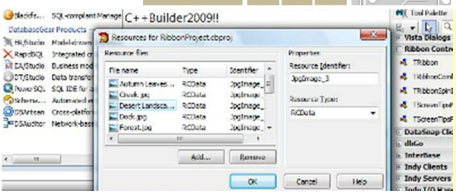


Compiladores 2

TEMA 1 LENGUAJES INTERPRETADOS

Tema 1
Lenguajes interpretados

1. INTÉRPRETES

El objetivo de los compiladores es poder traducir, sin errores, el código de un lenguaje de programación a código máquina que pueda ejecutar la CPU. El diferente conjunto de instrucciones que puede ejecutar cada CPU se denomina código máquina.

Compilador	Intérprete
Se compila una vez, se ejecuta *n* veces.	Cada vez que se ejecuta se traduce.
El proceso de compilación puede tener una visión global del programa y, por lo tanto, la gestión de errores es más eficiente.	Permite más interacción con el código en tiempo de ejecución.
La ejecución es más rápida.	Necesita menos memoria.

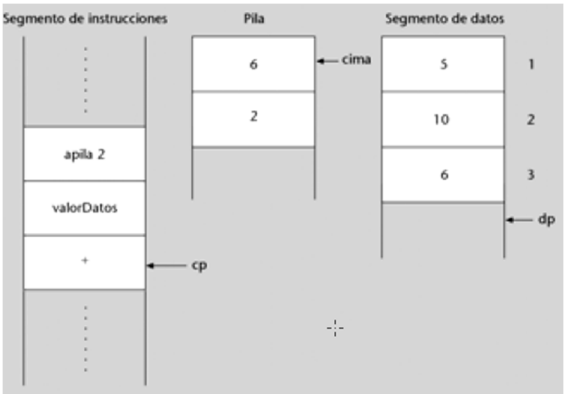
Un intérprete es un tipo de traductor que no genera código objeto equivalente al código fuente, sino que analiza y ejecuta, una a una, las sentencias del programa fuente. Un intérprete sigue un proceso iterativo en el cual toma cada sentencia del programa fuente y la ejecuta sobre la plataforma; cada sentencia implica una ejecución, en caso de existir errores en la sentencia, se tiene que notificar.

Aunque puede ser útil emplear un lenguaje interpretado en alguna fase del desarrollo, los lenguajes compilados son mucho más rápidos de ejecutar. Muchos entornos integrados de programación actuales ofrecen tanto intérpretes como compiladores para un mismo lenguaje de programación. Esto permite utilizar el compilador para compilar y detectar el máximo número de errores, y el intérprete para la depuración y las pruebas. Una vez el programa está acabado, se llevará a cabo la compilación definitiva para poner la aplicación en producción.

Todavía existe otro modo de combinar los compiladores y los intérpretes para generar un nuevo tipo de traductor, a medio camino entre un compilador y un intérprete: el intérprete precompilado o compilador interpretativo. En este tipo de herramientas, primero se compila y se genera una especie de código intermedio y, después, se pasa a un intérprete que lo ejecuta sobre la plataforma.

2. LA MÁQUINA DE PILA ABSTRACTA

La gran cantidad de particularidades que se pueden encontrar a la hora de implementar el intérprete, tales como CPU diferentes, restricciones de disponibilidad de memoria, segmentos de memoria distintos para los datos y las instrucciones, arquitectura de los registros del sistema, restricciones de seguridad (como acceso restringido al disco), etc., provocan que la arquitectura de un intérprete sea muy dependiente de la plataforma de ejecución y que sea difícil encontrar una arquitectura homogénea para estos ingenios.



Un buen modo de estudiar el funcionamiento de un intérprete es utilizar un programa muy simple que permita interpretar notación postorden: la máquina de pila abstracta. Una máquina de pila abstracta, como la de la imagen, se basa en una estructura de pila, una memoria independiente para las instrucciones y los datos y tres conjuntos de instrucciones: las aritméticas, las de manipulación de la pila y las de control de flujo.

Así, para calcular la expresión (2+4)*5, se ejecutarían las siguientes instrucciones (24+5*):

1. Lee el valor 2 y lo inserta en la pila
2. Lee el valor 4 y lo inserta en la pila

3. Lee el oprador +, extrae los dos primeros valores de la pila, los suma e inserta el resultado (6)
4. Lee el valor 5 y lo inserta en la pila
5. Lee el operador *, y extrae los dos primeros valores de la pila, ejecuta la operación e inserta el resultado (30).

La máquina abstracta trabaja con números enteros y soporta las instrucciones aritméticas habituales; cada uno de estos operadores aritméticos (+, -, *, /) define una operación que se ha de aplicar sobre los dos primeros valores de la cima de la pila. Cuando la máquina lee un operador, extrae los dos primeros valores de la pila, aplica la operación e introduce en resultado en la pila.

El conjunto de instrucciones está formado por 18 elementos. En binario se pueden codificar utilizando 5 bits, con lo que se pueden generar 32 elementos (2^5); por tanto, en una codificación de 5 bits, todavía dispondríamos de 14 posiciones libres para ampliar el conjunto de instrucciones.

Dado que un intérprete para la máquina de pila abstracta se podría implementar para muchas máquinas diferentes, es muy importante que la especificación recoja aspectos propios del hardware como el tamaño en bits de los datos o el tamaño de la memoria. Es responsabilidad del intérprete compatibilizar estas especificaciones con el hardware específico sobre el cual se ejecuta. Por ello los intérpretes para lenguajes de programación portables en diferentes plataformas también se llaman **máquinas virtuales**.

Es posible implementar un intérprete obviando algunas de estas cuestiones si se programa mediante un lenguaje portable como Java. Un intérprete programado en Java ya es portable porque el mismo lenguaje asegura un mismo comportamiento del código sobre distintas plataformas: por ejemplo, un entero en Java siempre tendrá un tamaño de 32 bits, sea cual sea la máquina sobre la cual se ejecute el programa.

TEMA 2 LENGUAJES DE MARCAS

1. LA ESPECIFICACIÓN XML

La publicación de páginas mediante HTML ha sido muy útil durante muchos años, pero su principal virtud, que es su publicación por Internet, se convierte tiempo después en su principal defecto, pues quizá nos interesaría optimizar páginas para pantallas de ordenador, pero también para teléfonos móviles, mediante archivos de voz, para dispositivos sin teclado o con pantalla táctil, etc. La familia de aplicaciones XML ofrece varias ventajas en su lugar:

- EN XML se puede escribir información estructurada independiente de la presentación.
- Los documentos XML se pueden asociar a información de presentación que puede particularizarse para cada medio de presentación (hojas de estilo).
- Los documentos XML se pueden transformar en otros documentos XML o de otro tipo.
- Se pueden expresar operaciones y secuencias de interacción complejas, específicas para comunidades de personas, denominadas vocabularios.

Dos de los problemas más importantes que tiene **el lenguaje HTML** son que la información de formato y estructura está mezclada y que la variedad de maneras aceptables de expresar una marca es elevada, ya que los pequeños errores siguen siendo bien interpretados por la mayoría de los navegadores de Internet.

El primer problema se soluciona con las hojas de estilo en cascada (CSS) que podemos aplicar sin problemas en el lenguaje HTML actual; el segundo, en cambio, requiere un nuevo lenguaje más estricto basado en XML: el XHTML.

Suele ser recomendable dejar toda la información de estilo de un sitio web completo en u documento aparte, compartido por todos los documentos HTML; así un cambio en el documento de estilo afectará inmediatamente a todos los documentos que usen ese estilo. Se les llama hojas de estilo en cascada o superpuestas porque a un mismo documento se le pueden aplicar varios estilos antes de ser presentado, siendo el orden de aplicación el siguiente:

- Declaraciones importantes del autor.
- Declaraciones importantes del lector.
- Declaraciones normales del autor.
- Declaraciones implícitas (atributos HTML) del autor.
- Declaraciones normales del lector.
- Valores por defecto del agente de usuario (navegador).

Estas hojas en cascada crecen y evolucionan con la web, pero a pesar de ello, el lenguaje HTML está limitado desde el comienzo para la representación de páginas web; en éstas y otras muchas situaciones es necesario un lenguaje para representar datos estructurados: el XML.

El lenguaje XML (extensible markup language) es el formato universal para documentos estructurados y datos en la web. Un documento XML puede ser perfectamente presentado en la web con formato tanto de forma como de contenido, y además una persona o un programa podrán automáticamente obtener datos del mismo que pueden ser de nuevo procesados.

Por tanto XML puede servir para **representar y transportar información** estructurada como la que se puede guardar en una base de datos, también sirve para representar información.

El XML es **más restrictivo** que el HTML, pero se permite que diferentes comunidades pueden intercambiar documentos siempre que se pongan de acuerdo en el nombre de los elementos a incluir y la forma de organizarlos; varias

organizaciones han definido su propio vocabulario (nombres de elementos y atributos particulares) y conjuntos de restricciones para construir documentos aceptables dentro de su comunidad.

También XML puede **combinar elementos o atributos** definidos por varias comunidades, así el mecanismo de declaración de espacios de nombres permite asociar un prefijo a cada espacio de nombres o incluso definir el espacio de nombres por defecto para no tener que escribir constantemente el prefijo.

La función de los **esquemas XML** es definir y restringir el contenido y estructura de documentos XML expresado en XML y, por tanto, sustituir los DTD (herencia de SGML) por esquemas XML. Un esquema XML resulta bastante más largo que un DTD, pero a cambio, la verificación de los datos es más rigurosa y la exportación e importación de datos es más sencilla para cualquiera que use un procesador de documentos XML que previamente valide el documento contra un esquema.

XHTML es la formulación de HTML 4.01 en lenguaje XML 1.0; el objetivo es reducir la complejidad que supone procesar documentos HTML y que hace difícil incorporar un navegador a un dispositivo de capacidad reducida. Se trata de "limpiar" HTML, de crear una nueva base para modularizar y extender el lenguaje, de facilitar que nuevos dispositivos puedan "navegar" por la web, pero sin romper con HTML.

XSLT es un lenguaje para expresar hojas de estilo; un documento que describe cómo mostrar un documento XML de cierto tipo. Tiene 3 partes:

- Transformaciones XSL: un lenguaje para transformar documentos XML (representados como una estructura de datos arborescente).
- El lenguaje de caminos (XPath): un lenguaje de expresiones que usa XSLT para acceder a partes de un documento XML o referenciarlas.
- Objetos de formato XSL: Un vocabulario XML para expresar el formato de presentación de un documento, que define objetos y propiedades del formato de un documento y sus componentes.

El lenguaje XSL es muy extenso y sigue evolucionando.

2. ANALIZADORES SINTÁCTICOS DE XML

Un analizador sintáctico de XML es un programa que lee datos XML de una fuente de entrada y realiza el análisis léxico y sintáctico. Si durante el proceso de análisis se producen errores, estos programas deben retornar mensajes que indican el error detectado y el lugar donde se ha producido.

La mayoría de los analizadores sintácticos XML tienen dos modos de funcionamiento:

- Sin validación: Sólo comprueba si el documento está bien formado.
- Con validación: Comprueba tanto si el documento está bien formado como si es válido.

Los analizadores sintácticos XML están pensados como componentes de procesamiento de datos que pueden ser integrados dentro de los programas; no tienen mucho sentido si no se conectan a otros componentes que hagan algo con los datos procesados.

Esto provoca que resulte muy importante disponer de una API (interfaz para programar la aplicación) que permita conectar fácilmente el analizador con el resto del programa.

Existen dos definiciones de API que siguen, o deberían seguir todos los fabricantes de analizadores sintácticos XML: API SAX y API DOM. Estas dos Api

tienen propósitos y orígenes diferentes, pero con ellas se puede realizar cualquier tipo de tratamiento sobre documentos XML.

La API SAX

SAX es el acrónimo de Simple API for XML. SAX es software de libre distribución, totalmente gratuito y con código abierto. SAX lee un documento XML de manera secuencial y a lo largo del proceso de lectura efectúa un análisis sintáctico y genera un conjunto de eventos cada vez que identifica algo significativo.

Con respecto al tratamiento de errores, SAX desencadena eventos que notifican errores cuando identifica durante el proceso de análisis, alguna circunstancia anómala, como un elemento que no tiene etiqueta de cierre o el valor de un atributo que no está entre comillas. Todos los métodos reciben como parámetro la excepción SAXParseException. Este objeto contiene el número de línea en el que se ha detectado el problema, el URI del documento y la información típica de todas las excepciones java (como la descripción del error y el rastro de la pila de llamadas a métodos. SAX genera tres tipos de errores:

1. Warnings: Que constituyen más bien notificaciones de anomalías o situaciones extrañas.
2. Errores no fatales: Que son errores que no impiden que el analizador pueda continuar adelante.
3. Errores fatales: Que provocan que el análisis deba finalizar.

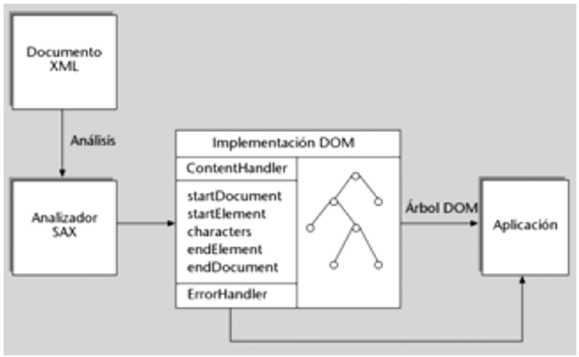
Tanto los warnings como los errores no fatales suelen estar asociados a errores provocados por documentos no válidos, mientras que los errores fatales normalmente son errores provocados por documentos mal formados que no se ajustan a la sintaxis XML.

SAX resulta muy útil para procesos de búsqueda de elementos o de acceso a información específica dentro del documento XML. Es muy rápido y no gasta memoria; además, cuando se localiza el elemento buscado, permite parar el proceso de análisis.

SAX no permite un acceso aleatorio a un documento XML, sino que lee el documento de un modo secuencial y va lanzando acontecimientos mientras dura este proceso de lectura: si una vez leído el documento se quisiera volver a acceder a la información de un elemento determinado, no habría otro remedio que volver a analizar el documento desde el principio hasta encontrar el elemento en cuestión.

La API DOM

Para poder acceder rápidamente y tantas veces como se quiera a la información de los elementos del documento XML, sería necesario construir una representación en memoria, esto es justo lo que hace un analizador DOM. DOM define una estructura de árbol con métodos y propiedades para poder navegar por la estructura y obtener los datos almacenados.



Los analizadores sintácticos basados en DOM retornan un árbol DOM a las aplicaciones que los utilizan. En caso de producirse errores durante el análisis, también informan de ello a la aplicación.

Un analizador sintáctico basado en DOM consta básicamente de dos partes: el analizador propiamente dicho (que se encarga de leer el documento y crear el árbol DOM) y un conjunto de clases de apoyo que implementan la jerarquía DOM (y permiten navegar y acceder a la información almacenada en memoria).

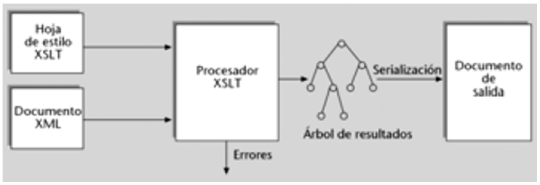
DOM requiere que se cargue el documento entero en memoria y se almacene en una estructura de árbol. Por esta razón, necesita una cantidad de memoria proporcional al tamaño y complejidad del documento XML. También hay que tener en cuenta que alguna de las operaciones sobre estructuras de árbol son caras y conllevan un gasto temporal importante.

En general, el uso de DOM en documentos grandes no resulta muy recomendable (libros o manuales son poco tratables mediante DOM). En Java existe una alternativa a DOM que tiene el propósito de ofrecer una solución de alto rendimiento tanto en el análisis de documentos XML como en la posterior manipulación de la estructura DOM en memoria, la **API JDOM**.

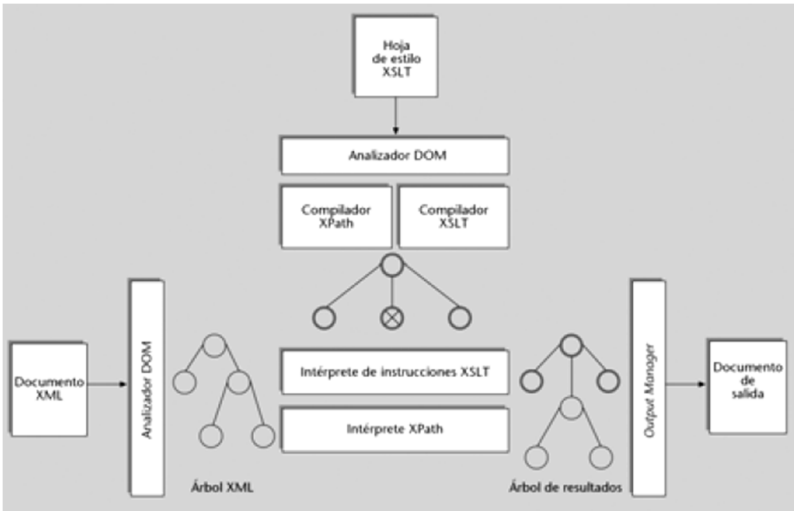
Esta API es una iniciativa de software libre que consiste en poder integrar tan rápidamente como sea posible todo lo que puede mejorar el rendimiento en el análisis y procesamiento de la estructura DOM**.** Los métodos de la jerarquía de clases JDOM son mucho mas intuitivos y fáciles de utilizar en Java que los de DOM y además permite conectarse con otros analizadores y efectuar traducciones de árboles JDOM a árboles DOM.

3. PROCESADORES XSLT

Las reglas XSLT don procesadas por programas denominados procesadores XSLT que tienen como entradas un documento XML y un documento XSLT. Los procesadores XSLT aplican las instrucciones definidas en el documento XSLT sobre un documento XML y generan como salida un árbol de resultados que se puede serializar para convertirlo en un documento de salida.



Un procesador XSLT es un traductor con la particularidad de que se le pueden definir, mediante un lenguaje de transformación (XSLT), los criterios de la traducción. Y esto se puede realizar porque la entrada (el documento XML) no es un lenguaje de programación, sino un lenguaje de marcas. Transformar lenguajes de marcas consiste en filtrar la información, reestructurarla y formatearla. Por otra parte, es asumible crear un lenguaje para este propósito (XSLT) y un ingenio (el procesador XSLT) capaz de procesarlo. En cambio, sería mucho más complejo planteárselo para lenguajes de programación.



La arquitectura interna de un procesador XSLT es una combinación de muchos componentes que ya conocemos. El documento XML y la hoja de estilo XSLT se cargan en memoria utilizando un analizador sintáctico XML basado en DOM. En el caso de la hoja de estilo, los nodos rodeados corresponden a nodos que se copiarán directamente al árbol de resultados y los nodos rodeados y marcados con una cruz corresponden a nodos que contienen instrucciones para aplicar sobre uno o diferentes nodos de la fuente XML.

El funcionamiento resulta muy simple, se va recorriendo cada nodo del árbol DOM de la hoja de estilo:

- Cuando se encuentra un nodo rodeado, se copia directamente en el árbol de resultados.
- Cuando se encuentra un nodo rodeado y marcado con una cruz se procesa de la manera siguiente:
    - Se activa el intérprete Xpath, que se encarga de llevar a cabo la busca solicitada sobre el árbol de la fuente XML.

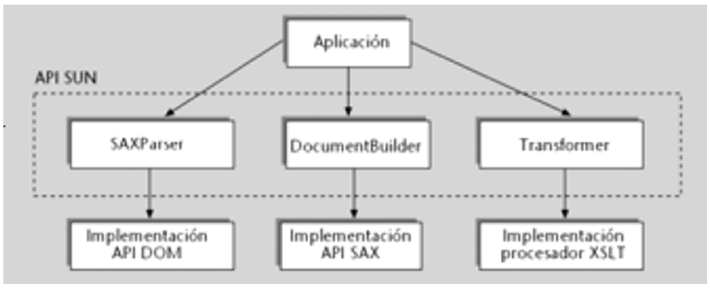
  - Se activa el intérprete de instrucciones XSLT, que aplica las transformaciones sobre los elementos que le proporciona el intérprete Xpath en el paso anterior y copia los nodos resultantes en el árbol de resultados.
- El output manager se encarga de serializar el árbol de resultados en un archivo.

---

4. La API JAXP

En el año 2000 el número de analizadores sintácticos XML y de procesadores XSLT empezaba a ser elevado, muchos habían nacido como iniciativas de investigación dentro de empresas y universidades. A menudo resultaba difícil saber qué funcionalidades y prestaciones ofrecía cada uno.

Sun Microsystems, teniendo en cuenta que estas herramientas estaban todavía en evolución, en lugar de crear sus propias implementaciones y entrar en competencia con las que ya había, creó JAXP (Java Api for XML Processing): una API java que ofrecía una única manera de conectar las aplicaciones Java con cualquier analizador sintáctico XML y procesador XSLT. Sun ofrece a los programadores un conjunto de interfaces, mientras que cada fabricante ofrece las clases que implementan estas interfaces y que encajan sus productos.



JAXP ofrece una capa de conectividad con cualquier analizador XML y procesador XSLT, e independiza las aplicaciones respecto a las herramientas de análisis y tratamiento XML que se utilicen.

TEMA 3 GENERACIÓN AVANZADA

1. PARADIGMAS DE PROGRAMACIÓN

Tema 1
Generación avanzada

La principal diferencia entre compiladores de lenguajes de diferentes paradigmas se encuentra en el tipo de código que se genera: mientras que los imperativos y los orientados al objeto generan código a nivel de ensamblador o inferior, el resto de paradigmas suelen generar código C o C++.

Programación funcional

Los lenguajes funcionales, como Lisp, Mirando o Haskell, se basan en la idea de que un programa es una función con un parámetro de entrada y un resultado. La ejecución del programa se convierte entonces en la aplicación de la función en la entrada para obtener como resultado la salida.

Estos lenguajes se caracterizan porque:

Programa funcional (codificado en Haskell)	Programa imperativo (codificado en C, versión iterativa)

`fact 0 = 1`
`fact n = n * fact (n-1)`
	`int fact (int n) {`
	`  int k = 1;`
	`  while (n > 0) {`
	`  k *= n;`
	`  n--;`
	`  }`
	`  return k;`
	`}`

- Promueven un estilo declarativo en el que los programadores han de especificar sólo qué se debe calcular y dejan al compilador y al sistema de ejecución decidir cómo, dónde y cuándo lo han de realizar.
- Incluyen construcciones de programación para conseguir el nivel de abstracción requerido (por ejemplo, funciones de alto nivel) que complican el compilador y el sistema de ejecución.
- Incluyen facilidades sintácticas gestionadas en el analizador léxico y el sintáctico: regla de fuera de juego, notación de listas, listas exhaustivas y agrupación de patrones.
- Presentan aspectos fundamentales como tipos polimórficos, transparencia referencial, funciones de alto orden o evaluación diferida.
- Se ha verificado que se obtiene una reducción de código de 10 a 1 entre los lenguajes imperativos y los funcionales, lo que reduce mucho el esfuerzo de programación.

Programación lógica

La programación lógica se basa en la especificación de relaciones entre términos de hechos que utilizan estas relaciones y de las reglas para inferir nuevos hechos a partir de los ya existentes. Las reglas y los hechos juntos se denominan cláusulas.

El punto clave de la programación lógica es la posibilidad de inferir hechos específicos preguntando a las cláusulas, combinándolas de distintas maneras, retrocediendo cuando no conducen a ningún sitio y, a menudo, también cuando se llega al lugar deseado.

Los compiladores lógicos habitualmente generan código C. De todos modos, como la expresión más natural de la vuelta atrás utiliza rutinas imbricadas, se suele usar alguna versión de C que las admita, como GNU C. Incluso existen algunos compiladores que aumentan un nivel y generan directamente código Prolog.

Programación paralela y distribuida

Los sistemas paralelos y distribuidos consisten en múltiples procesadores que se pueden intercomunicar. Los lenguajes de programación para estos sistemas soportan construcciones específicas para gestionar la concurrencia y la comunicación. Las dos arquitecturas más significativas son:

- Multiprocesador: En estas estructuras todos los recursos acceden, como mínimo, a un espacio de memoria compartido y se suelen comunicar leyendo y escribiendo datos en este espacio común. Se pueden construir, por ejemplo, conectando múltiples procesadores a un único bus (existen otras tipologías mucho más sofisticadas).

- Multiordenador: Estas estructuras consisten en diferentes ordenadores conectados en red que ejecutan procesos en distintos espacios de direcciones. Los procesos se comunican enviando mensajes por la red.

También resulta interesante estudiar cómo se expresa el paralelismo. Existen lenguajes con paralelismo implícito, en los que el compilador intenta paralelizar el programa de manera automática generando el código necesario para mantener las comunicaciones. Estos compiladores se suelen denominar compiladores heroicos porque resultan muy complejos de construir. En los lenguajes con paralelismo explícito, el programador ha de expresar el paralelismo de manera explícita utilizando sentencias que coordinen las actividades paralelas. Encontramos varios modelos de programación paralela:

- El modelo más sencillo de programación paralela consiste en una colección de procesos que se comunican utilizando variables compartidas. Hay, como mínimo, una parte de la memoria que es compartida por los distintos procesos, de manera que todos pueden acceder a las mismas variables, comparten información y se comunican a través de éstas. Puede existir sincronización por exclusión mutua (un proceso sólo puede acceder a una determinada variable en un momento dado) o un monitor (el cual contiene los datos y las operaciones necesarias para acceder a ellos, pero todas las operaciones dentro del monitor son mutuamente exclusivas).
- El modelo de paso de mensajes sirve también para multiordenadores. Cada proceso accede sólo a su propio espacio de datos locales. EL intercambio de información se lleva a cabo pasándose mensajes de uno a otro mediante la utilización de dos primitivas básicas: enviar y recibir. Este paso de mensajes puede ser asíncrono o síncrono, y de recepción explícita o implícita.
- Objetos: Se encapsulan los datos en objetos y son accesibles mediante operaciones definidas dentro de los objetos. Esto aumenta la estructuración de los programas y facilita la reutilización de los componentes. El paralelismo se introduce en estos lenguajes, lo cual permite que diferentes objetos se ejecuten al mismo tiempo y deja que un proceso se ejecute dentro del objeto.
- Los lenguajes con datos paralelos, a diferencia de las tareas paralelas, todos los procesadores ejecutan el mismo algoritmo, pero operan en diferentes partes del conjunto de datos. Estos lenguajes normalmente permiten que el programador decida qué se puede ejecutar en paralelo y dejan que el compilador distribuya automáticamente los cálculos y los datos entre los distintos procesadores.

2. GESTIÓN DEL CONTEXTO

El análisis léxico y el sintáctico tienen relación con fenómenos locales y sólo relacionan ítems con sus vecinos. La gestión del contexto se lleva a cabo durante el análisis semántico, después de los análisis léxico y sintáctico. La gestión del contexto hace referencia a relaciones más lejanas: relaciona el tipo de una variable definido en una sentencia de declaración con el uso de la variable en una expresión o relaciona la posición de una etiqueta (el punto de su definición) con su uso en una sentencia label.

Estas relaciones se alcanzan a través de los identificadores: todas las apariciones de un identificador id se conectan con la declaración de id (mediante su ubicación en la tabla de símbolos), de donde se extrae información de su tipo, del tiempo de vida, etc.

Hay dos hechos que tienen un papel muy directo en la verificación estática de la mayoría de estos lenguajes: la **identificación** y la **verificación**.

La **identificación** es el proceso de busca de la primera vez (y generalmente única) que aparece en el código fuente la definición de un identificador o de un operador: el punto de definición. En esta sentencia de definición se proporciona información del identificador: la clase (constante, variable, módulo…) el tipo, un posible valor inicial, unas posibles propiedades de asignación

de memoria, etc. Todas las demás apariciones del identificador se denominan puntos de uso o sentencias de uso porque utilizan esta información.

Hay dos excepciones: determinados lenguajes permiten sentencias en avanzada con lo que los identificadores pueden tener más de un punto de definición; y hay lenguajes que no requieren que los identificadores sean declarados en ningún sitio: el nombre del identificador contiene la información o bien ésta se encuentra distribuida entre las diferentes apariciones del símbolo.

Algunos espacios de nombres, especialmente los de los lenguajes estructurados en bloques se organizan según los **ámbitos**. De las distintas organizaciones de la tabla de símbolos, la más utilizada es la de dispersión (hash) pero para acelerar su funcionamiento, especialmente cuando se elimina un ámbito, se puede mejorar esta estructura incorporando una pila de ámbitos. Cada posición de esta pila contendrá una lista encadenada con todos los símbolos del ámbito (cada registro de la lista estará compuesto de dos apuntadores: uno apuntará a su posición de la tabla de dispersión que contiene el símbolo, y el otro al siguiente elemento de la lista).

En un lenguaje que permite **sobrecarga**, un identificador no oculta las definiciones anteriores con el mismo nombre pero con características diferentes. Algunos lenguajes permiten, también, sobrecargar los operadores. En estos tipos de lenguajes, el proceso de identificación no trabajará con una única definición de un nombre, sino con un conjunto de definiciones. La ambigüedad que provoca la sobrecarga se resuelve a partir del contexto. Otros lenguajes permiten construcciones que realizan la copia de ámbitos ámbitos importados.

Acabamos de ver como la identificación de un símbolo proporciona un apuntador a la tabla de símbolos que contiene sus propiedades. El primer paso en la verificación de tipos consiste en determinar la clase del símbolo: variable, constante, función, tipo, módulo, etc. Esta verificación de la clase del símbolo es una tarea sencilla, ya que solo se ha de utilizar la información de la tabla de símbolos para verificar si el símbolo pertenece al contexto adecuado.

Puede haber declaración explícita o implícita, en este último caso, el tipo no tiene un nombre en el programa fuente y el compilador lo deberá inventar y crear una representación interna. En las declaraciones de tipo algunos lenguajes permiten efectuar referencias a identificadores que todavía no han sido declarados, lo que permite definir tipos mutuamente recursivos. Constituyen las denominadas **referencias a la avanzada**. Estas referencias complican al compilador porque:

- Las referencias a la avanzada se han de resolver. Cuando se detecta una, se añade a la tabla de símbolos y se marca como referencia a la avanzada. Cuando, más adelante, se encuentre la declaración del tipo para esta referencia, se debe modificar la posición en la tabla de símbolos para representar el tipo correcto.
- Se ha de añadir una verificación para finales perdidos. Al final de un ámbito en el que ha habido alguna referencia a la avanzada, se debe verificar que se hayan resuelto todas las referencias a la avanzada.
- Se ha de añadir una verificación de circularidad. Una consecuencia de las referencias a la avanzada es que el programador puede efectuar definiciones de tipo circular, lo que probablemente será ilegal.

Como ya sabemos, todos los tipos de datos utilizados en un programa se guardan en la **tabla de tipos**, que debe contener toda la información necesaria para llevar a cabo la verificación de tipos y la generación de código. La información que se guarda en la tabla de símbolos depende del lenguaje que se compila; en cambio, la estructura utilizada en el compilador depende del lenguaje de implementación, el lenguaje en el que está escrito el compilador: si está escrito en lenguaje imperativo se utiliza un registro con una parte variable (o una unión) para los campos que dependen del tipo de constructor; si el lenguaje es orientado al objeto, se podría utilizar una clase que contuviera los campos y métodos que

comparten todos los constructores de tipos y, para cada tipo de constructor individual, se utilizaría su propia subclase con sus extensiones particulares.

Realizar una **equivalencia de tipos** consiste en verificar los tipos que intervienen en una expresión o comparar los tipos esperados en los parámetros de una rutina con los que realmente se reciben. Cuando dos tipos tienen la misma representación y se pueden utilizar uno en el lugar de otro, se dice que son equivalentes. Hay dos clases de equivalencia de tipos:

- Equivalencia de nombres: Es la utilizada en los lenguajes imperativos y orientados al objeto modernos. Dos tipos son equivalentes si tienen el mismo nombre.
- Equivalencia estructural: Dos tipos son estructuralmente equivalentes cuando sus variables pueden tener los mismos valores y permitir las mismas operaciones.

Cuando en una posición del programa se encuentra un tipo T1 y se esperaba un tipo T2, se ha de verificar si T1 y T2 son de tipos equivalentes. Si no lo son, se deben consultar las reglas del lenguaje para decidir qué se ha de hacer. Según el lenguaje, el compilador dará un error al detectar esta situación o bien insertará una instrucción de conversión de entero a real en el código generado. Esta conversión implícita de datos y tipo se denomina **coerción**.

3. REPRESENTACIÓN Y GESTIÓN DE LOS DATOS DEL LENGUAJE FUENTE

Cada tipo de datos del lenguaje fuente posee una representación equivalente en el lenguaje objeto y el programador del compilador ha de crear la función de mapeo de los tipos de datos del lenguaje fuente a lenguaje objeto:

- **Tipo básico**: Podemos encontrar los caracteres (se suelen mapear a bytes individuales del lenguaje objeto), los números enteros (de distintas longitudes y números en coma flotante. Además de las comparaciones se suelen realizar operaciones aritméticas básicas, teniendo una traducción directa al lenguaje objeto) y void (en algunos lenguajes es un tipo explícito que no se corresponde con ningún valor real; en otros, se incluye sólo explícitamente).
- **Tipos enumerados**: Definen un conjunto de nombres para ser los valores. Su representación en tiempo de ejecución será un número entero, normalmente con valores que van del cero al número de valores enumerados del tipo menos 1. Como en tiempo de compilación se conocen los valores del tipo enumerado, se puede elegir un tipo entero adecuado para representarlos. Un tipo enumerado que suelen incorporar muchos lenguajes es el tipo booleano con los valores enumerados true (1) y false (valor 0).
- **Tipos apuntadores**: La mayoría de los lenguajes imperativos y orientados al objeto soportan algún tipo apuntador. Su representación en tiempo de ejecución es un entero sin signo, con longitud suficiente para representar la dirección y, por lo tanto se pueden aplicar todas las operaciones de números enteros. Eso sí, tienen una operación particular, la desreferenciación, que consiste en obtener el valor guardado en la estructura de datos indicada por el apuntador.
    - **Apuntadores colgados**: Hay algunas situaciones en las que el apuntador no hace referencia a un valor correcto, con lo que resulta peligroso el uso de apuntadores. En algunos lenguajes como C, las acciones ante un programa incorrecto no se encuentran definidas y lo mejor que se puede esperar es que el programa finalice erróneamente andes de dar resultados correctos. Otros lenguajes, como Java obvian este problema al no permitir el uso de apuntadores.
- **Tipo Registro**: Un registro o estructura es un elemento en el que convive agrupado un conjunto fijo de campos miembros. En la representación en lenguaje objeto, los miembros se guardan uno tras otro en posiciones

consecutivas de memoria. Las 3 operaciones básicas sobre los registros son: la selección de un campo de un registro (en el lenguaje objeto esta selección se consigue mediante posición+desplazamiento), la copia de un registro en otro y la comparación de registros.

- **Tipo array**: Define unas estructuras de datos que consisten en series de elementos del mismo tipo. Puede presentar una dimensión (vector), dos (matriz) o más. Se accede a cada elemento del array utilizando un índice por cada dimensión del array.
- **Tipo interfaz**: Es como una clase que sólo consiste en una serie de especificaciones de métodos. Una interfaz se puede heredar extendiendo una interfaz padre. No se instancian como objetos, pero se pueden declarar variables e invocar los métodos que se especifican (las variables son apuntadores Java a objetos). La principal ventaja de las interfaces radica en el hecho de que una clase objeto puede implementar una o más de estas interfaces y que un tipo interfaz es compatible con cualquier objeto que lo implemente.

Hay muchas características propias de los lenguajes orientados al objeto y distintas maneras de implementarlas. Podemos empezar con la **herencia**, que permite al programador basar una clase B en una clase A, de manera que B además de sus propios métodos y atributos, herede todos los de A. Esta característica también se denomina **extensión de tipo**: la clase B extiende la clase A con uno o más métodos y atributos. La clase A es la **superclase** (o padre) de B y la clase B es una subclase o hija de A.

Cuando una clase extiende una clase A, puede redefinir alguno de los métodos de A; es decir, cuando una superclase A define un método y todas sus subclases lo heredan, lo pueden redefinir, reescribiéndolo, de modo que sus implementaciones del método serán diferentes de las de la superclase.

Otra característica es el **polimorfismo**. Cuando una clase B extiende una clase A y el lenguaje permite que un apuntador a la clase B sea asignado a un apuntador de la clase A se dice que el lenguaje soporta el polimorfismo. Así, un apuntador de A puede apuntar a objetos de la clase A o a objetos de sus subclases.

La **herencia múltiple** se produce cuando se permite que los objetos puedan heredar de más de una clase. En estas situaciones no es posible representar un objeto como un apuntador a la tabla de ejecución seguido de los atributos del objeto

4. RUTINAS Y MÉTODOS

Cuando se llama a una rutina, generalmente se han de llevar a cabo las tareas siguientes:

- Crear un nuevo entorno en memoria temporal que contenga, como mínimo, las variables locales de la rutina.
- Pasar información al nuevo entorno: los parámetros.
- Transferir el flujo de control al nuevo entorno, con –en principio- un punto de retorno asegurado.
- Al acabar la ejecución de la rutina, se deberá retornar información desde el nuevo entorno: el o los valores de retorno.

Estas cuatro tareas muestran que el ingrediente esencial de la activación de una rutina lo constituye en nuevo entorno. La estructura de datos utilizada para guardar los datos pertinentes en la invocación de una rutina o de un método de un objeto se denomina **registro de activación** o marco.

En la mayoría de los lenguajes, las rutinas se activan en un orden LIFO, en el que la primera en entrar es la última en salir: cuando la rutina A llama a la rutina B, A no podrá continuar su ejecución hasta que B no acabe. Este comportamiento se suele modelizar utilizando una pila para guardar los registros de activación de las

rutinas a medida que se van creando, aunque, como se verá, en algunos casos la pila puede resultar insuficiente y es necesario utilizar la memoria montículo. Esta estructura de pila permite una gestión implícita de la activación de las rutinas y facilita mucho la programación del compilador.

Cuando una **rutina** es llamada se crea un registro de activación para guardar los datos y se transfiere el flujo de controla su primera instrucción de código. En este momento se dice que la rutina se ejecuta.

Las rutinas que se ejecutan o que se encuentran suspendidas, tienen un registro de activación creado y están activas. En caso de que una rutina esté activa más de una vez simultáneamente, será una rutina recursiva, pero en un momento dado, sólo puede estar en ejecución una única invocación.

Cuando una rutina se acaba, se elimina su registro de activación y cuando la última activación de la rutina se ha acabado y se han eliminado todos los registros de activación de la rutina, ésta se convierte en inactiva.

Sobre una rutina se puede realizar una **llamada**, que crea un registro de activación y transfiere el flujo de control del programa al código de la rutina.

Como se ha visto, una parte de la secuencia de acciones que se llevan a cabo en la llamada de una rutina se realiza en la parte de la rutina invocadora y otra, en la parte de la rutina invocada. La reserva de espacio para el registro de activación será la primera acción que se llevará a cabo y se puede realizar utilizando una pila o una memoria montículo. Si se utiliza una pila, la reserva es más o menos automática y los componentes se ponen en la pila en el orden correcto. Si se utiliza la memoria montículo, se ha de reservar un trozo secuencial de memoria de una longitud suficiente para guardar todo el registro de activación. Como la situación normal es la primera, asumiremos que se utilizada una pila.

En una pila se pueden llevar a cabo las dos acciones siguientes:

- Push: Poner un elemento en la cima.
- Pop: Sacar el último elemento entrado en la cima.

Existen distintos mecanismos posibles para realizar el **paso de parámetros**: el más sencillo es el paso por valor: el valor actual del parámetro se utiliza como valor inicial del parámetro formal correspondiente. Después de retornar, la rutina que ha realizado la llamada no ve ningún cambio en el valor del parámetro. Muchos lenguajes soportan una forma de parámetros de salida que permite a una rutina que las modificaciones en los valores de los parámetros actuales sean reflejados en los correspondientes parámetros formales:

- Paso por referencia: Se copia la dirección del parámetro actual en el registro de activación.
- Paso por valor-resultado: Se copia la dirección del parámetro actual en el registro de activación. Una variable local sustituye al parámetro en las sentencias del código de la rutina. Esta variable se inicializa con el valor del parámetro. El cambio de valor se efectúa a la vuelta: cuando finaliza la rutina, el valor de la variable local se copia en el parámetro original.

Área de variables locales

Una vez transferido el control, la rutina invocada crea la parte de variables locales donde residirán las variables locales y temporales del compilador. Normalmente, el compilador determina su longitud considerando el área como un registro en el que las variables locales de la rutina forman los campos. También ha de tener en cuenta los requerimientos de longitud y alineación de la memoria.

Tal como ocurría con los parámetros, algunos datos pueden tener componentes dinámicos; en este caso, se deberán guardar en la memoria montículo o en el área de asignación dinámica del registro de activación, como se explicará a continuación.

Pila de trabajo

Esta pila se utiliza para guardar resultados anónimos de expresiones. También se pueden guardar límites de sentencias for (aunque es más normal guardarlas en variables temporales). Analizando el código de la rutina, el compilador puede decidir fácilmente la longitud máxima de la pila e incorporarla como un bloque fijo detrás del área de las variables locales.

Área de memoria dinámica

Si el lenguaje objeto permite registros de activación que puedan modificar su longitud dinámicamente, se puede incorporar una parte separada de memoria dinámica para las variables locales y para los parámetros con componentes dinámicos. Como ésta es la única parte del registro de activación de la que no se puede saber la longitud en tiempo de compilación, se debe situar al final del registro.

Las principales ventajas que supone utilizar áreas de memoria dinámica son que no será necesario eliminar las variables cuando la rutina finalice, y que se mejora la velocidad, ya que la gestión se efectúa utilizando el registro SP. El inconveniente es que el área de pila de trabajo ya no podrá utilizar el SP, sino que se deberá simular el funcionamiento por software.

Retorno de valores

Si la rutina invocada tiene que retornar un valor, se pueden seguir diferentes esquemas para que se percate de ello la rutina invocadora:

- Si el valor es un tipo de datos simple, se guarda el resultado en el registro de resultado de función.
- Si el valor no cabe de manera natural en el registro, la situación se complica y será necesario aplicar una de las 3 soluciones siguientes:
    - Si el compilador conoce la longitud del resultado, puede reservar espacio en el área de variables temporales de la rutina invocadora y pasar su dirección como parámetro extra a la rutina invocada.
    - La rutina invocada puede reservar espacio dinámicamente en el registro de resultado de función y retornar un apuntador a esta dirección.
    - El resultado de puede dejar en la memoria dinámica del registro de activación de la rutina invocada o en una variable local, y poner un apuntador a ella en el registro de resultado de función.

La secuencia de llamada y retorno

La secuencia de pasos que se llevan a cabo al invocar una rutina es la siguiente:

- Crear un registro de activación.
- Evaluar los parámetros y guardarlos en el registro de activación.
- Llenar el área de administración del registro de activación: apuntador marco de la rutina invocadora (enlace dinámico), dirección de retorno, apuntador léxico y quizá algunos registros de la máquina.
- Transferir el control a la rutina invocada.
- Hacer que el apuntador de marco apunte al nuevo registro de activación de la rutina invocada.
- Actualizar el apuntador de pila reservando espacio suficiente para el área de variables locales.
- Si hay parte de memoria dinámica, se llena a medida que se va requiriendo.

La secuencia de pasos necesaria para el retorno es la siguiente:

- Si la rutina invocada retorna un valor, lo debe guardar en el área designada para esto.
- Si se han guardado los registros de la máquina en el área de administración, hay que restaurarlos tal como fueron guardados.

- Actualizar el apuntador de pila a fin de que apunte al apuntador de marco.
- Restaurar el apuntador de marco desde el área de administración.
- Transferir el control a la rutina invocadora utilizando la dirección de retorno.
- Eliminar el registro de activación de la rutina invocada.

5. GESTIÓN DE ERRORES EN TIEMPO DE EJECUCIÓN

Existe una inmensa variedad de problemas que se pueden presentar en el programa: desbordamiento de números enteros, división por cero… El código generado, el sistema de ejecución, el sistema operativo o el hardware, según las circunstancias, pueden detectar el error y, probablemente, algún nivel del sistema dará un mensaje de error y el programa finalizará.

La gestión de los errores en tiempo de ejecución supone dos acciones: detección y gestión.

Respecto a la **detección** decir que un error puede estar causado por motivos externos o bien por el mismo programa. Para detectar errores externos dependemos mucho del sistema, así un error de memoria puede o no ser transmitido al programa. Con respecto a los internos, los provocados por el mismo programa, en algunos casos el sistema operativo puede provocar una interrupción al detectarlos, pero el compilador también puede generar código extra después de las operaciones que pueden ser conflictivas para detectarlos. Como estas verificaciones suponen incrementar mucho el tiempo de proceso, la mayoría de los compiladores permiten desactivarlas en el proceso de generación del código objeto.

La **gestión** de errores en tiempos de ejecución más comunes son las señales y las excepciones. Debemos tener en cuenta que si el lenguaje no permite tratar los errores en tiempo de ejecución, el sistema debe generar un mensaje inteligible y finalizar el programa creando una copia de la memoria que pueda ser analizada a posteriori con un depurador.

Las **señales** se invocan cuando se produce un error en tiempo de ejecución, encargándose, por ejemplo, de cerrar ficheros, liberar recursos, mostrar un mensaje de error y cerrar el programa; también se puede tratar el error y continuar la ejecución después justo de la instrucción que lo ha provocado.

Las **excepciones** son un método más flexible que las señales y que se suele encontrar en muchos lenguajes de programación, consiste en utilizar gestores de excepciones, que especifican un conjunto de instrucciones que se deberían ejecutar si la condición de error que gestiona se produce.

Normalmente cada bloque o rutina posee su propio gestor de excepciones, Incluso, se pueden tener distintos gestores de excepciones para diferentes condiciones de error.