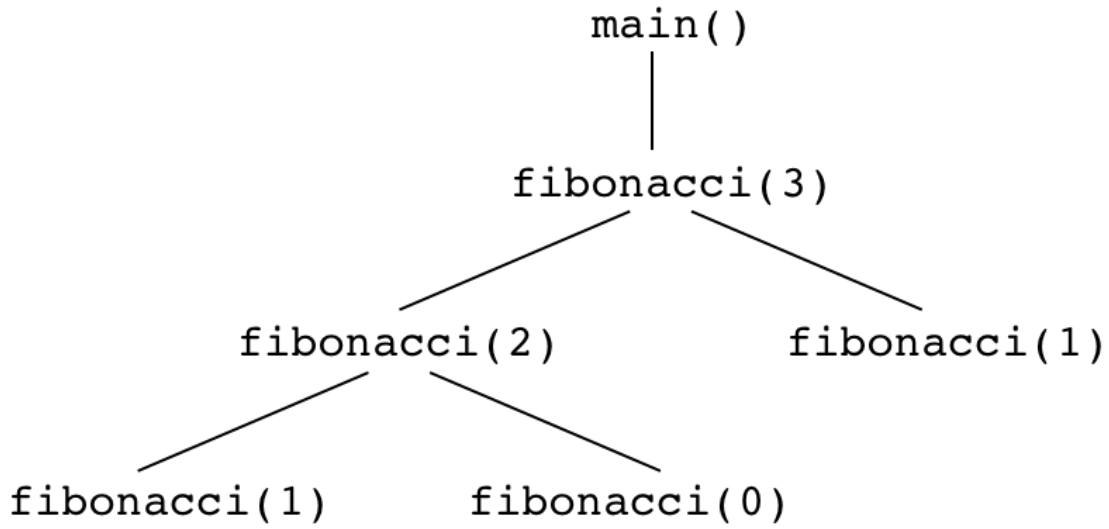


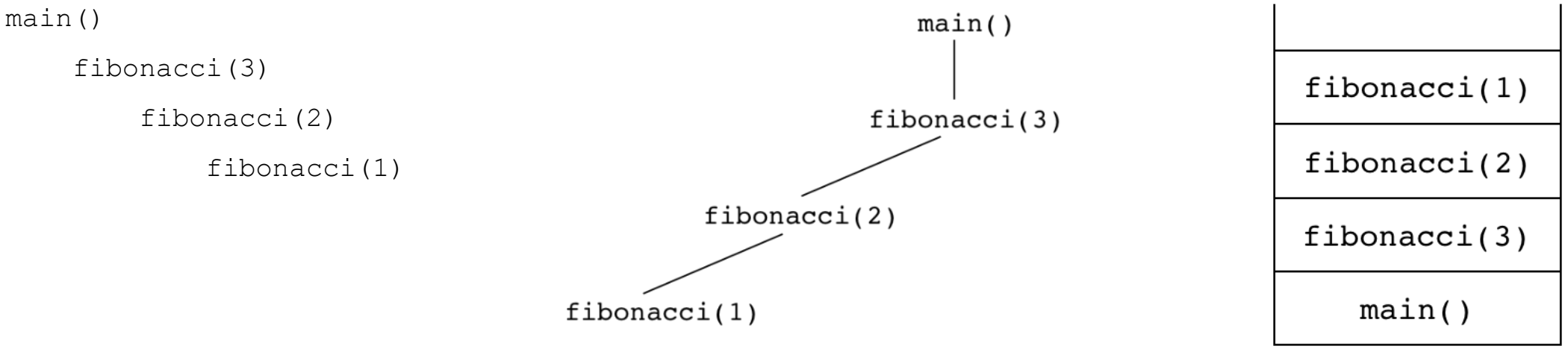
Figura 2.

©Copyright Universidad Europea Todos los derechos reservados.

5. Memoria de pila (*Stack*): pila de control

Estas invocaciones y *returns* generalmente son gestionadas por la denominada *pila de control*, que almacena *registros de activación* de manera que el registro inicial (la raíz del árbol de activación) permanece en la base de la pila, y la secuencia de activaciones correspondientes al camino activo en el árbol se dispone de forma secuencial hacia la cima de la pila.

En el ejemplo anterior podríamos encontrarnos una situación como la siguiente en un momento intermedio de la ejecución:



Punto intermedio de ejecución. Árbol de activación. Pila de control.

Figura 3.

©Copyright Universidad Europea Todos los derechos reservados.

5. Memoria de pila (*Stack*): registros de activación

Aunque los contenidos de un registro de activación pueden variar con el lenguaje implementado, por lo general contienen los siguientes elementos.

- Valores de los parámetros pasados en la llamada a la subrutina.
- Valor devuelto, cuando se trata de una invocación a una función.
- Un puntero de control hacia el registro de activación del invocador.
- Un puntero de acceso hacia datos disponibles en otro espacio de memoria.
- El estado del sistema antes de la invocación a la subrutina.
- Variables locales a la subrutina asociada a este registro de activación.
- Valores temporales, generalmente asociados a la evaluación de expresiones

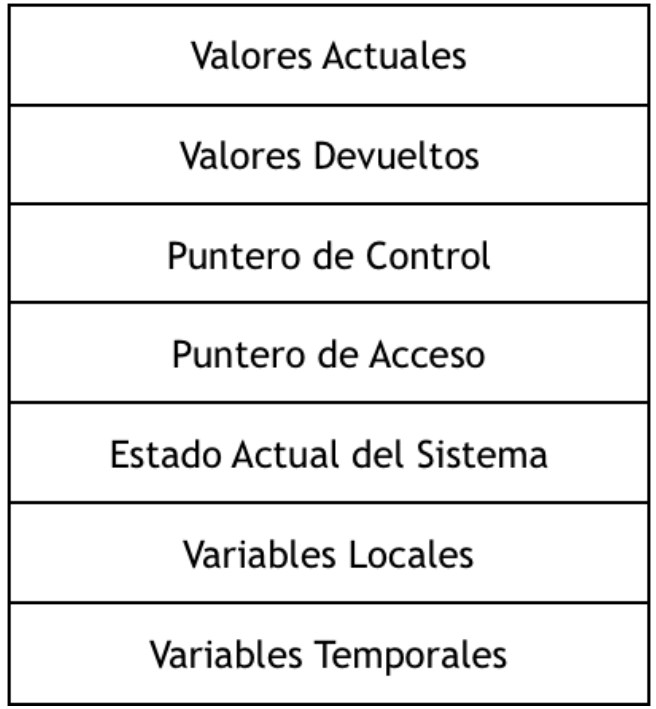


Figura 4. Distribución de un registro.

©Copyright Universidad Europea Todos los derechos reservados.

5. Memoria de pila (Stack): registros de activación

El siguiente ejemplo muestra el estado de la pila de control en un punto de la ejecución del cálculo de Fibonacci. Para una mayor claridad del proceso, se han utilizado registros de activación simplificados.

- Cuando el programa ejecuta la primera invocación desde el `main()`, se activa la función `fibonacci(n)` y se inserta en la pila su registro de activación.
- Cuando la ejecución de `fibonacci(2)` invoca a `fibonacci(1)`, se inserta en la pila el registro correspondiente. Sin embargo, cuando el control vuelve de `fibonacci(1)`, este registro se elimina de la pila.
- Después, `fibonacci(2)` invoca a `fibonacci(0)`, quedando el flujo en el estado de la figura.

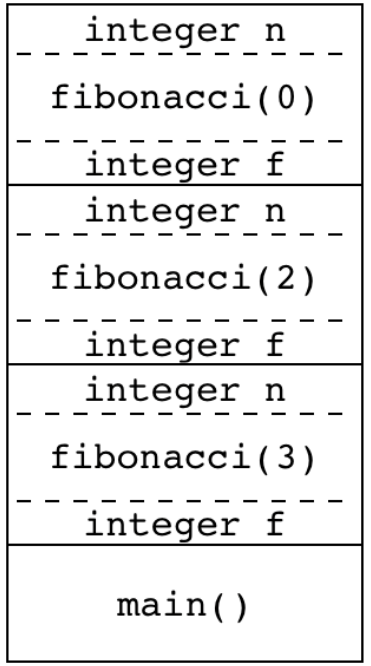


Figura 5. Pila de control.

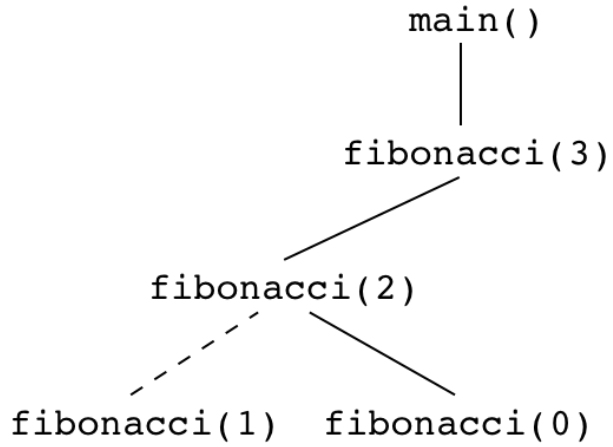


Figura 6. Árbol de activación.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap)*

En esta área de la memoria se almacenan los datos que tienen una vida indefinida, hasta que el programador o el propio programa los eliminan. Por ejemplo, tanto C++ como Java permiten crear nuevos objetos a través del operador `new`, y pasarlos de método a método sin que cada uno de los métodos tenga una copia del objeto original, sino un puntero al objeto original. Éstos objetos se almacenan en el *heap*.

Por otro lado, mientras en C++ deberíamos borrar de forma activa estos objetos a través del operador `delete`, en el caso de Java entra en juego lo que se denomina como un recolector de basura (*garbage collector*), que es un proceso encargado de encontrar espacios de memoria que ya no son utilizados por objetos activos y liberarlos para reutilizarlos en la creación de nuevos objetos.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap):* el gestor de memoria

El gestor de memoria es un sistema que se encarga de llevar un registro del espacio libre en memoria en todo momento, permitiendo realizar dos funciones básicas:

- Reserva (*allocation*). Cuando un programa solicita memoria para una variable u objeto, el gestor de memoria genera un espacio contiguo de memoria del tamaño necesario.
- Liberación (*deallocation*). El gestor de memoria libera espacio ocupado y lo añade al espacio libre, para que pueda ser reutilizado.

La complejidad de este sistema viene dada por dos razones principales:

1. Cada petición de memoria suele tener un tamaño diferente, por lo que la reserva y liberación de memoria termina llevando a situaciones donde la memoria está fragmentada.
2. Cada objeto tiene una vida da duración diferente, por lo que no se puede utilizar un enfoque similar al de la pila.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap):* fragmentación

Para entender mejor el problema que supone la fragmentación de la memoria, vamos a analizar primero el funcionamiento que debería proporcionar el gestor de memoria si todos los bloques se solicitaran siempre con un tamaño fijo. En este caso, el *heap* solamente necesitaría gestionar dos listas de memoria: una con la memoria reservada y otra con la memoria libre.

La siguiente figura muestra un ejemplo donde se ha reservado memoria para 4 objetos (A, B, C y D) y posteriormente se ha liberado la memoria que ocupaba el objeto C, y que se encontraba en la posición 3. Si por ejemplo ahora quisiéramos liberar memoria para un nuevo objeto E, el sistema muy bien podría asignarnos la posición C, porque tenemos garantizado que el tamaño del objeto será exactamente el mismo.

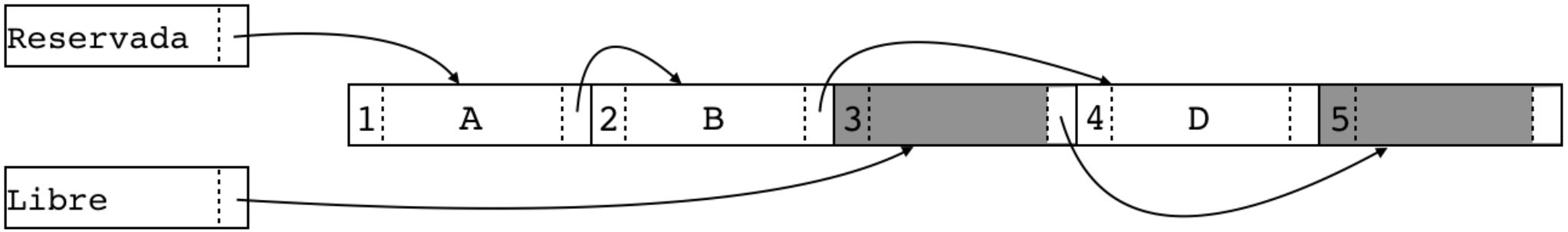


Figura 7.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap):* fragmentación

Supongamos ahora un caso similar, pero en el que cada bloque pueda tener un tamaño diferente que se debe especificar cuando se realiza una petición para reservar memoria. Esto está reflejado en la figura del ejemplo con un nuevo campo que contiene el tamaño en bytes.

Si en la situación actual tratáramos de insertar un nuevo campo con un tamaño de 25 bytes y, suponiendo que el de la figura es todo el espacio de memoria disponible para el *heap* (en este caso, 100 bytes), el gestor no podría realizar la reserva, puesto que, pese a que existen más de 25 bytes disponibles si sumamos las posiciones 3 y 5, un objeto debe ocupar un único espacio de memoria contiguo.

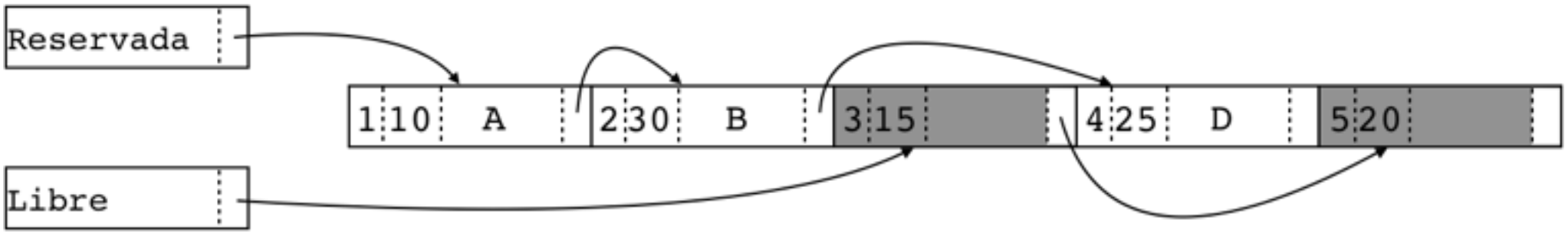


Figura 8.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap):* fusión *(Coalescence)*

Con el fin de evitar, o al menos minimizar, el impacto de la fragmentación, los sistemas de gestión de memoria utilizan técnicas para fusionar espacios libres cada vez que se libera un bloque. Simplificando, la idea es detectar cuando dos o más bloques contiguos están libres y unirlos para formar un único bloque.

Una opción sería recorrer la lista de bloques libres cada vez que se produjera una liberación de memoria, pero este método resultaría poco eficiente en tiempo, de manera que lo que se suele hacer es añadir un nuevo campo a cada bloque que simplemente indica si está libre u ocupado (en color rojo en la figura).

Siguiendo con los ejemplos anteriores, si eliminamos el objeto D, el gestor detectaría que las posiciones 3, 4 y 5 ahora quedan libres, y las podría fusionar para formar un solo bloque.

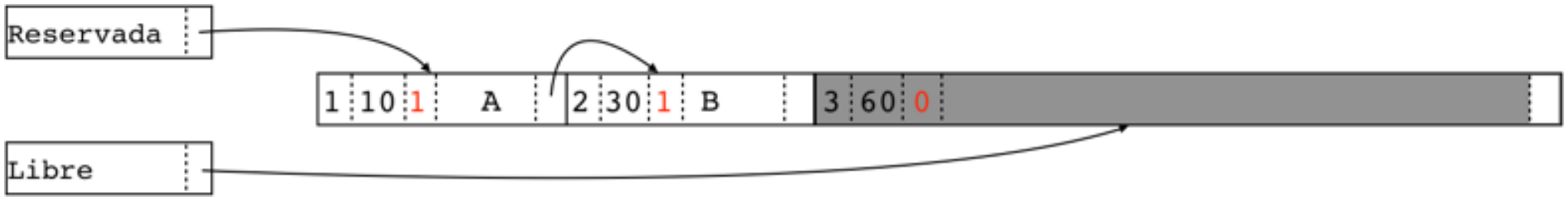


Figura 9.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap):* recolección de basura *(Garbage Collection)*

La liberación implícita de memoria (conocida generalmente como recolección de basura) es un proceso de liberación automática de la memoria que ya no está en uso en la ejecución del programa. Este proceso alivia al programador de la responsabilidad de liberar memoria. Algo que suele ser proclive a errores y lleva a situaciones en las que se agota la memoria del sistema de manera innecesaria.

La mayoría de los lenguajes modernos de programación incorporan un recolector de basura, como pueden ser Java, Python Haskell o Perl.

Existen dos aproximaciones principales para afrontar la recolección de basura, atendiendo a la manera en la que se determina que un bloque de memoria pertenece al denominado como conjunto de basura (o *garbage set*).

1. Se consideran como pertenecientes al garbage set todos aquellos bloques a los que no enlaza ningún puntero.
2. Se incluyen en el conjunto todos los bloques que no son alcanzables desde algún objeto que no pertenezca al heap.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap): reference Counting*

Reference counting es un método de recolección de basura que almacena en cada bloque el número de punteros que lo referencian. Por tanto, cuando este valor decae a cero, se declara el bloque como basura.

El contador de referencias se puede actualizar en diversas situaciones:

- Cuando se reserva la memoria para crear un bloque, se inicializa su valor a 1.
- Cada vez que se crea una copia del puntero al bloque, su contador de referencias se incrementa en 1.
- Cuando un puntero a un bloque se elimina, su contador de referencias se disminuye en 1.

Conviene resaltar que, en el caso de que el recolector de basura libere la memoria de un bloque que contenga referencias a otros bloques de memoria, esto últimos no quedan automáticamente liberados. Para ello, el recolector deberá detectar posteriormente que estos bloques tengan a su vez el contador de referencias a cero y eliminarlos individualmente.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap): reference Counting*

La siguiente figura muestra un ejemplo en el que se acaba de eliminar la única referencia que existía al objeto B.

Con el método *Reference Counting* el recolector de basura detectará que B tiene cero punteros de entrada y, por tanto, lo eliminará.

Después de eliminarlo, disminuirá en 1 el contador de referencias de E y, al detectar que su valor queda en cero, procederá también a su liberación.

Nótese que, si se eliminara el puntero al objeto A, su contador de referencias quedaría en 1, pero nunca sería liberado, de la misma manera que D, debido a la referencia circular que se produce entre ambos.

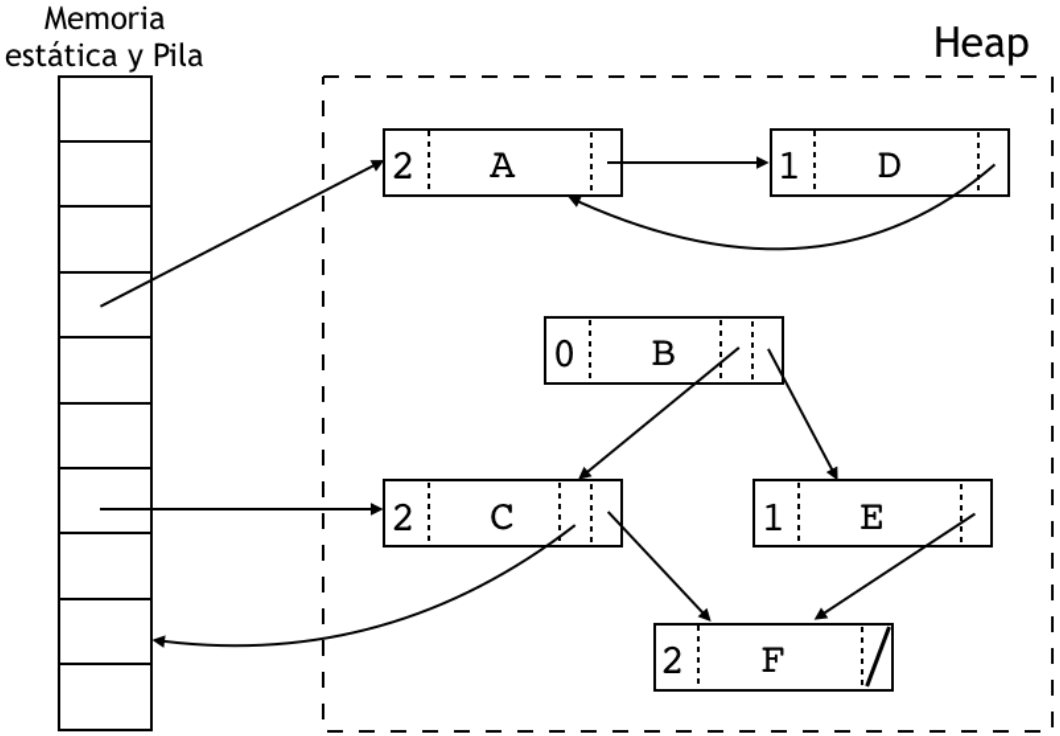


Figura 10. Árbol de activación.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap): Mark and Sweep (o Mark and Scan)*

Este método de recolección de basura mejora los resultados del *Reference Counting* al ser capaz de liberar incluso la memoria circular.

Este algoritmo comprende dos fases:

1. Fase de marcado, en la que se marcan todos los bloques que son alcanzables.
2. Fase de barrido, en la que se explora toda la memoria reservada y libera la memoria de todos aquellos bloques que no estén marcados como alcanzables.

Este algoritmo se suele combinar con una última fase de compactación (*compaction*), que a grandes rasgos consiste en mover todos los bloques reservados a una zona de memoria concreta y los bloques libres a otra zona distinta. Este proceso convierte la recolección de basura en algo menos eficiente en el tiempo, pero evita la fragmentación y es la mejor manera de recuperar toda la memoria no utilizada

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap): Mark and Sweep* (Fase de marcado)

La fase de marcado se basa en dos principios básicos: los bloques enlazados directamente desde la memoria estática y la de pila (a partir de ahora, memoria de programa) se consideran como alcanzables, y los bloques alcanzables desde un bloque alcanzable se consideran su vez como alcanzables.

Durante el proceso de marcado se parte por tanto de la memoria de programa y, de forma recursiva, se van marcando todos los bloques alcanzables desde sus punteros, siguiendo un recorrido en profundidad sobre un grafo. Si este recorrido encuentra un bloque sin punteros de salida, o que ya ha sido marcado, vuelve hacia atrás y continúa con el recorrido.

Como el número de bloques alcanzables es finito y ningún bloque se procesa más de una vez, este proceso de marcado se realiza en un tiempo lineal.

Para representar el estado marcado, en lugar de utilizar el campo *contador de referencias* utilizaremos un campo *marcado*.

©Copyright Universidad Europea Todos los derechos reservados.

6. Memoria de montón *(Heap): Mark and Sweep* (Fase de barrido)

Una vez marcados todos los bloques alcanzables, el algoritmo ya solo debe recorrer los bloques y realizar dos posibles acciones:

1. Si está marcado como alcanzable, limpia la marca de cara a la próxima fase de marcado.
2. Si no está marcado como alcanzable, libera la memoria.

Podemos aprovechar el proceso para ir combinando las posiciones adyacentes que queden libres, generando por tanto menos bloques libres de memoria, pero de mayor tamaño.

En la figura podemos ver cómo el único bloque que quedaría como no marcado sería el bloque B.

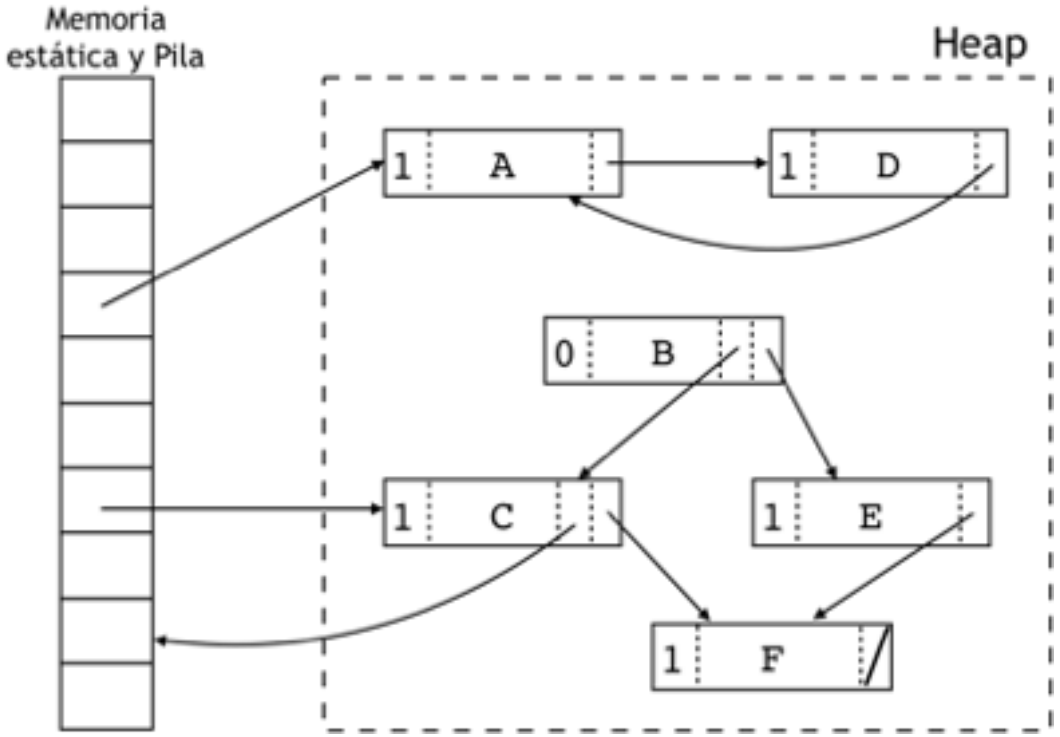


Figura 11. Árbol de activación.

©Copyright Universidad Europea Todos los derechos reservados.

7. Conclusiones

1. Un programa en ejecución debe gestionar tres tipos de memoria: estática, de pila y de montón.

2. El almacenamiento de los parámetros de llamada a las subrutinas y las variables locales se realiza en la pila de control.

3. El *heap* se encarga de almacenar los datos que tienen un tiempo de vida indefinido y necesitan ser eliminados de forma explícita por el programa.

4. La recolección de basura es un proceso de liberación automática de la memoria que ya no está en uso en la ejecución del programa, y alivia al programador de la tarea de liberar memoria.

©Copyright Universidad Europea Todos los derechos reservados.

8. Referencias bibliográficas

1. Aho, Alfred V., Lam, Monica S., Sethi, Ravi, Ullman, Jeffrey, 2007. *Compilers: Principles, Techniques, and Tools*. Addison Wesley.
2. Appel, Andrew W, 2002. *Modern Compiler Implementation in Java*. Cambridge University Press.
3. Grune, Dick, van Reeuwijk, Kees, Bal, Henri E., Jacobs, Ceri, 2012. *Modern Compiler Design*. Springer.

©Copyright Universidad Europea Todos los derechos reservados.

© Todos los derechos de propiedad intelectual de esta obra pertenecen en exclusiva a la Universidad Europea de Madrid, S.L.U. Queda terminantemente prohibida la reproducción, puesta a disposición del público y en general cualquier otra forma de explotación de toda o parte de la misma.
La utilización no autorizada de esta obra, así como los perjuicios ocasionados en los derechos de propiedad intelectual e industrial de la Universidad Europea de Madrid, S.L.U., darán lugar al ejercicio de las acciones que legalmente le correspondan y, en su caso, a las responsabilidades que de dicho ejercicio se deriven.