Tema 1 Introducción

1.1 Qué es un lenguaje de programación.

El lenguaje ensamblador, con su dependencia respecto a cada máquina, su bajo nivel de abstracción y su dificultad para escribirse y comprenderse, no es lo que usualmente pensamos como un lenguaje de programación.

Los programadores se dieron cuenta de que un nivel más elevado de abstracción mejoraría su capacidad de escribir instrucciones concisas y comprensibles que podrían ser utilizadas con pocos cambios de una máquina a otra. Utilizan construcción estándar como la asignación, los bucles, selecciones y opciones.

Los programas se hicieron relativamente independientes de las máquinas.

Legibilidad por parte del ser humano.

Esta es una idea menos precisa y menos comprendida. Requiere que un lenguaje de programación proporciones abstracciones de las acciones de los computadores fáciles de comprender para las personas, consecuencia por lo que los lenguajes de programación tienden a parecerse a los lenguajes naturales.

1.2 Abstracciones en los lenguajes de programación.

Los lenguajes de programación no necesitan basarse en ningún modelo de computadora.

Una evolución paralela también ha evitado que el uso de los lenguajes de programación sirva únicamente para que los seres humanos se comuniquen con las computadoras; ahora estos lenguajes, permiten que los seres humanos se comuniquen entre sí.

Un lenguaje de programación es un sistema notacional para describir computaciones en una forma legible tanto para la máquina como para el ser humano.

Computación.

- Cualquier proceso que puede ser ejecutado por una computadora.
- A veces se diseñará un lenguaje de programación teniendo en mente un tipo específico de procesamiento (lenguajes de propósito especial), nos concentraremos en lenguajes de propósito general.

Legibilidad por parte de la máquina.

- Debe tener una estructura lo suficientemente simple par que permita una traducción eficiente.
- Deberá existir un algoritmo para la traducción del lenguaje.
- El algoritmo no debe tener la complejidad demasiado elevada.
- Las abstracciones de los lenguajes de programación se agrupan en dos clases generales de datos y de control.

1.2.1 Abstraccions de datos.

Abstracciones básicas.

Resumen la representación interna de valores de datos comunes en una computadora. Ej. los valores enteros de datos se almacenan en complemento a 2, proporcionando operaciones estándar como la adición y multiplicación.

Las localizaciones en memoria que contienen valores de datos se abstraen dándoles un nombre y se conocen como variables.

El tipo de valor de datos también recibe un nombre y se conoce como tipo.

Abstracciones estructuradas.

La estructura de datos es el método principal para la abstracción de colecciones de valores.

Ej. un registro de empleado puede incluir nombre, dirección.
Una estructura de datos típica es el arreglo, que reúne una secuencia de elementos. int a[10]

Abstracciones unitarias.
Útil y necesario reunir códigos, relacionados entre sí, en localizaciones específicas dentro del programa, en forma de archivos por separado o estructuras por separado dentro de un archivo.
Encapsulado de datos y ocultación de la información, estos mecanismos pueden variar de un lenguaje a otro pero están relacionados con los tipos abstractos de datos.
Ej. Módulo en ML y Haskell y el paquete en Ada y Java.
Mecanismo de las clases orientadas a objetos.
Las clases ofrecen encapsulamiento de datos.
Importancia por su reutilización, capacidad de reutilizar la abstracción de datos en programas diferentes.
Componentes (piezas de programa) o contenedores (estructuras de datos que contienen otros definidos por el usuario.
Las abstracciones de datos unitarios se convierten en la base de mecanismos de bibliotecas.

1.2.2 Abstracciones de control.

Abstracciones básicas.
Sentencias en un lenguaje que combinan unas cuantas instrucciones de máquina en una sentencia abstracta más comprensible. Ej. enunciado de asignación, resume cómputo y almacenamiento de un valor en la localización de una variable. x = x + 1

Abstracciones estructuradas.
Dividen un programa en grupos de instrucciones que están anidadas dentro de pruebas que gobiernan su ejecución:

Ej if, case de Pascal, switch de C
Una ventaja de las estructuras de control es que se pueden anidar.
Los mecanismos de bucle o ciclos estructurados:
While, for y do de C y C++.

Un mecanismo adicional, útil para estructurar el control es el procedimiento. Esto le permite al programador considerar una secuencia de acciones como si fuera una sola acción.
Primero debe definirse el procedimiento dándole un nombre y asociándolo con las acciones que se van a llevar a cabo (declaración de procedimiento), y después el procedimiento debe ser llamado en el punto que las acciones deben ejecutarse (invocación o activación del procedimiento).
Una llamada de procedimiento requiere el almacenamiento de información sobre el estado del programa en el punto de llamada (entorno de ejecución).
Una función puede considerarse como un procedimiento que devuelve un valor o un resultado a su invocador.
En algunos lenguajes los procedimientos se consideran como casos especiales de las funciones que devuelven un valor, C, C++ y Java se les llama funciones nulas (void).

Abstracciones unitarias.
Tienen la finalidad de incluir una colección de procedimientos que proporcionan servicios relacionados lógicamente con otras partes del programa y que forman una parte unitaria.
Los procedimientos que producen estas operaciones pueden reunirse en una unidad de programa que puede traducirse por separado.
Varios lenguajes de programación incluyen mecanismos que permiten la ejecución en paralelo de alguna parte de los programas, proporcionan sincronización y comunicación entre dichas partes. Java contiene mecanismos para declaración de hilos o hebras.

1.3. Paradigmas de computación.

Un lenguaje caracterizado por la ejecución secuencial de instrucciones, el uso de variables en representación de localizaciones de memoria y el uso de la asignación para cambiar el valor de las variables, se conoce como lenguaje imperativo. (procedurales)
La mayoría de los lenguajes de programación son imperativos. Se convierten en un paradigma o patrón, para que los sigan los lenguajes de programación.
Paradigma funcional, se basa en la noción abstracta de una función.
Paradigma lógico, se basa en la lógica simbólica.
Una paradigma que ha tomado una enorme importancia en la última década es la orientación a objetos, permiten escribir código reutilizable y ampliable, que opera imitando al mundo real, siendo una extensión del lenguaje imperativo.

Programación orientada a objetos.
Se basa en la idea de un objeto que puede describirse como una colección de localizaciones de memoria, junto con todas las operaciones que puedan cambiar los valores de dichas localizaciones de memoria.
El ejemplo estándar de un objeto es una variable. En muchos lenguajes orientados a objetos éstos se agrupan en clases. JAVA.

Programación funcional.
Basa la descripción de las computaciones en la evaluación de funciones o en la aplicación de funciones a valores conocidos. (lenguajes aplicativos).
Tiene como mecanismo básico la evaluación de una función o llamada de función.
No involucra la idea de variable o asignación de variables.
Las operaciones repetitivas no se expresan mediante ciclos, sino mediante funciones recursivas.
Ventajas: el lenguaje se hace más independiente del modelo de máquina, los programas funcionales se parecen a las matemáticas, resulta más fácil extraer conclusiones precisas.
Ej. Ada, Haskell y sobre todo LISP.

Programación lógica.
Se basa en la lógica simbólica. Conjunto de enunciados que describen lo que es verdad con respecto a un resultado deseado.
Un lenguaje de programación lógico puro no tiene necesidad de abstracciones de control como ciclos o selección, todo lo que necesita es el enunciado de las propiedad del cómputo, por ello se le conoce también como programación declarativa.
Lenguajes de muy alto nivel. Prolog
En Prolog la forma de un programa es una secuencia de enunciados, cláusulas.

1.4. Definición de lenguaje.

La mejor forma de lograr, la necesidad de independencia de la máquina o independencia de la implementación, es a través de la estandarización. ANSI, ISO: han publicado definiciones para muchos lenguajes.
Los requisitos de una definición formal aportan disciplina durante el diseño de un lenguaje.
La definición del lenguaje se puede dividir aproximadamente en dos partes: sintaxis (escritura) y semántica (significado):

Sintaxis.
Es la descripción de las maneras en que las diferentes partes del lenguaje pueden ser combinadas para formar otras partes.
La sintaxis de casi todos los lenguajes está dada utilizando gramáticas libres de contexto.
Un problema íntimamente relacionado con la sintaxis de un lenguaje de programación es su estructura léxica. (estructura de palabras del lenguaje: tokens)

Semántica del lenguaje.
La semántica o significado de un lenguaje, es mucho más compleja y difícil de describir con precisión.
El significado de un mecanismo en particular pudiera involucrar interacciones con otros mecanismos en el lenguaje, por lo que una descripción completa de su significado en todos los contextos puede llegar a ser extremadamente compleja.
Se han desarrollado varios sistemas de notación para definiciones formales, incluyendo la semántica operacional, semántica denotacional y semántica axiomática.

1.5 Traducción del lenguaje.

Para que un lenguaje sea útil debe tener un traductor.
Un traductor que ejecuta un programa directamente se conoce como intérprete y un traductor que produce un programa equivalente en una forma adecuada para su ejecución se conoce como compilador.
La interpretación es un proceso que consta de un paso, en donde tanto el programa como la entrada le son dados al intérprete y se obtiene una salida.
Un intérprete se puede considerar como un simulador para una máquina cuyo lenguaje máquina es el lenguaje que se está traduciendo.
La compilación consta de dos pasos: el programa original (fuente) es la entrada del compilador, y la salida del compilador es un nuevo programa (objetivo). Este programa objetivo puede ser entonces ejecutado, si está en una forma adecuada para una ejecución directa (lenguaje máquina), lo más común ensamblador.
Es posible tener traductores intermedios entre intérpretes y compiladores.
Tanto compiladores, como intérpretes deben llevar a cabo operaciones similares al traducir un programa fuente. Analizador léxico y analizador sintáctico o gramatical, y finalmente un analizador semántico.
Un traductor también tiene que mantener un entorno de ejecución, en el cual se asigna el espacio adecuado de memoria para datos del programa y registra el avance de su ejecución.
Las propiedades de un lenguaje de programación pueden ser determinadas antes de la ejecución son propiedades estáticas, mientras que las que solo puede determinar durante la ejecución se conocen como propiedades dinámicas.

1.6. Diseño del lenguaje.

La legibilidad de máquina y del ser humano son los requisitos que prevalecen en el diseño.
El reto del diseño del lenguaje de programación, es lograr la potencia, expresividad y comprensión que requiere la legibilidad del ser humano, mientras que se conservan al mismo tiempo la precisión y simplicidad necesarias para la traducción de máquina.
En lenguaje exitoso de programación tiene utilerías para (abstracción de datos) y (abstracción de control).
La meta prevaleciente de la abstracción en el diseño de lenguajes de programación es el control de la complejidad.
Los errores se pueden clasificar de acuerdo con la etapa de traducción en la que ocurren.
Errores ortográficos como “whillle” en vez de “while”, los detectara el analizador sintáctico.
Los errores semánticos pueden ser estáticos (tipos incompatibles, variables no declaradas).
Errores lógicos: son aquellos que comete el programador, hacen que el programa se comporte de una manera errónea.

Tema 3 Principios de diseño

La legibilidad del ser humano, así como el mecanismo de control de la abstracción de la complejidad son requisitos clave del lenguaje de programación moderno.

El diseño del lenguaje de programación depende del uso pretendido y de los requisitos de este uso.

Historia y criterios de diseño:

Al iniciarse los lenguajes de programación, el criterio principal de diseño era la ***eficiencia en la ejecución***.

La razón principal de la existencia de un lenguaje de alto nivel era facilitar la escritura del programa.

Capacidad de escritura: cualidad de un lenguaje que le permite a un programador su uso para expresar una computación con claridad, corrección, de manera concisa y con rapidez.

Esta capacidad de escritura quedó siempre en segundo plano en relación con la eficiencia.

Legibilidad: cualidad de un lenguaje que le permite a un programador comprender y entender de manera fácil y con precisión la naturaleza de una computación.

Para controlar la complejidad de un lenguaje era necesario utilizar mecanismos de abstracción y disminuir el número de reglas y restricciones que los programadores tenían que aprender.

Las ***metas de diseño*** de mayor éxito han sido: la correspondencia de los mecanismos de abstracción con las necesidades de las tareas de programación, el uso de bibliotecas para ampliar la resolución de tareas específicas y el uso de técnicas orientadas a objetos para aumentar la flexibilidad y reutilización del código.

Eficiencia:

Pueden darse varios tipos de eficiencia:

- ***Eficiencia del código***. El diseño del lenguaje debe permitir generar código eficiente (***Optimizabilidad***).
- ***Eficiencia de la traducción***. Traducción rápida y usando un traductor de tamaño reducido.
- La verificación de errores puede ser un problema grave de eficiencia, ya que verificar la existencia de un error en tiempo de traducción puede hacer ineficiente al traductor, y la generación de código para verificar el error durante la ejecución podría hacer que el código objetivo resulte ineficiente.
- ***Eficiencia de la programación***. Rapidez a la hora de escribir programas con dicho lenguaje.
- ***Capacidad de implementación***. Eficiencia con la que se puede escribir un traductor.
- ***Legibilidad***. Capacidad de un lenguaje para ser comprendido.
- ***Confiabilidad***. Asegurar que no hay comportamientos extraños.
- ***Eficiencia de ejecución***.

Los ingenieros del software estiman que ocupan mucho mas tiempo en eliminar errores y en el mantenimiento que en la codificación original, por lo que la ***legibilidad*** y la ***capacidad de mantenimiento*** pueden ser en último término uno de los mayores problemas de eficiencia.

La eficiencia con la cual se puede crear software depende de la ***legibilidad*** y de la ***capacidad de darle mantenimiento*** en tanto que tiene menos importancia la ***facilidad de escritura***.

Regularidad:

Expresa lo bien que está integrado el código.

La regularidad se subdivide en tres conceptos:

- ***Generalidad***: Elimina casos especiales y agrupa constructores similares.
- ***Ortogonalidad***: Los constructores deben comportarse de la misma forma en todos los contextos.
- ***Uniformidad***: Cosas similares no parecen ser o se comportan de manera similar y cosas distintas se comportan de manera similar cuando no deberían.

Al juzgar si una no Regularidad es razonable, uno debe relacionarla con las metas del diseño y las complicaciones que pudieran ocurrir en caso de que se eliminen. Si una no regularidad no puede justificarse claramente, entonces es un error de diseño.

Principios adicionales:

- ***Simplicidad***.
- ***Expresividad***. Facilidad de un lenguaje para expresar procesos y estructuras complejas.
- ***Extensibilidad***. Mecanismos para añadir nuevas estructuras o características al lenguaje.
- ***Capacidad de restricción***. Programar con mínimos conocimientos y constructores.
- ***Consistencia***. Lenguaje fácil de aprender y comprender.
- ***Mínimo asombro***. Las cosas no deben comportarse de forma inesperada.
- ***Precisión***. La existencia de una definición precisa para un lenguaje, de tal forma que el comportamiento de los programas sea predecible. Un paso para lograr la precisión es la publicación de un manual o informe del lenguaje por parte del diseñador.
- ***Independencia de la máquina***. Utilizar tipos de datos predefinidos que no involucren detalles de asignación de memoria o de la arquitectura de la máquina.
- ***Seguridad***. Minimizar los errores de la programación. La seguridad está íntimamente relacionada con la *confiabilidad* y con la *precisión*.

Tema 4 Sintaxis

La sintaxis es la estructura de un lenguaje.

Sistema Rotacional para describir las gramáticas libres del contexto, se utiliza la **Forma Backus Naur** o **BNF**.

Estas BNF se presentan en tres formas básicas: **BNF, BNFE (Extendida)** y **diagramas sintácticos**.

4.1 Estructura léxica de los lenguajes de programación.

La estructura léxica de un lenguaje de programación es la estructura de sus palabras o tokens. Puede estudiarse por separado de la estructura semántica, pero está relacionada íntimamente.

Los tokens se clasifican en varias clases:

- ***Palabras reservadas***. También llamadas palabras ***clave***.
- ***Literales o constantes***. Como un número o una cadena de caracteres.
- ***Símbolos especiales***. Como el ";", "< =", "+".
- ***Identificadores***.

Se llaman palabras reservadas porque un identificador no puede tener la misma cadena de caracteres que una palabra reservada.

Los identificadores predefinidos son aquellos a los cuales se les ha dado una interpretación inicial, pero que pueden ser capaces de redefinición, como pueden ser ***integer*** o ***boolean***.

Los identificadores de longitud variable pueden presentar problemas con las palabras reservadas, esto es, el identificador ***doif*** puede verse como un simple identificador o como dos palabras reservadas "**do"** e **"if"**. Por ello, se utiliza el ***principio de la subcadena de mayor longitud***.

Este principio es una regla estándar en la determinación de tokens, en cada punto se reúne en un solo token la cadena más larga posible de caracteres. Esto significa que ***doif*** sería tratado como un único token y por tanto un identificador.

El analizador léxico no especifica el orden de los tokens.

La definición y el reconocimiento de un orden apropiado es tarea de la sintaxis y del análisis sintáctico.

4.2 Gramáticas libres de contexto y BNF.

Una gramática libre de contexto consiste en una serie de reglas gramaticales: las reglas están formadas por un lado izquierdo que es un solo nombre de estructura y a continuación una flecha "→", seguido de un lado derecho formado por una secuencia de elementos que pueden ser símbolos u otros nombres de estructura. Los nombres de las estructuras se conocen como ***no terminales***.

A las palabras o símbolos de token también se les conoce como ***terminales***, dado que jamás se subdividen. Las reglas gramaticales también se llaman ***producciones***, puesto que "producen" las cadenas del lenguaje utilizando derivaciones.

Una gramática libre del contexto también tiene un no terminal distinguido conocido como ***símbolo inicial*** y es con el que se inician todas las derivaciones.

Un lenguaje libre de contexto define un lenguaje conocido como el ***lenguaje de la gramática***. Este lenguaje es el conjunto de todas las cadenas de terminales para las cuales existe una derivación que empieza con el símbolo de inicio y termina con dicha cadena de terminales.

Un ejemplo de la descripción de expresiones aritméticas enteras con adición y multiplicación es el siguiente:

Expresión → expresión + expresión | expresión * expresión | (expresión) | número.

Número → número dígito | digito.

Dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Las reglas BNF son el conjunto de reglas utilizadas para expresar una gramática libre del contexto.

4.3 árboles de análisis sintáctico y árboles de sintaxis abstracta.

La sintaxis establece una estructura, no un significado. Pero el significado de una oración tiene que estar relacionado con su sintaxis.

Semántica dirigida por la sintaxis: es el proceso de asignar la semántica de una construcción a su estructura sintáctica.

El método estándar para utilizar la estructura sintáctica de un programa que determine su semántica es mediante un ***árbol de análisis sintáctico***. Éste describe de manera gráfica el proceso de reemplazo dentro de una derivación.

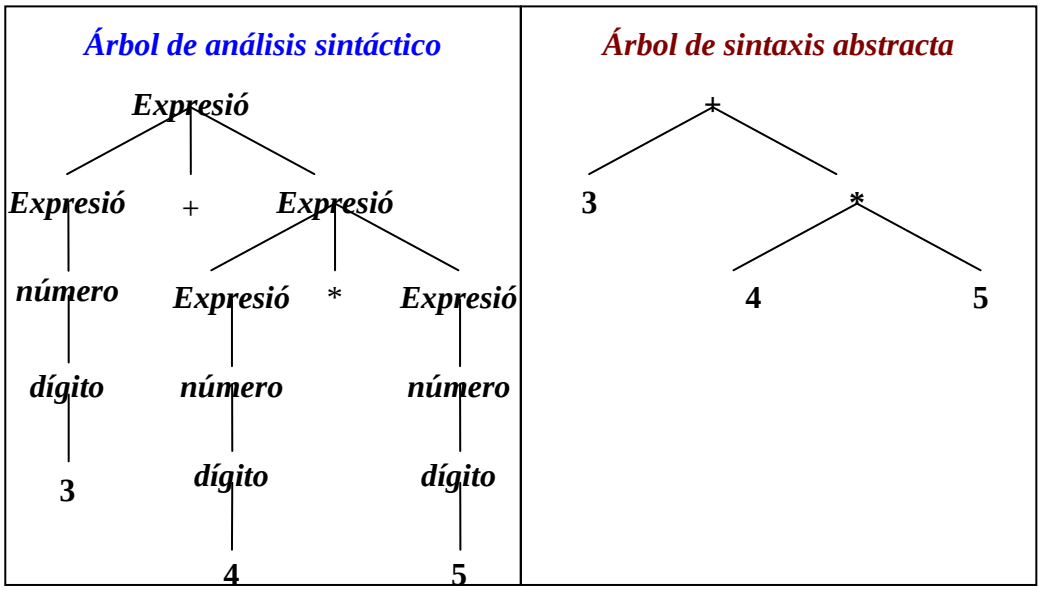
Un ***árbol de análisis gramatical*** está identificado mediante no terminales en los nodos interiores y por terminales en las hojas.

La estructura de un árbol está especificada por las reglas gramaticales del lenguaje y por una derivación de no terminales.

Todos los terminales y no terminales de una derivación están incluidas en un árbol, pero no todos los terminales y no terminales pudieran ser necesarios para determinar la estructura sintáctica de una expresión o de una oración.

Por ejemplo:

El árbol de análisis sintáctico para la expresión 3+4*5 y su árbol de sintaxis abstracta serían los siguientes:



La sintaxis ordinaria se distingue de la abstracta por el nombre de concreta.

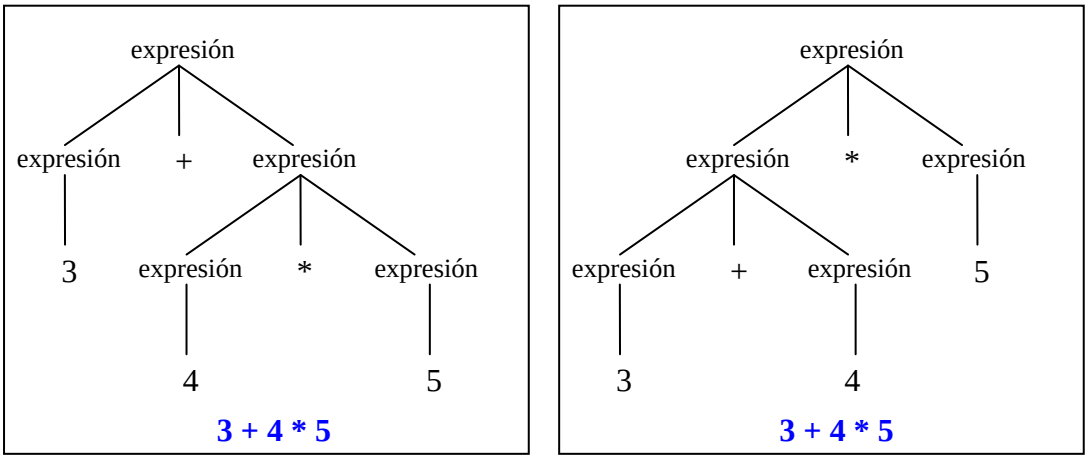
La sintaxis abstracta es la que expresa la estructura esencial de un lenguaje y un traductor construirá un árbol de sintaxis más que un árbol completo de análisis sintáctico.

4.4 ambigüedad, asociatividad y precedencia.

Dos derivaciones diferentes pueden conducir al mismo árbol de análisis sintáctico.

Número=> número **digito**	Número => número **digito**
=> número **4**	=> número **digito digito**
=> número **digito 4**	=> **digito digito digito**
=> número **3 4**	=> **2 digito digito**
=> **digito 3 4**	=> **2 3 digito**
=> **2 3 4**	=> **2 3 4**

Sin embargo, diferentes derivaciones también pueden conducir a diferentes árboles de análisis gramaticales.



Una gramática para la cual son posibles dos árboles diferentes de análisis sintáctico para la misma cadena es ***ambigua***.

La ambigüedad de una gramática puede probarse buscando dos derivaciones diferentes por la izquierda de la misma cadena. Si existen, entonces la gramática debe ser ambigua.

Las gramáticas ambiguas presentan dificultades ya que no expresan una estructura clara. Para resultar útil, se debe eliminar la ambigüedad.

Para eliminar dicha ambigüedad se debe revisar la gramática o incluir nuevas reglas que eliminen la ambigüedad.

Por ejemplo:

La siguiente gramática es ambigua:

Expresión → expresión + expresión | expresión * expresión | (expresión) | número

Número → número dígito | dígito

Dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Las ambigüedades se producen en ***expresión + expresión*** (cuando son números negativos) y en ***expresión * expresión***.

Para eliminar dicha ambigüedad modificamos la regla Expresión por:

Expresión → expresión + ***término*** | ***término*** (recursiva por la izquierda) o

Expresión → ***término*** + expresión | ***término*** (recursiva por la derecha)

Y definimos ***término*** como:

Término → término * término | (expresión) | número

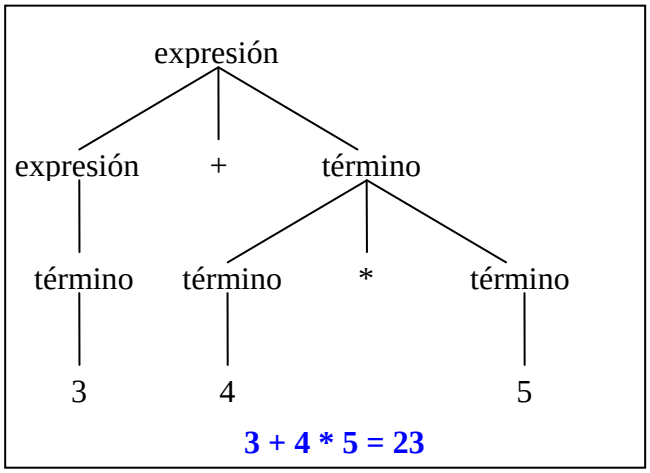
De esta forma eliminamos la ambigüedad de la gramática quedando finalmente como sigue:

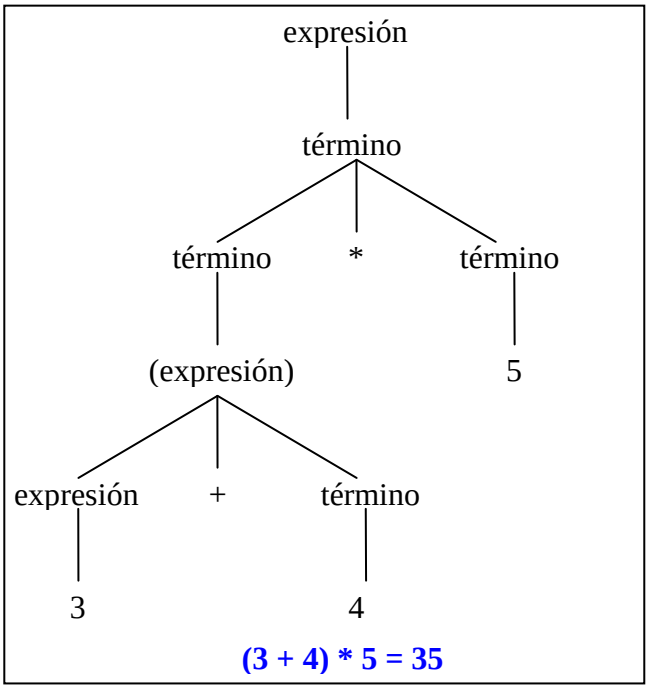
Expresión → expresión + término | término

Término → término * término | (expresión) | número

Número → número dígito | dígito

Dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9





4.5 EBNF y diagramas sintácticos.

La notación EBNF se utiliza para representar las repeticiones sucesivas, añadiendo a la notación BNF los siguientes símbolos:

Las llaves {} : representan cero o más repeticiones.

Los corchetes [] : representan partes opcionales.

Un fallo en las gramáticas EBNF es que no pueden escribirse directamente los árboles de análisis sintáctico ni los árboles sintácticos. Por ello, se utiliza siempre la notación BNF para su representación.

La representación gráfica de los diagramas sintácticos utiliza círculos u óvalos para los terminales, cuadrados o rectángulos para los no terminales y líneas y flechas para indicar la secuencia.

4.6 Técnicas y herramientas de análisis sintáctico.

La forma más simple de un analizador sintáctico es un ***reconocedor***, un programa que acepta o rechaza cadenas en función de si son aceptadas o no en el lenguaje.

Un método de análisis intenta hacer coincidir una entrada con los lados derechos de las reglas gramaticales. Cuando se encuentra una coincidencia, el lado derecho es reemplazado (reducido) por el no terminal de la izquierda.

A estos analizadores se les llama analizadores de ***abajo arriba***, ya que construyen árboles desde las hojas hacia la raíz.

También se les llama analizadores de ***desplazamiento reducción***.

Otro método de análisis es el de ***arriba abajo***, los no terminales se expanden para coincidir con los tokens de entrada y construyen una derivación.

El analizador de ***abajo arriba*** es más poderoso que el de ***arriba abajo***.

Un método muy efectivo es el ***análisis sintáctico por descenso recursivo***: opera convirtiendo los no terminales en un grupo de procedimientos mutuamente recursivos, cuyas acciones están basadas en los lados derechos de los BNF.

En los procedimientos, los lados derechos se interpretan de la siguiente manera:

Los tokens se hacen coincidir directamente con tokens de entrada.

Los no terminales se interpretan como llamadas a los procedimientos de los no terminales.

Un ejemplo de análisis sintáctico por descenso recursivo es:

En la gramática inglesa simplificada, los procedimientos para ***oración***, ***frase-sustantiva*** y ***artículo*** se escribirían de la siguiente forma:

```
Void oración (void)
{   Frase-sustantiva ():
    Frase-verbal();
}
```

```
Void Frase-sustantiva (void)
{   artículo ();
    Sustantivo()
}
```

```
Void artículo (void)
{   if (token == "a") coincidencia ("a");
    Else if (token == "the") coincidencia ("the");
    Else error ();
}
```

Por otro lado, si tenemos una regla como ***expresión*** → ***Expresión*** + ***Término*** | ***Término***, se producen dos errores graves:

Para el procedimiento que representa a ***Expresión*** + ***Término*** se produciría una llamada recursiva antes de encontrar el signo +, por lo que entraría en un bucle infinito.

```
Void Expresión (void)
{ Expresión () + Término ();
}
```

Otro problema es que no hay forma de determinar que opción tomar, si ***Expresión*** + ***Término*** o ***Término***.

Para seguir manteniendo la asociatividad por la izquierda y eliminar la recursividad, utilizamos la notación EBNF en la que las llaves representan la eliminación de la recursividad.

Expresión → **{+** ***Término*****}**

```
Void Expresión (void)
{   Término ();
    Mientras (token == "+")
    {   Coincidencia ("+");
        Término ();
    }
}
```

Cuando las estructuras son opcionales se realiza de la siguiente forma:

Expresión → ***Término*** **[+** ***Expresión*****]**

```
Void Expresión (void)
{   Término ();
    Si (token == "+")
    {   Coincidencia ("+");
        Expresión ();
    }
}
```

Al realizar esta implementación se debe prohibir antes, que ningún token que inicie la parte opcional pueda venir después de ésta, es decir, no existirá ninguna construcción de la forma ***Término*** + ***Expresión*** +.

4.7 Léxico comparado con la sintaxis y con la semántica.

Los límites entre la sintaxis y el léxico no están claramente definidos, ya que si utilizamos la notación BNF, EBNF o diagramas sintácticos, se pueden incluir las estructuras léxicas como parte de la descripción del lenguaje.

Se ha definido la sintaxis como todo lo que se puede definir mediante una gramática libre de contexto, y semántica como todo lo demás que no se puede definir así.

Algunos autores incluyen como sintaxis la declaración de variables antes de que sean utilizadas y la regla de que no se pueden declarar identificadores dentro de un procedimiento. Estas son reglas sensibles al contexto y por lo tanto un analizador sintáctico debería tener información sobre el contexto.

Otro conflicto es cuando un leguaje requiere que ciertas cadenas sean ***identificadores predefinidos*** y no ***palabras reservadas***.

Por ejemplo:

Ada permite la declaración **integer: bolean;**.

La expresión en C (***x***)-***y*** puede entenderse de dos formas:

Si ***x*** es un **tipo**, entonces representa una asignación del valor - ***y*** a ***x***.

Si ***x*** es una **variable**, entonces representa una resta.

Por estas razones, el analizador sintáctico debería tener información del contexto sobre los identificadores disponibles a fin de eliminar ambigüedades.

Tema 5 Semántica básica

Sintaxis: es la forma en la que aparecen los constructores del lenguaje.

Semántica: es lo que los constructores del lenguaje hacen.

Existen varias formas de especificar la semántica:

- Mediante un ***manual de referencia*** de lenguaje. Éste es el método más común. Los inconvenientes son la falta de precisión debido a las descripciones en lenguaje natural y las omisiones o ambigüedades que puede tener.
- Mediante un ***traductor definidor***. Un inconveniente es que las preguntas relacionadas con el comportamiento de un programa no pueden contestarse por adelantado, debemos ejecutar el programa para saber lo que hace. Otro inconveniente es que los errores y las dependencias con la máquina se convierten en parte de la semántica del lenguaje.
- Mediante una ***definición formal***. Estos métodos matemáticos son precisos, complejos y abstractos. El mejor método es la semántica denotacional, que describe a la semántica mediante una serie de funciones.

5.1 Atributos, ligaduras y funciones semánticas.

El uso de ***nombres*** o ***identificadores*** es la forma básica de abstracción en un lenguaje de programación.

Además de los nombres, una descripción de la semántica de un lenguaje de programación requiere de los conceptos de ***localización*** y de ***valor***. Las localizaciones son lugares donde se pueden almacenar los valores.

El significado de un nombre queda determinado por las propiedades o ***atributos*** asociados al mismo.

Por ejemplo:

const n = 5;	Asocia al nombre **n** el atributo de tipo de datos "*constante entera*" y el atributo de valor 5.
Double f (int n){…}	Asocia el atributo "*función*" al nombre ***f*** y los siguientes atributos: Cantidad de parámetros, nombres y tipos. Tipo de datos del valor devuelto. Cuerpo del código cuando se llama a la función.

Ligadura.: Proceso de asociación de atributos a un nombre.

Tiempo de ligadura del atributo: Tiempo en que se está calculando y vinculando un atributo con un nombre durante el proceso de traducción o ejecución. El tiempo de ligadura es el tiempo más corto que las reglas del lenguaje permiten que el atributo esté vinculado.

Los tiempos de ligadura pueden clasificarse en: ***ligadura estática*** y ***ligadura dinámica***.

Una ligadura puede ser ***estática*** o ***dinámica*** dependiendo si los atributos son asociados en ***tiempo de compilación*** o ***tiempo de ejecución*** respectivamente.

Por ejemplo:

int x; asocia estáticamente el tipo de datos "entero".

x=2; asocia dinámicamente el "valor" 2 al nombre x.

Un atributo puede vincularse durante el análisis gramatical o durante el análisis semántico (tiempo de traducción), durante el encadenamiento del programa con las bibliotecas (tiempo de ligado) o durante la carga del programa para su ejecución (tiempo de carga).

Se tienen los siguientes tiempos de ligadura para los atributos de nombres:

- Tiempo de definición del lenguaje.
- Tiempo de implementación del lenguaje.
- Tiempo de traducción (tiempo de compilación).
- Tiempo de ligado.
- Tiempo de carga.
- Tiempo de ejecución.

Todos, con excepción del último, representan ligaduras estáticas.

El traductor debe conservar las ligaduras de tal manera que se den significados apropiados a los nombres durante la traducción y la ejecución. Un traductor hace lo anterior creando una estructura de datos para mantener la información. Podemos pensar en ellos como una función que expresa la ligadura de los atributos a los nombres. Esta función se conoce como la ***tabla de símbolos***. La tabla de símbolos es una función de nombres hacia atributos.

Existe una diferencia fundamental entre la forma en la que se mantiene una tabla de símbolos a través de un intérprete y la forma en la que la mantiene un compilador.

Compilador:

La tabla de símbolos para un compilador puede ilustrarse como:

Nombres → Tabla de símbolos → Atributos estáticos.

Durante la ejecución de un programa compilado, deben mantenerse ciertos atributos como por ejemplo las localizaciones en memoria y los valores. Un compilador genera código que conserva estos atributos. La parte de asignación de memoria en este proceso, es decir, la ligadura de nombres a localizaciones, se denomina ***entorno***:

Nombres → Entorno → Localizaciones.

Las ligaduras de las localizaciones de almacenamiento junto con los valores se conocen como la ***memoria***, ***almacén*** o ***estado***:

Localizaciones → Memoria → Valores.

Intérprete:

En un intérprete, se combinan la tabla de símbolos y el entorno, ya que durante la ejecución se procesan atributos estáticos y dinámicos. Por lo general, en esta función también se incluye la memoria:

Nombres → Entorno → Atributos (incluyendo localizaciones y valores).

5.2 Declaraciones, bloques y alcance.

Las ***Declaraciones*** son un método para establecer ligaduras. Las ligaduras pueden determinarse mediante una declaración, ya sea de manera implícita o explícita.

Por ejemplo:

Int x	Establece explícitamente el tipo de datos de ***x*** utilizando la palabra clave ***int***, pero la localización está vinculada implícitamente.
Int x=0	Vincula de manera explícita a ***0*** como valor inicial de ***x***.

Definiciones: declaraciones que vinculan a todos los atributos potenciales.

Declaraciones: declaraciones que especifican sólo parcialmente atributos.

Por ejemplo:

Double f (int x);	**Declaración.** Especifica únicamente el tipo de datos de la función, el tipo de datos devuelto, la cantidad de parámetros así como el tipo y nombre de cada uno de ellos, pero no especifica el código de la función ***f***. No se considera una definición por faltar algún parámetro.
Struct x;	**Declaración**. Especifica un tipo incompleto, por lo que tampoco se considera una definición.
Int x = 0;	**Definición**. Vincula todos los atributos: tipo de datos ***int***, nombre ***x***, valor ***0***.
Char * name;	**Declaración**. Falta el atributo de valor.
Typedef int* IntPtr=0;	**Definición.**
Struct rec;	**Declaración.** Tipo de datos incompleto.
Int gcd(int, int);	**Declaración**. Falta el código de la función.
Double y=1.0;	**Definición.**
Extern int z;	**Declaración**. Falta el atributo de valor.

Un ***bloque*** consiste en una secuencia de declaraciones seguidas por una secuencia de enunciados y rodeado por marcadores sintácticos. Las declaraciones que se encuentran dentro del bloque se conocen como ***locales*** y las que se encuentran fuera de cualquier enunciado compuesto se llaman ***globales*** o ***externas***.

Por ejemplo:

```
Int x
Double y                    Declaraciones externas o globales
Main ()
{   int i, j                Declaraciones locales
    …
}
```

Otros constructores del lenguaje son fuentes importantes de declaraciones: todos los tipos de datos estructurados se definen utilizando declaraciones locales asociados con el tipo. Por ejemplo:

```
Struct A
{       int x;
        Double y;                       Local en A.
        Struct
        {       int* x;
                Char y;                 Declaraciones miembro anidadas.
        } z;
};
```

En lenguajes orientados a objetos, la ***clase*** es una fuente importante de declaraciones.

Las declaraciones vinculan varios atributos a los nombres, dependiendo del tipo de declaración. El ***alcance de un vínculo*** es la región del programa sobre la cual se conserva el vínculo.

Alcance léxico: cuando el alcance de un vínculo queda limitado al bloque en el cual aparece su declaración asociada.

En C el alcance de una declaración se extiende en el punto justo después de la misma hasta el fin del bloque en el que está localizada.

Una característica de los bloques es que las declaraciones en bloques anidados toman precedencia sobre las declaraciones anteriores:

```
Int x;
Void p (void)
{       char x;
        x='a';   /* asigna el valor "a" a la variable local x */
        …
}
Main ()
{       x=2;    /* asigna el valor "2" a la variable global x */
        …
}
```

Dentro del cuerpo de **p**, la declaración **char x** tiene precedencia sobre la declaración global **int x**. Por lo tanto, el entero global ***x*** no puede ser accedido desde dentro de **p**. La declaración global de **x** se dice que tiene una ***apertura en el alcance*** dentro de **p**.

La ***visibilidad*** incluye únicamente aquellas regiones de un programa donde las ligaduras de una declaración son aplicables, mientras que el ***alcance*** incluye a los agujeros del alcance.

En C++, el ***operador de resolución de alcance ::*** (doble dos puntos) puede utilizarse para tener acceso a las declaraciones ocultas, siempre y cuando éstas sean globales.

```
Int x;
Void p (void)
{       char x;
        x = 'a';          /* asigna el valor "a" a la variable local x */
        ::x = 42;         /* asigna el valor 42 a la variable global x */
        …
}
Main ()
{       x=2;    /* asigna el valor "2" a la variable global x */
        …
}
```

Las declaraciones y los modificadores de declaraciones pueden alterar el acceso y el alcance de las declaraciones.

Por ejemplo en el código anterior en C, podríamos incluir la sentencia "***::x= 1;***" dentro del bloque **p**, para asignar el valor "1" a la variable global **x** en vez de a la variable **x** local. A los dobles dos puntos se le denomina operador de resolución de alcance.

En Ada, el operador de resolución de alcance se hace mediante el nombre del procedimiento seguido de un punto y la variable local que se desea asignar. Si el ejemplo anterior estuviese escrito en Ada, para asignar un valor a la variable **x** dentro de **p** desde main(), se añadiría la sentencia **p . x = 1**.

Las vinculaciones establecidas por las declaraciones se conservan mediante la ***tabla de símbolos***. La forma en la que la tabla de símbolos procesa las declaraciones debe corresponder con el alcance de cada declaración.

Reglas de alcance diferentes requieren de un comportamiento diferente de la tabla de símbolos.

5.3 La tabla de Símbolos.

La tabla de símbolos es como un diccionario variable: debe poder insertar, buscar y cancelar nombres con sus atributos asociados representando las vinculaciones.

El mantenimiento de la información de alcance en un lenguaje con alcance léxico y estructura de bloques requiere que las declaraciones sean procesadas en forma de pila.

Se puede considerar a la tabla de símbolos como una estructura de datos que contiene un conjunto de nombres, cada uno de ellos con una pila de declaraciones asociada.

Si la tabla de símbolos es manejada por un compilador y las ligaduras de las declaraciones son estáticas, entonces procesará las declaraciones de manera estática.

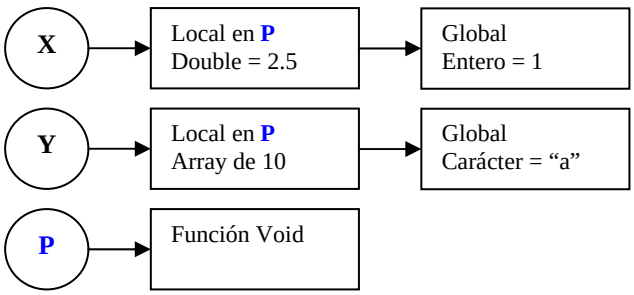
Si la tabla de símbolos está administrada dinámicamente, entonces las declaraciones se procesan conforme se van encontrando. Esto da como resultado una regla de alcance diferente, lo que por lo general se conoce como alcance dinámico.

Ejemplo:

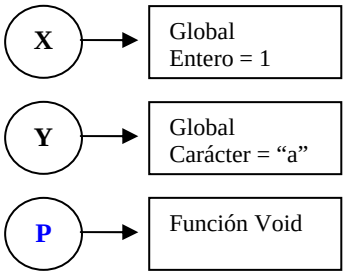
```
1     Int x =1;
2     Char y =’a’;
3     Void P (void){
4           double x = 2.5; printf (“%c\n”, y);
5           {     int y[10];
6           }
7     }
8     Void Q (void){
9           int y = 42; printf (“%d\n”, x);
10          P ();
11    }
12    Main(){
13          char x = ‘b’; Q (); Return 0;
14    }
```

Alcance estático:

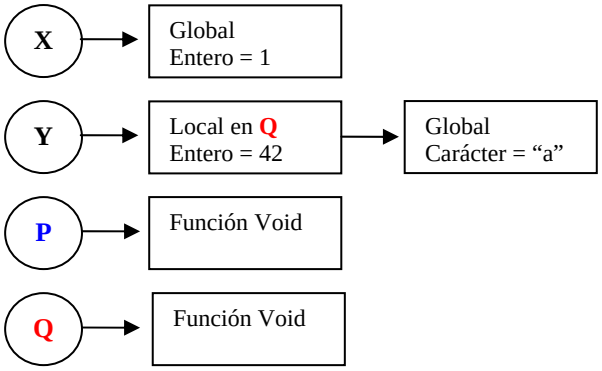
Líneas 1 a 5:



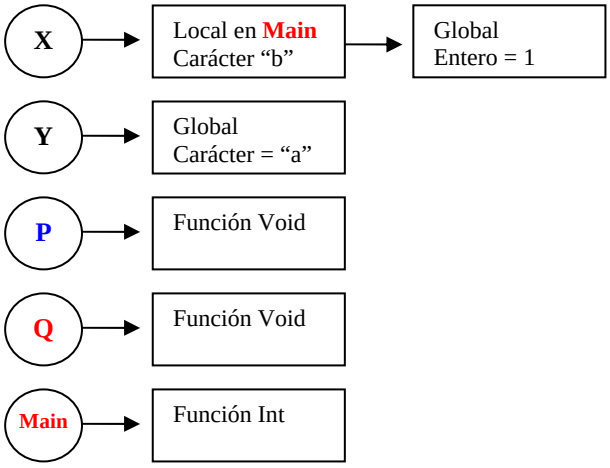
Líneas 6 y 7: Al cerrarse el bloque definido en la línea 6, el primer elemento de la pila correspondiente a Y desaparece. Lo mismo ocurre en la línea 7, pero esta vez se trata de la función P, por lo que desaparece el primer elemento de la pila X quedando finalmente:



Líneas 8 a 10:



Líneas 11 a 14:

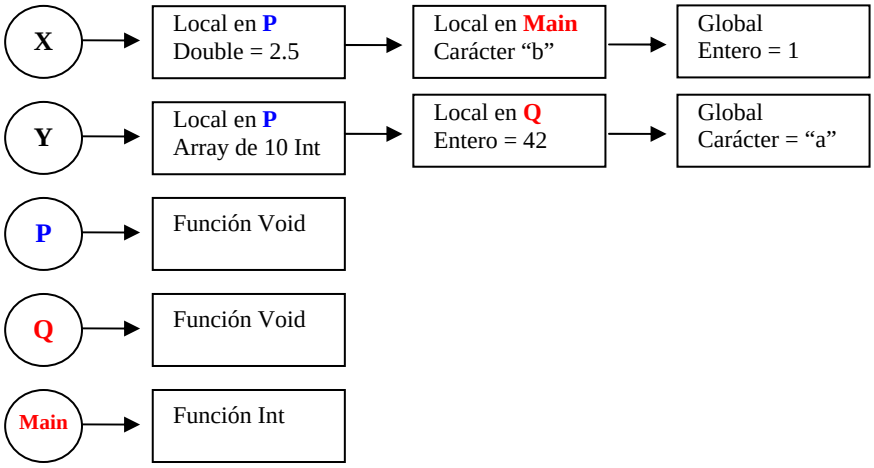


Alcance Dinámico:

La ejecución del código sería en el siguiente orden:

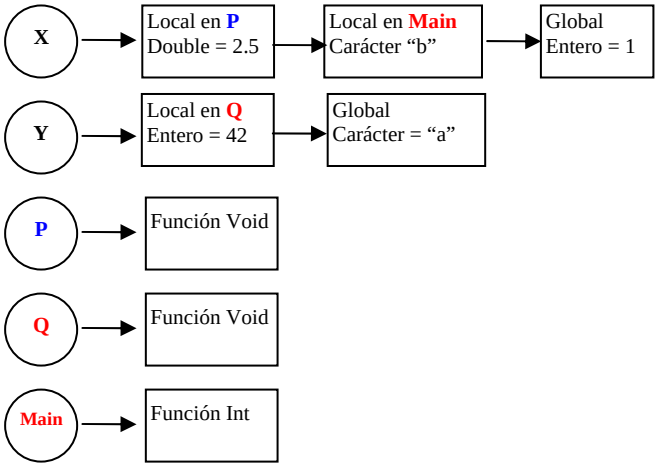
```
1     1     Int x =1;
2     2     Char y =’a’;
3     12    Main(){
4     13          char x = ‘b’; Q ();
5     8     Void Q (void){
6     9           int y = 42; printf (“%d\n”, x);
7     10          P ();
8     3     Void P (void){
9     4           double x = 2.5; printf (“%c\n”, y);
10    5           {     int y[10];
11    6           }
12    7     }
13    11    }
14    14    Return 0;}
```

La Tabla de símbolos hasta la línea 10 inclusive sería la siguiente:

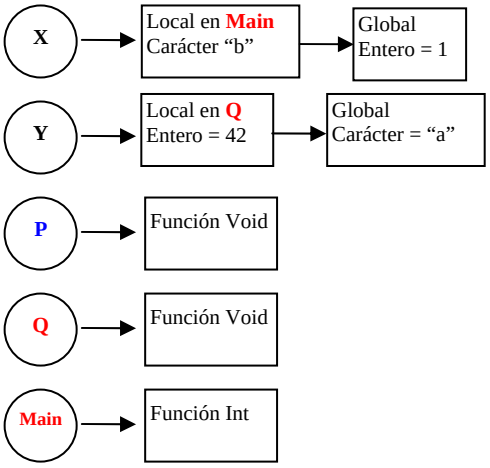


Conforme se ejecutan las líneas 11, 12, 13 y 14, se van eliminando los elementos de las pilas correspondientes, quedando de la siguiente forma:

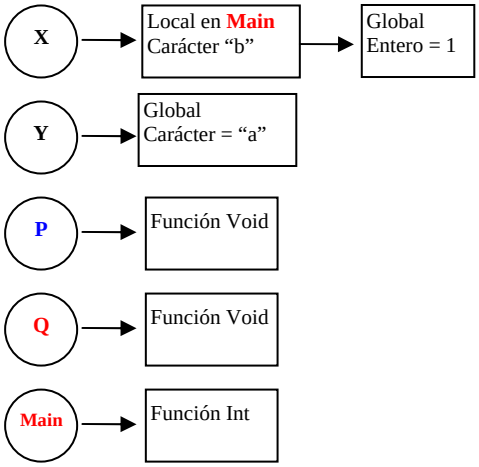
Línea 11 }:



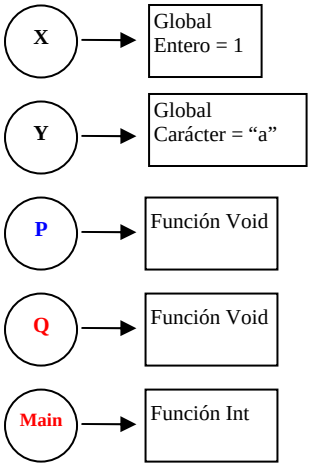
Línea 12 }:



Línea 13 }:



Línea 1:



El primer problema es que bajo el alcance dinámico, la declaración no puede determinarse mediante la simple lectura del programa. En vez de ello, el programa deberá ser ejecutado y diferentes ejecuciones pueden producir diferentes resultados. Por lo tanto, la semántica de una función puede cambiar conforme avanza la ejecución del programa.

Dado que las referencias a variables no locales no pueden predecirse antes de la ejecución, tampoco pueden definirse los tipos de datos de estas variables.

La ligadura estática de los tipos de datos y el alcance dinámico son incompatibles.

El entorno en tiempo de ejecución es más simple mediante el alcance dinámico.

Cualquier estructura de alcance que pueda ser referenciada directamente en un lenguaje, también debe tener su propia tabla de símbolos.

Las declaraciones ***struct*** deben contener declaraciones adicionales de los campos de datos dentro de ellas, y éstas deben ser accesibles siempre que estén en el alcance. Esto significa dos cosas:

Una declaración ***struct*** realmente contiene una tabla de símbolos locales que es en sí un atributo, y

Esta tabla de símbolos local no puede eliminarse hasta que la variable ***struct*** sea eliminada de la tabla de símbolos “global” del programa.

5.4 Resolución y sobrecarga de nombres.

Sobrecarga. Cuando un símbolo se refiere a dos o más operaciones completamente distintas. Por ejemplo el símbolo "+" puede referirse a la suma de números o a la concatenación de cadenas.

Resolución de sobrecarga. Consiste en ampliar la operación del mecanismo de búsqueda en la tabla de símbolos a fin de llevar a cabo búsquedas basadas no únicamente en el nombre de la función, sino también en la cantidad de sus parámetros y en sus tipos de datos.

Cuando se sobrecarga un operador incorporado, debemos aceptar las propiedades sintácticas del operador; no podemos cambiar su asociatividad o su precedencia.

No existe diferencia semántica entre operadores y funciones, únicamente la diferencia sintáctica que los operadores escriben en forma ***infija*** y las funciones en forma ***prefija***.

Utilizar el mismo nombre para cosas de tipos completamente diferentes puede resultar muy confuso, y los programas tienen poco que ganar utilizando esta sobrecarga.

Sin embargo, este tipo de sobrecarga resulta útil como manera de limitar la cantidad de nombres que uno realmente tiene que recordar en un programa.

Por ejemplo:

```
Typedef struct A A;
Struct A
{       int datos;
        A * siguiente;
};
```

Tan sólo deberemos recordar la estructura llamada A cuyos campos son A.datos y A.siguiente.

```
Class A
{       A A(A A)      // tipo de retorno, nombre de la función, tipo del parámetro y parámetro.
        {       A:
                For(;;)
                {if (A.A(A) == A) break A;}
                Return A;
        }
}
```

5.5 Asignación, tiempo de vida y el entorno.

Dependiendo del lenguaje, el entorno se puede construir estáticamente, dinámicamente, o una mezcla de ambos.

No todos los nombres en un programa están vinculados con localizaciones. En un lenguaje compilado, los nombres de las constantes y de los tipos de datos pueden representar puramente cantidades en tiempo de compilación que no existen en tiempo de carga o en tiempo de ejecución.

Las declaraciones se utilizan para construir el entorno así como la tabla de símbolos.

En un compilador, las declaraciones son utilizadas para indicar el código de asignación que el compilador tiene que generar.

En un intérprete, se combinan la tabla de símbolos y el entorno, por lo que la ligadura de atributos por una declaración en un intérprete incluye la ligadura de las localizaciones.

En un lenguaje con estructura de bloques, las variables globales se asignan estáticamente y las variables locales dinámicamente cuando la ejecución llega al bloque en cuestión.

Cuando se entra en cada uno de los bloques, las variables declaradas al principio de cada bloque se asignan, y cuando se sale de cada bloque, estas mismas variables se desasignan.

Si además el lenguaje tiene alcance léxico, se pueden asociar el mismo nombre con varias localizaciones diferentes aunque únicamente una de ellas pueda ser accedida.

Un objeto es un área de almacenamiento asignada en el entorno como resultado del procesamiento de una declaración y puede ser accedido más allá de su tiempo de vida.

El ***tiempo de vida*** o extensión de un objeto es la duración de su asignación en el entorno.

Un ***apuntador*** es un objeto cuyo valor almacenado es una referencia a otro objeto.

Para permitir la asignación arbitraria y la desasignación, el entorno debe tener un área en la memoria conocida como ***montículo*** o ***montón***.

En la implementación típica del entorno, la pila (para la asignación automática) y el montón (para la asignación dinámica) se mantienen en secciones diferentes de la memoria.

Estas tres secciones de la memoria se podrían colocar en cualquier parte de la memoria, una estrategia común es colocar las tres seguidas, en primer término el área global, seguida de la pila y finalmente el montón, creciendo éste último y la pila en direcciones opuestas para evitar colisiones.

Muchos lenguajes requieren que la desasignación del montón sea administrada de manera automática al igual que la pila, excepto que la asignación debe conservarse bajo control del usuario. Todos los lenguajes

funcionales requieren que el montón sea completamente automático, sin ninguna asignación o desasignación bajo control del programador. Java, por otra parte, permite la asignación, pero no la desasignación.

El motivo por el que los lenguajes de programación no permiten el control sobre la asignación y desasignación del montón, es que éstas son operaciones poco seguras y pueden introducir un comportamiento erróneo en tiempo de ejecución.

En un lenguaje con estructura de bloques con asignación de montones, existen tres tipos de asignación en el entorno: ***estático*** (para variables globales), ***automático*** (para variables locales) y ***dinámicos*** (para asignación de montones). Esas categorías también se conocen como ***clases de almacenamiento*** de la variable.

5.6 Variables y Constantes.

Una ***Variable*** es un objeto cuyo valor almacenado puede cambiar durante la ejecución.

La forma principal en que una variable cambia su valor es a través del enunciado de ***asignación***.

Dado que una variable tiene a la vez una localización y un valor almacenado en dicha localización, es importante distinguir claramente entre ambos. El valor almacenado en la localización de una variable se conoce como ***valor r*** y la localización de la variable como ***valor l***.

Asignación por compartición. Cuando se copian las localizaciones en vez de los valores.

Asignación por clonación. Cuando se asigna una localización nueva, se copia el valor de la variable y se vincula la anterior variable a la nueva.

Una ***Constante*** tiene un valor fijo durante la duración de su existencia dentro de un programa. Una constante es igual que una variable, excepto que no tiene un atributo de localización, sino solamente un valor.

Decimos a veces que una constante tiene ***semántica de valor*** en lugar de la semántica de almacenamiento de una variable.

Una vez computado su valor, no puede modificarse, y la localización de la constante no se puede referir de manera explícita a través de un programa.

Una constante es esencialmente el nombre de un valor. La representación de los valores se conocen como ***literales***.

Las constantes pueden ser estáticas y dinámicas:

Una constante ***estática*** es aquella cuyo valor se puede calcular antes de la ejecución, en tanto que una constante dinámica tiene un valor que se puede computar únicamente durante la ejecución. Las constantes estáticas también pueden ser de dos tipos: algunas constantes pueden ser calculadas en tiempo de traducción (tiempo de compilación), en tanto que otras solamente pueden ser computables en tiempo de carga o precisamente al principio de la ejecución del programa.

Una constante en tiempo de compilación puede utilizarse por un compilador para mejorar la eficiencia de un programa y no necesita realmente ocupar memoria, en tanto que una constante en tiempo de carga o ***dinámica*** debe ser computada ya sea al arranque o conforme avanza la ejecución y debe ser almacenada en la memoria.

Una ***constante de manifiesto*** es el nombre de un literal, como por ejemplo ***const int a = 2***. Por ejemplo:

```
#include <stdio.h>
#include <time.h>
const int a =2;
const int b = 27+2*2;
/* Advertencia, código ilegal en C, ya que sólo puede ser calculado a partir de literales */
const int c= (int) time(0);
int f(int x)
{       const int d =x+1;
        return b+c;
}
```

En este código, **a** y **b** son constantes en tiempo de compilación (a es una constante de manifiesto), en tanto que **c** es una constante estática (en tiempo de carga) y **d** es una constante dinámica.

5.7 Alias, referencias pendientes y basura.

Existen varios problemas con las reglas convencionales de nombramiento y asignación dinámica en los lenguajes estructurados por bloques, y estos son: los Alias, las referencias pendientes y la basura.

5.7.1 Alias.

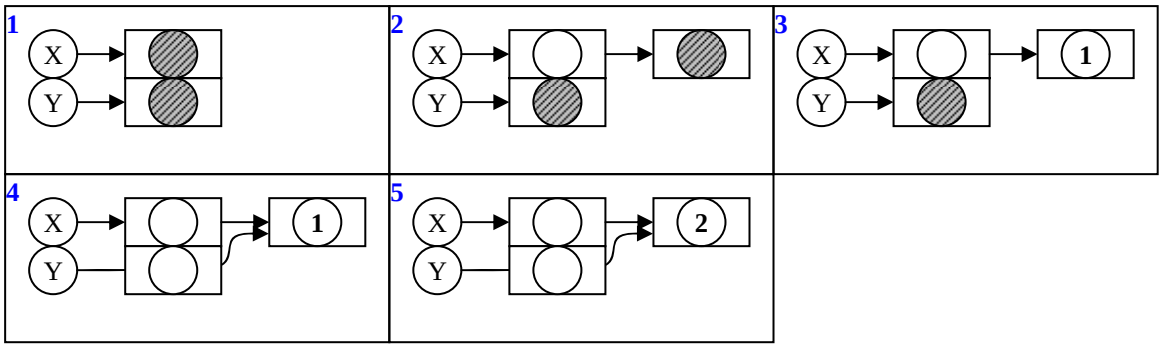
Un Alias ocurre cuando el mismo objeto está vinculado a dos nombres diferentes al mismo tiempo.

El aliasado puede ocurrir cuando se llama a un procedimiento o a través del uso de apuntadores y también a través de la asignación por compartición.

Los problemas que presentan los alias son los efectos colaterales que producen, dichos efectos son altamente dañinos y el uso de alias presenta dificultades para controlarlos.

Por ejemplo:

```
Int *X, *Y;
X=(int*) malloc (sizeof (int));
*X =1
Y=X;
*Y=2;
Printf("%d\n",*X);
```



En este código, la línea 5 ha cambiado también el valor de X sin que en el código se haga mención explícita de ello.

Se define como ***efecto colateral*** de un enunciado como cualquier cambio en el valor de una variable que persiste más allá de la ejecución del enunciado.

El aliasado debido a la asignación de apuntadores es difícil de controlar.

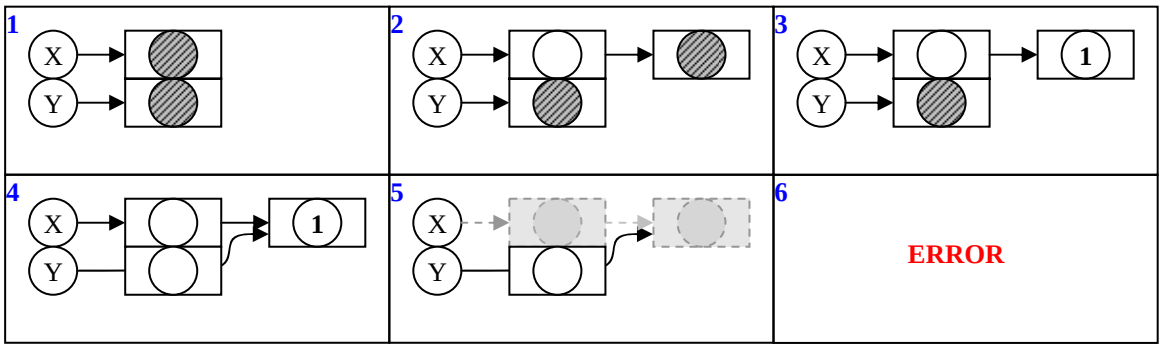
5.7.2 Referencias pendientes.

Las referencias pendientes es un segundo problema que se puede presentar con el uso de apuntadores.

Una ***Referencia pendiente*** es una localización que ha sido desasignada del entorno, pero a la que el programa puede tener acceso todavía. Otra definición es que una localización puede ser accedida más allá del tiempo de vida en el entorno.

Por ejemplo:

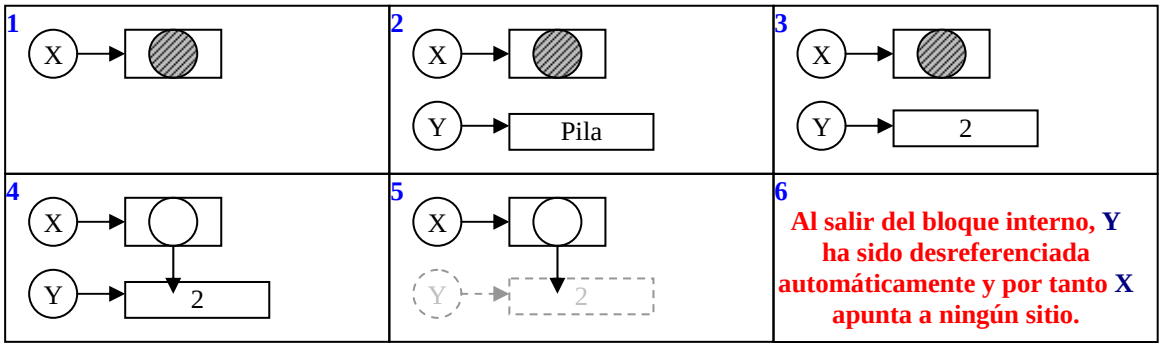
```
Int *X, *Y;
X=(int*) malloc (sizeof (int));
*X=1;
Y=X;
Free (X);
Printf("%d\n"; *Y);
```



También es posible que se produzcan referencias pendientes con la desasignación automática al salir de un bloque.

Por ejemplo:

```
{ int *X;
{ int Y;
Y=2;
X= &Y;
}
/* X es ahora una referencia pendiente */
}
```



5.7.3 Basura.

La manera más fácil de eliminar el problema de las referencias pendientes es no llevar a cabo ninguna desasignación de ningún tipo en el entorno. Esto causa tener el problema de basura, es decir datos almacenados en memoria que no se van a utilizar más por el programa.

La basura es un problema en la ejecución de un programa dado que se trata de memoria desperdiciada.

Los programas que producen basura tienen menos errores serios que los programas que contienen referencias pendientes.

Un programa que tiene acceso a referencias pendientes, puede ejecutarse pero producir resultados incorrectos, corromper otros programas o causar errores en tiempo de ejecución difíciles de localizar.

Los lenguajes que recuperan automáticamente la basura se dice que llevan a cabo la ***recolección de basura***.

La administración de la memoria basada en pilas en un entorno de un lenguaje estructurado en bloques se puede llamar un tipo simple de recolección de basura: cuando se sale del alcance, el entorno recupera la localización asignada a dicha variable al desapilar la memoria que se le había asignado antes.

Tema 6 Tipos de datos

La mayoría de los lenguajes incluyen un conjunto de datos simples, como ***enteros***, ***reales*** y ***booleanos***, así como mecanismos para construir nuevos datos a partir de estos. Estas abstracciones contribuyen en todas las metas del diseño, que son: legibilidad, capacidad de escritura, confiabilidad e independencia de la máquina.

También debemos ser conscientes de los problemas a los cuales nos puede conducir esta abstracción. Uno es que las dependencias de la máquina son parte de la implementación de dichas abstracciones.

Por ejemplo: pensamos en los enteros como un conjunto infinito, pero todo ordenador tiene un valor de representación mínimo y otro máximo, convirtiendo dicho conjunto en uno ***finito***.

Las razones para utilizar la verificación de tipos estática son:

1. la información de tipos estáticos permite a los compiladores asignar memoria con eficiencia y generar código que manipula los datos eficientemente, con lo cual mejora la eficiencia de la ejecución.
2. un compilador puede utilizar los tipos estáticos a fin de reducir la cantidad de código con lo que se mejora la eficiencia de traducción.
3. permite que muchos errores estándar de programación sean detectados rápidamente, lo que mejora la capacidad de escritura, es decir, la eficiencia de la codificación de programas.
4. la verificación de tipos estáticos mejora la seguridad y la confiabilidad de un programa al reducir la cantidad de errores de ejecución que pueden ocurrir.
5. los tipos explícitos mejoran la legibilidad al documentar el diseño de los datos.
6. pueden utilizarse los tipos explícitos para eliminar ambigüedades en los programas.
7. los programadores pueden utilizar los tipos explícitos combinados con la verificación de tipos estáticos como una herramienta de diseño.
8. la tipificación estática de interfaces mejora el desarrollo de los programas grandes al verificar la consistencia y corrección de la interfaz.

6.1 Tipos de datos e información de tipos.

Un tipo aporta algunas propiedades, como los valores que pueden almacenarse, y la forma en que esos valores se representan internamente.

Un tipo de datos es un conjunto de valores, junto con un conjunto de operaciones sobre dichos valores y con ciertas propiedades.

El proceso por el cual pasa un interprete para determinar si la información de tipos es consistente se conoce como ***verificación de tipos***.

El proceso de asignar tipos a expresiones se conoce como ***inferencia de tipos***. Este proceso puede considerarse como una operación por separado y llevarse a cabo durante la verificación de tipos o considerarse como parte de la verificación de tipos misma.

Todo lenguaje ofrece una diversidad de maneras para construir tipos más complejos, basándose en los tipos básicos; estos mecanismos se conocen como ***constructores de tipo***, y los tipos creados se conocen como ***tipos definidos por el usuario***.

Ejemplo: int a [10]. Crea en C una variable cuyo tipo es “arreglo de enteros” y cuyo tamaño es 10.

Las declaraciones de tipo tienen reglas, y éstas se conocen como algoritmos de ***equivalencia de tipo***. Los métodos utilizados para la construcción de tipos, el algoritmo de equivalencia de tipos, y las reglas de inferencia y de corrección de tipos, se conocen como un ***sistema de tipos***.

Un tipificado fuerte asegura que la mayoría de los programas peligrosos (es decir, programas con errores que corrompen datos) se rechazarán en tiempo de traducción, y aquellos que no se rechacen, causarán un error antes de cualquier corrupción de datos.

Los lenguajes sin sistemas de tipificación estática por lo general se conocen como ***lenguajes sin tipificación*** o ***lenguajes con tipificación dinámica***.

En un lenguaje sin tipos, toda la verificación de seguridad se lleva a cabo en tiempo de ejecución.

6.2 Tipos simples.

En esta sección se utilizarán nombres genéricos y en próximas secciones veremos como se diferencian en los distintos lenguajes.

Todos los lenguajes tienen tipos genéricos a través de los cuales se construyen los demás tipos. Los tipos predefinidos son principalmente ***simples***: tipos que no tienen otra estructura que su aritmética inherente o una estructura secuencial.

Sin embargo existen tipos simples que no están predefinidos: Los tipos ***enumerados*** y los tipos de ***subrango***.

Los tipos ***enumerados*** son conjuntos cuyos elementos se denominan y se listan de manera explícita. Los tipos enumerados en la mayoría de los lenguajes presentan un orden atendiendo al orden en el cual se listan, y normalmente existe una función predecesora o sucesora.

Los tipos ***subrangos*** son subconjuntos contiguos de tipos simples especificando por lo menos el menor y el mayor elemento. Todo valor tiene un elemento anterior y siguiente. Estos tipos se conocen como tipos ordinales pues tienen un orden discreto en el conjunto. Como tipos ordinales en cualquier lenguaje tenemos: enteros numéricos, enumeraciones y los subrangos. Estos siempre tienen operadores de comparación (<=, <>…) y a menudo también tienen operaciones de sucesor y predecesor. No todos los tipos con operadores de comparación son ordinales, como ejemplo tenemos los reales que tienen operaciones de comparación pero no sucesor ni predecesor.

Los esquemas de asignación para los tipos simples están condicionados por el hardware subyacente, ya que para obtener eficiencia la implementación de estos tipos normalmente se apoya en el hardware. Sin embargo cada vez más los lenguajes están demandando representaciones normalizadas como el estándar IEE 754.

6.3 Constructores de tipos.

Ya que los tipos de datos son conjuntos, se pueden utilizar las operaciones de conjuntos para crear nuevos tipos de datos a partir de los existentes. Estas operaciones incluyen: el ***producto cartesiano***, la ***unión***, el ***conjunto potencia***, el ***conjunto de función*** y el ***subconjunto***.

Cuando se aplican estas operaciones a los tipos se les denomina ***constructores de tipos***.

También existen algunas operaciones de conjuntos que no corresponden con ningún constructor como por ejemplo la intersección.

6.3.1 Producto cartesiano.

Dados los conjuntos U y V, podemos formar el producto cartesiano o cruz formado por todos los pares ordenados de elementos de U y V;

En muchos lenguajes el constructor de tipos del producto cartesiano está disponible como la construcción de estructuras o de registros. Por ejemplo: el siguiente código en C construye el tipo de producto cartesiano ***int x char x double***.

```
Struct IntCharReal {
    int i;
    Char c;
    Double r;
}
```

Existe una diferencia entre un producto cartesiano y una estructura de registro: en una estructura los componentes tienen nombre, en tanto que en un producto cartesiano se hace referencia a ellos por su posición.

Las proyecciones en la estructura de registro están dadas por la ***operación de selector de componentes***: si ***x*** es del tipo **IntCharReal**, entonces ***x.i*** es la proyección de ***x*** hacia enteros.

La mayoría de los lenguajes de programación consideran los nombres componentes como parte del tipo definido por un registro, por lo que la siguiente estructura puede considerarse diferente de la anterior aunque representen el mismo producto cartesiano.

```
Struct IntCharReal {
    int j;
    Char ch;
    Double d;
}
```

Algunos lenguajes tienen una forma más pura del tipo estructura de registro, que es en esencia idéntica al producto cartesiano, donde a menudo se les denomina tuplas. Por ejemplo en ML podemos definir IntCharReal como:

Type IntCharReal = int * char * real;

Las funciones de proyección se escriben entonces como #3 (2, # “a”, 3.14) = 3.14.

Un tipo de datos que se encuentra en los lenguajes orientados a objetos, que está relacionado con las estructuras, es la clase.

Un esquema típico de asignación para los tipos de producto cartesiano es la asignación secuencial, según el espacio que requiere cada componente. Por ejemplo:

Type IntCharReal = int * char * real;

Requiere un espacio de 13 bytes: 4 para el entero, 1 para el carácter y 8 para el Real.

6.3.2 Unión.

Una segunda construcción es la unión de dos tipos: esta se forma con la unión teórica de los valores que forman cada conjunto.

Las uniones (*unions*) tienen un cometido parecido a las estructuras, esto es, agrupar en una sola variable varios valores. Sin embargo, al contrario que las estructuras, las uniones sólo pueden guardar un valor a la vez. Esto significa que si asignamos valor a uno de los componentes de una variable unión, los demás componentes dejarán de tener un valor asignado.

Unión: una unión es un tipo de datos compuesto que puede almacenar valores de diferentes tipos, aunque no a la vez.

Se puede pensar en las uniones como tipos “mutantes", es decir, como variables que pueden almacenar valores de distintos tipos. Al igual que pasa con las variables ordinarias el hecho de asignar a una unión un valor destruye automáticamente los valores anteriores (sean del tipo que sean). Al fin y al cabo, lo que el compilador hace es reservar espacio para una variable de tamaño del elemento de mayor tamaño de la unión.

Las uniones se pueden presentar de dos formas: las uniones discriminadas y las indiscriminadas.

Una unión es ***discriminada*** si se le agrega una etiqueta o discriminador para distinguir el tipo de elemento que es, es decir, de que conjunto proviene.

Las uniones ***indiscriminadas*** no tienen etiquetas y deben de suponerse el tipo de cualquier valor en particular (este tipo de unión en un lenguaje hace que el sistema de tipos sea inseguro).

En C y C++:

```
Union IntOrReal {
    int i;
    Double r;
};
```

Al igual que con struct existen nombres para diferenciar los distintos componentes (i y r). Los nombres son necesarios por que comunican al intérprete el tipo con el que deben interpretarse los bits dentro de la unión.

Ada cuenta con mecanismos de unión completamente seguro, llamado registro con variantes:

```
Type Disc is (IsIjnt, Irreal);
Type IntOrReal (whith: Disc) is
Record
Case which is
When IsInt => i : integer;
When IsReal => r: float;
End case;
End record;
…
X: IntOrTeal := (IsReal, 2.3);
```

Las uniones pueden resultar útiles para reducir los requerimientos de asignación de memoria para las estructuras cuando no se necesitan, simultáneamente, diferentes elementos de datos. Esto se debe a que a las uniones se les asigna un espacio de memoria equivalente al mayor necesario para cada uno de sus componentes y los valores de cada componente se almacenan en regiones superpuestas de la memoria.

Las uniones, sin embargo, no son necesarias en lenguajes orientados a objetos, ya que en un mejor diseño sería utilizar la herencia para representar diferentes requerimientos de datos que no se superponen. Por lo tanto Java no utiliza uniones, C++ sí utiliza uniones principalmente por compatibilidad con C.

6.3.3 Subconjuntos.

En matemáticas se pueden definir subconjuntos al dar una regla para distinguir sus elementos, como pos_int = {x|x es un entero y x > 0}. En los lenguajes de programación se puede hacer algo parecido para definir nuevos tipos que serán subconjuntos de tipos conocidos.

En ocasiones los subconjuntos heredan operaciones de sus tipos padres.

Una perspectiva alternativa a la relación de subtipos es definirla en términos de operaciones compartidas. Esto es, un tipo S es subtipo de un tipo T si y sólo si todas las operaciones de los valores de T también pueden aplicarse a valores del tipo S.

La herencia en los lenguajes orientados a objetos se puede considerar como un mecanismo de subtipo, en el mismo sentido de compartir operaciones.

6.3.4 Arreglos y funciones.

El conjunto de todas las funciones f:U→V puede dar lugar a un nuevo tipo de dos formas: como un tipo arreglo o como un tipo de función. Cuando U es un ordinal, la función puede considerarse como un arreglo con un tipo de índice U, y tipo de componente V. En C, C++ y Java el conjunto de índices siempre es un rango de enteros positivos que comienzan por 0.

Arreglo: Un arreglo es una colección **ordenada** de elementos de un mismo tipo de datos, agrupados de forma consecutiva en memoria. Cada elemento del arreglo tiene asociado un **índice**, que no es más que un número natural que lo identifica inequívocamente y permite al programador acceder a él.

Los arreglos pueden definirse con o sin tamaño, pero para definir una variable de tipo arreglo hay que asignarle un tamaño ya que los arreglos son asignados estáticamente o en la pila.

Cualquier intento de definir dinámicamente el tamaño de un arreglo es incorrecto tanto en C como en C++.

C permite que arreglos sin tamaño sean pasados como parámetros de función (parámetros que son básicamente apuntadores). Normalmente se le pasará como parámetro a parte un número que indique el tamaño del arreglo.

A diferencia de C, Java si puede asignar arreglos de forma dinámica, en el montón, y su tamaño puede especificarse en forma totalmente dinámica.

Los arreglos probablemente son los constructores más utilizados ya que su implementación puede hacerse en forma muy eficiente.

Los lenguajes funcionales por lo general no contienen un tipo arreglo ya que estos están pensados para la programación imperativa. Usualmente los lenguajes funcionales utilizan listas en vez de arreglos.

6.3.5 Tipos apuntadores y recursivos.

Un constructor de tipos que no corresponde a una operación de conjuntos es un constructor de referencias o apuntadores, que construye el conjunto de todas las direcciones que se refieren a un tipo especificado.

En C, **typedef int* IntPtr**; construye el tipo de todas las direcciones en que haya enteros almacenados.

Los apuntadores están implícitos en lenguajes que tienen una gestión automática de memoria. Esto es el caso de java para el cual todos los objetos son apuntadores implícitos que se asignan de forma explícita (new).

En ocasiones los lenguajes hacen distinción entre referencias y apuntadores, definiendo como referencia la dirección de un objeto bajo el control del sistema, que no se puede utilizar como valor ni operar de forma alguna, en tanto que un apuntador puede utilizarse como un valor y el programador puede manipularlo.

Tal vez C++ sea el único lenguaje donde coexisten apuntadores y referencias.

En C++, los tipos de referencia se crean con un ***operador postfijo* &**, lo cual no debe confundirse con el operador prefijo de dirección &, que devuelve el valor del puntero.

En C y C++ los arreglos son implícitamente apuntadores constantes hacia el primero de sus componentes.

Los apuntadores son de gran utilidad en la creación de tipos recursivos: un tipo que se utiliza a sí mismo en su declaración. Estos tienen una gran importancia en las estructuras de datos y algoritmos, ya que corresponden naturalmente a los algoritmos recursivos y representan datos cuya estructura y tamaño no se conocen de antemano. Dos ejemplos típicos son las listas y los árboles.

```
Struct CharList
{    char data;
     Struct CharList next;        ¡Incorrecto en C!
};
```

La estrategia de asignación de datos en C requiere que todos los tipos de datos tengan un tamaño fijo, el cual se determina en tiempo de traducción.

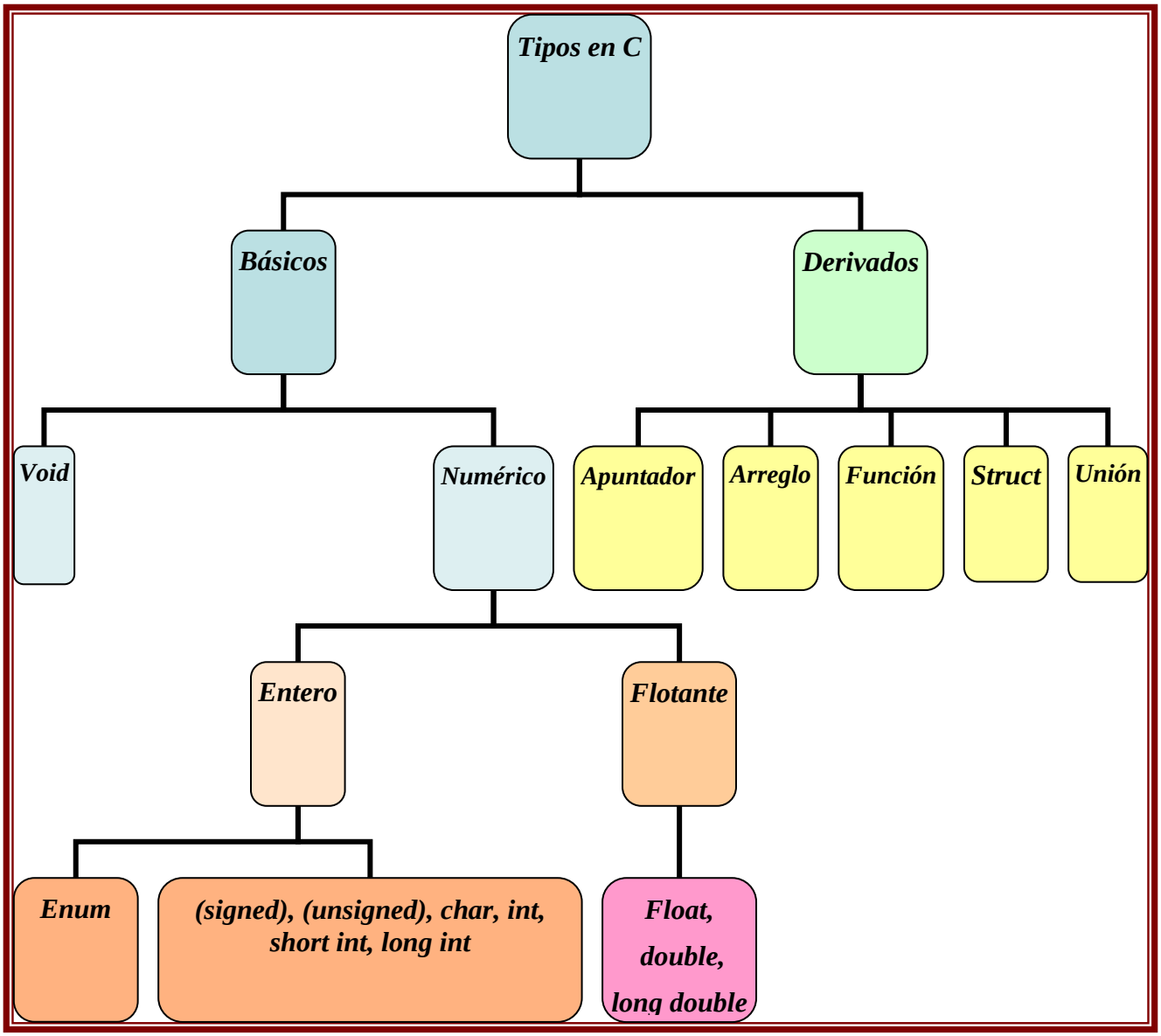
En C está prohibido el uso directo de la recursión en declaraciones de tipo, pero se permiten declaraciones recursivas indirectas por medio de apuntadores.

```
Struct CharListNode
{     char data;
      Struct CharListNode* next;          ¡Ahora es Correcto en C!
};
Typedef struct CharListNode* CharList;
```

6.4 Nomenclaturas de tipos en lenguajes de ejemplo.

C:

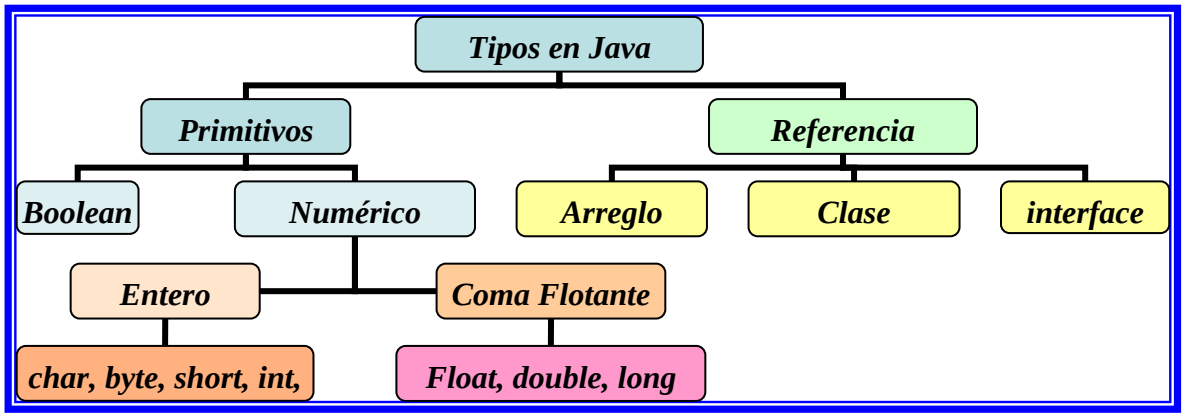
Los tipos simples se conocen como ***tipos básicos***, y los que se construyen mediante constructores de tipos se conocen como ***derivados***. Los tipos básicos incluyen el ***void*** (cuyo conjunto de valores está vacío) y los tipos ***numéricos***: los tipos ***enteros*** (ordinales) y los tipos ***flotantes***.



Java:

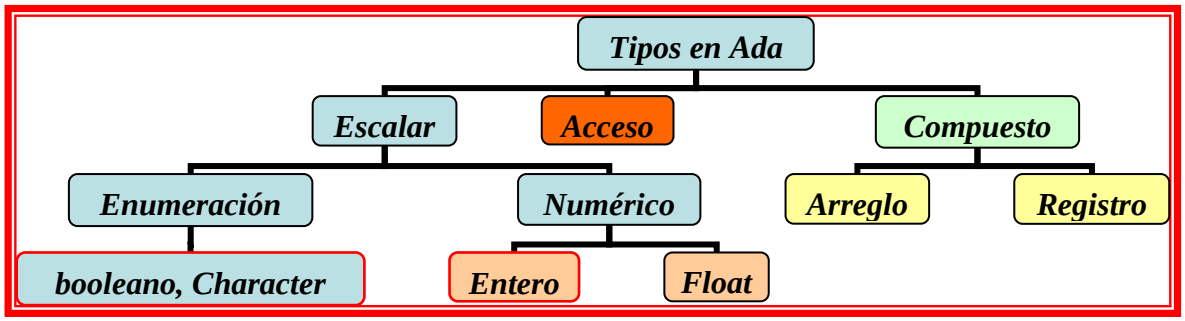
Los tipos simples se llaman tipos ***primitivos***, y los que se construyen utilizando constructores de tipos se llaman tipos de ***referencia***.

Los tipos primitivos se dividen en el tipo ***bolean*** y tipos de ***coma flotante*** (cinco enteros y dos en coma flotante). Sólo existen tres constructores de tipos: el ***arreglo***, la ***clase*** y la ***interface***.



Ada:

Los tipos simples se llaman ***tipos escalares***. Los tipos ordinales se llaman ***discretos***, los tipos ***numéricos*** comprenden los tipos ***reales*** y ***enteros***. Los tipos apuntador se llaman tipos ***access***. Los tipos de arreglo y de registro se llaman ***tipos compuestos***.



6.5 Equivalencia de tipos.

¿En que casos dos tipos son iguales? Una manera de responder a esta pregunta es comparar los conjuntos de valores simplemente como conjuntos. Dos conjuntos son iguales si contienen los mismos valores.

Por ejemplo: si tenemos el conjunto A x B, si el conjunto B no está definido de la misma manera que el tipo A, entonces A x B no es igual a B x A, ya que A x B contiene los pares (a, b), pero B x A contiene los pares (b, a).

Dos de las formas más habituales de equivalencia de tipos en los lenguajes de programación actuales son: la equivalencia estructural y la equivalencia de nombre.

La ***equivalencia estructural*** viene a decir que dos tipos son iguales si tienen la misma estructura: están construidos de la misma forma a partir de los mismos tipos simples.

Para la verificación de tipos estructuralmente, un intérprete puede representar dicho tipo como un árbol y verificar la equivalencia recursivamente en subárboles.

Typedef A1[10]	int	Son equivalentes estructuralmente si el tamaño del conjunto índice no es parte del tipo arreglo.
Typedef A1[20]	int	

```
Struct RecA          Struct RecB
{  char x;           {  char a;
   Int y;               Int b;
};                   };
```

Deberían ser estructuralmente equivalentes, sin embargo, no lo son, ya que las variables de las diferentes estructuras tendrían que usar diferentes nombres para acceder a los datos miembros.

Un factor que complica las cosas es el uso de nombres en las declaraciones.

Para determinar la equivalencia estructural en presencia de nombres basta con reemplazar en la declaración cada nombre por la expresión asociada a su tipo. En el caso de tipos recursivos, esta regla llevaría a un ciclo infinito.

Las razones principales para incluir nombres de tipos es permitir la declaración de tipos recursivos.

La ***equivalencia de nombres*** se refiere a que dos tipos son iguales sólo si tienen el mismo nombre.

Dos tipos son iguales sólo si tienen el mismo nombre, y dos variables son equivalentes en tipo sólo si sus declaraciones usan exactamente el mismo nombre de tipo.

La equivalencia de nombres en su estado más puro es incluso más fácil de implementar que la estructural, siempre y cuando estemos obligados a dar nombre a todos los tipos. Ada es un lenguaje que ha implementado una equivalencia de nombres muy pura.

C tiene una equivalencia que está entre la estructural y la de nombres y se puede decir que tiene una equivalencia de nombre para structs y uniones, y estructural para todo lo demás.

En Pascal, los constructores de tipos llevan a tipos nuevos no equivalentes. Los nuevos nombres para los nombres de tipos existentes son equivalentes a los tipos originales.

```
Type
   IntPtr = ^Integer;
   Age = Integer;
Var
   X: IntPtr;
   Y: ^Integer;
   I: Age;
   A: Integer;
```

X e Y NO son equivalentes en Pascal, pero I y A SI lo son.

Java tiene un método relativamente simple para la equivalencia de tipos. Primero no existe typedef, por lo que se minimizan los problemas con nombres. Segundo las declaraciones Class e interface crean implícitamente nuevos nombres de tipos y para estos tipos se utiliza la equivalencia de nombres. La única complicación es que los arreglos emplean equivalencia estructural con reglas especiales para establecer la equivalencia del tipo base.

ML tiene dos declaraciones de tipos (type y datatype), pero sólo el segundo construye un nuevo tipo, en tanto que el primero sólo construye un alias de tipos existentes.

Type Age = Int;	Age es equivalente a int, pero
Datataype NewAge = NewAge int;	NewAge es un nuevo tipo no equivalente a Int.

Ejemplo:

Diga qué variables son equivalentes en tipos según **(a)** equivalencia estructural, **(b)** equivalencia de nombres y **(c)** algoritmo de equivalencia de C.

```
Typedef struct
{   int X;
    Char Y;
} Rec1;
Typedef Rec1 Rec2;
Typedef struct
{   int X;
    Char Y;
} Rec3;
Rec1 a, b;
Rec2 c;
Rec3 d;
```

Equivalencia Estructural:

Rec1 y Rec3 son equivalentes estructuralmente, por tanto **a, b** y **d** serán equivalentes.

Rec2 está definida como del tipo de Rec1, y por tanto también será equivalente estructuralmente.

Equivalencia de nombres:

Las variables **a** y **b** están declaradas del tipo Rec1 y por tanto serán equivalentes en nombre.

Aunque Rec2 sea del tipo de Rec1, no es equivalente en nombre.

Algoritmo de Equivalencia de C:

La equivalencia de C se basa en que los arreglos y structs se comprueba la equivalencia en nombres, y para el resto de variables se utiliza la equivalencia estructural.

Rec2 es del tipo de Rec1, refiriéndose ambos al mismo struct. Las variables **a**, **b** y **c** se refieren al mismo struct y por tanto son equivalentes.

La variable **d** está definida del tipo Rec3 que corresponde a un struct diferente y por ello no es equivalente a las otras variables.

6.6 Verificación de tipos.

La verificación de tipos es el proceso que sigue el intérprete para verificar que todas las construcciones en un programa tengan sentido en términos de los tipos de constantes, variables, procedimientos y otras entidades.

Involucra al algoritmo de verificación de tipos para expresiones y asignaciones, y este puede modificar el algoritmo de equivalencias al contexto.

La verificación de datos puede dividirse en ***dinámica*** o ***estática***:

La verificación es dinámica si la información de tipos se conserva y se verifica en tiempo de ejecución. Por definición los intérpretes realizan verificación dinámica de los tipos. La verificación dinámica de los tipos es necesaria cuando los tipos sólo pueden determinarse en tiempo de ejecución.

En la verificación estática, los tipos de expresiones y de datos se extraen del texto del programa y el intérprete lleva a cabo la comprobación antes de la ejecución, por lo que estos lenguajes deben de tener tipificado estático.

Ejemplo 1: Los compiladores de C efectúan una verificación estática durante la traducción, pero realmente C no es un lenguaje con tipificado fuerte y muchas inconsistencias en los tipos no causan errores de compilación.

Ejemplo 2: El dialecto Scheme de Lisp es un lenguaje con tipificado dinámico, pero los tipos se verifican en forma rigurosa: Todos los errores de tipo provocan la terminación del programa.

Ejemplo 3: Ada es un lenguaje con tipificado fuerte y todos los errores de tipo generan mensaje de error en la compilación, pero sin embargo, incluso en Ada, ciertos errores, como los de rango en subíndice de arreglos, no pueden detectarse antes de la ejecución.

Una parte esencial en la verificación de tipos es la inferencia de tipos, en la que los tipos de las expresiones se determinan a través de las subexpresiones que la componen.

La inferencia de tipos y las reglas de corrección a menudo son las partes más complejas en la semántica de un lenguaje.

Los principales temas y problemas en las reglas de un sistema de tipos son:

6.6.1 Compatibilidad de tipos.

A menudo es necesario relajar las reglas de corrección de tipos de manera que los tipos de dos componentes no sean precisamente iguales, según el algoritmo de equivalencia de tipos. Dos tipos diferentes que incluso pueden ser correctos cuando se combinan en cierta forma se conocen como tipos compatibles.

Un término relacionado es la ***compatibilidad de asignación***, a menudo se utiliza para la corrección de tipos de la asignación e1 = e2. El lado izquierdo debe ser un valor o referencia **l**, y el lado derecho debe ser un valor **r**.

Igual que para la compatibilidad ordinaria, la compatibilidad de asignación puede ampliarse para casos en los que ambos lados son de diferente tipo.

6.6.2 Tipos implícitos.

Los tipos de las entidades básicas como las constantes o las variables pueden no establecerse explícitamente en una declaración. En estos casos el intérprete debe de inferir el tipo ya sea a través del contexto o a partir de alguna regla estándar. Estos tipos se conocen como implícitos.

En C, las variables son implícitamente enteros, si no se les declara ningún tipo.

En todos los lenguajes los literales son el ejemplo más claro de entidades tipificadas implícitamente.

6.6.3 Tipos que se superponen y valores de tipos múltiples.

Dos tipos pueden superponerse cuando contienen valores en común.

Subtipos y Subrangos pueden crear tipos cuyos conjuntos de valores se superponen. Normalmente es preferible que los tipos sean conjuntos no conexos, de forma que no surjan ambigüedades.

Es difícil evitar la superposición de valores.

Por ejemplo, en C los dos tipos **unsigned int** e **int** contienen una superposición sustancial. En Java, un entero pequeño pudiera ser un **short**, un **int** o un **long**.

6.6.4 Operaciones compartidas.

Los tipos tienen operaciones que usualmente están definidas de forma implícita. A menudo estas operaciones son compartidas entre varios tipos, o tienen el mismo nombre que otras operaciones que pueden ser diferentes. Se considera que estos operadores están ***sobrecargados***. En este caso el intérprete debe decidir a partir de los tipos de los operandos, a que operación se refiere.

6.7 Conversión de los tipos.

En todos los lenguajes de programación actuales, existe la necesidad bajo ciertas circunstancias de convertir un tipo a otro. Esta conversión de tipos puede incluirse en el sistema, de forma que las conversiones se lleven acabo de forma automática:

```
Int X = 3;
….
X= 2.3 + X / 2;
```

Al final de este código **X** sigue valiendo 3, ya que 3.3 es truncado por el proceso de conversión. En este ejemplo el intérprete ejecutó dos conversiones implícitas a veces conocidas como coacciones. La conversión de **int** a **doble** es de ***extensión***, mientras que al revés sería de ***restricción***.

La conversión ***implícita*** puede debilitar la verificación de tipos de forma que no se detecten errores.

La conversión ***explícita*** se presenta cuando las directrices de la conversión se escriben directamente en el código también llamada como ***conversión forzada***:

```
X = (int) (2.3 + (double) (x / 2);
```

Otra variante de conversión forzada es la sintaxis del llamado de funciones, con el tipo resultando como nombre de la función y el valor que se desea convertir como argumento:

```
X = int( 2.3 + double( X/2) );
```

Las ventajas de utilizar conversiones forzadas es que se documentan en forma precisa dentro del código.

Una alternativa a las conversiones forzadas es tener funciones predefinidas o de biblioteca, que lleven a cabo dichas conversiones.

Por ejemplo, en Java la clase **Integer** en la biblioteca **java.lang** contiene las funciones de conversión **toString**, que convierte un int en un string, y **parseInt** que convierte un string en un int.

En algunos lenguajes está prohibida la conversión implícita a favor de la explícita, la cual favorece la documentación de la conversión minimizando los riesgos de comportamientos extraños y facilita la sobrecarga. Como ejemplo de estos lenguajes tenemos a Ada. Un paso intermedio es permitir la conversión implícita siempre que no involucre corrupción de los datos, en este sentido Java sólo permite conversión por extensión.

Los lenguajes orientados a objetos tienen requerimientos especiales para la conversión, ya que la herencia puede interpretarse como un mecanismo de subtipificación y en algunos casos es necesario hacer conversiones de subtipos a supertipos y viceversa.

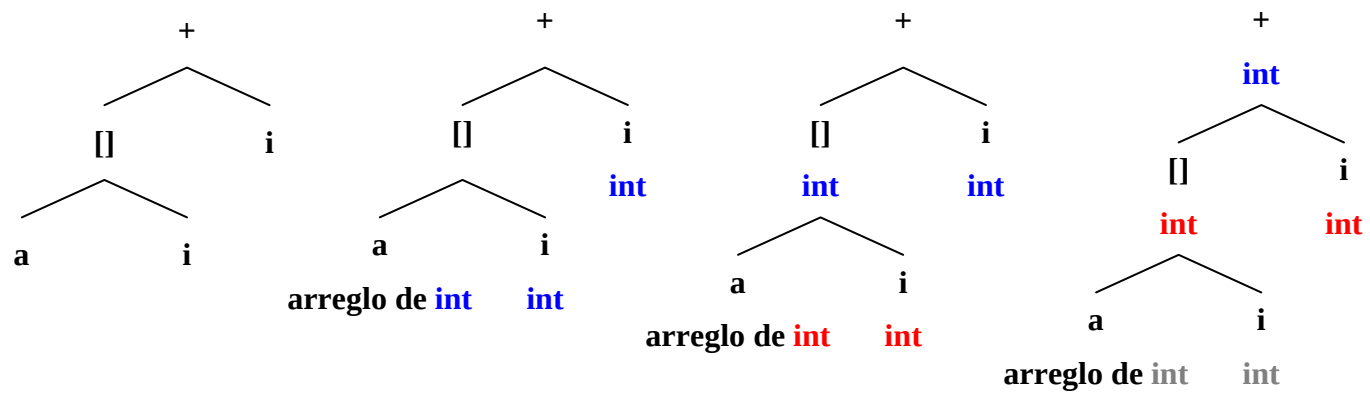
6.8 Verificación de tipos polimórficos.

La mayoría de los lenguajes de tipificado estático exigen que en todos los nombres de cada declaración se dé información sobre los tipos.

También es posible aplicar una forma de inferencia de tipos para determinar los tipos de los nombres en una declaración en la que no se hayan dado dichos tipos. Esta forma es la verificación de tipos **Hindley-Milner**:

- Se confeccionan los árboles sintácticos de la expresión.
- Primero se llenan los tipos de las hojas a partir de las declaraciones. Si no se conocen los tipos, se asignan tipos simulados.
- A partir de ellas, el verificador de tipos va ascendiendo por las ramas asignando y verificando que los tipos sean correctos.

Por ejemplo: **a[i] + i**



Una vez que una variable de tipo es reemplazada por un tipo real, todas las instancias de la variable deben actualizarse. Este proceso se conoce como ***instanciamiento*** de las variables tipo.

Cuando se dispone de una información sobre los tipos, las expresiones de tipos para las variables pueden cambiar de forma en diversas maneras. Este proceso se conoce como ***unificación***.

La unificación involucra tres casos:

1. Cualquier variable de tipo se unifica con cualquier expresión de tipos (y es instanciado según esa expresión)
2. Dos constantes de tipo (como **int** o **double**) sólo se unifican si son del mismo tipo.
3. Dos construcciones de tipo (como el **arreglo** o el **struct**) sólo se unifican si son aplicaciones del mismo constructor de tipo y todos sus tipos componentes también se unifican.

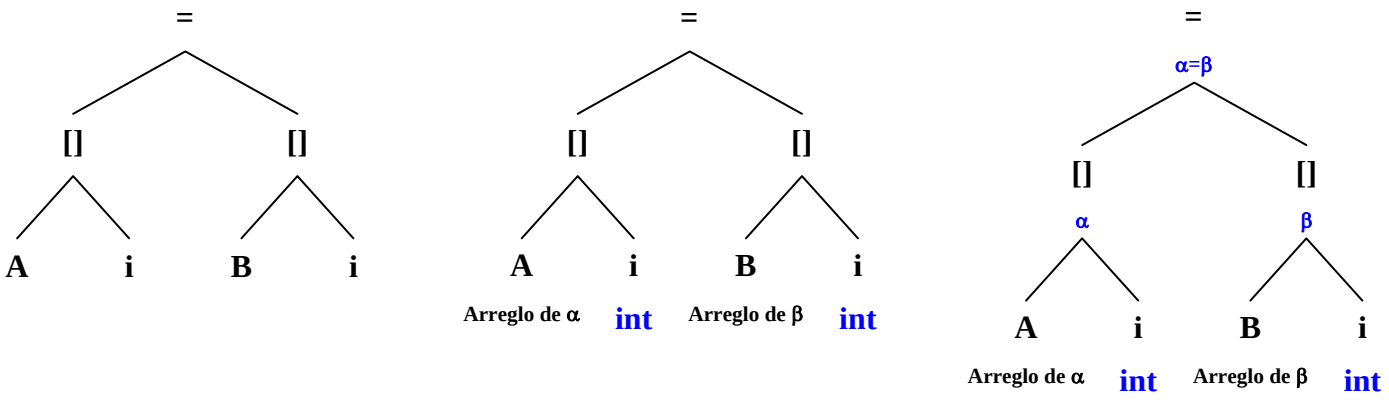
Por ejemplo, al unificar la variable de tipo β con la expresión de tipo “arreglo de α” se produce el caso 1, y β es inicializado a “arreglo de α”. Unificar **int** a **int** es un ejemplo del caso 2. Al unificar “arreglo de α” con “arreglo de β” se produce el caso 3.

La verificación de tipos Hindley-Milner aporta una ventaja en la verificación de tipos simples, ya que los tipos pueden conservarse tan generales como sea posible, a la vez que se verifica su consistencia de forma estricta.

Ejemplo: **A[i] = B[i]**

Esta expresión asigna el valor de **B[i]** a **A[i]**. La verificación de Hindley-Milner establecerá que **i** deberá ser un **int**, A deberá ser un “arreglo de α” y B debe ser un “arreglo de β”, y luego α= =β.

La verificación de tipos concluye considerando los tipos de A y B como restringidos a “arreglo de α”, pero α sigue siendo una variable irrestricta, que podría ser de cualquier tipo. Se dice que esta clase de expresiones son ***Polimórficas***.



Polimórfico significa "***de muchas formas***". En los lenguajes de programación se aplica a nombres que pueden tener más de un tipo al mismo tiempo.

La sobrecarga es una clase de polimorfismo.

El tipo "arreglo de α" es en realidad un conjunto de tipos ***múltiples e infinitos***, dependiendo de la variable α. Esta clase de polimorfismo se conoce como ***polimorfismo paramétrico***, ya que α es un parámetro de tipo que puede ser reemplazado por cualquier expresión de tipo.

Polimorfismo paramétrico implícito, los parámetros de tipo son introducidos implícitamente por el verificador de tipos.

Polimorfismo paramétrico explícito, los parámetros de tipo son introducidos explícitamente por el programador.

Todo objeto de tipo polimórfico que se pasa a una función como parámetro debe tener una especialización fija a lo largo de toda la función.

Para que un intérprete lleve a cabo la traducción de código con tipos polimórficos se puede realizar de dos formas:

1. ***Expansión***: Se examinan todas las llamadas a la función y se genera una copia del código para cada uno de los tipos empleados usando el tamaño apropiado para cada tipo.
2. ***Encuadrado y Etiquetado***: Se fija un tamaño para todos los datos que pueden contener valores escalares, se agrega un campo de bits que etiqueta el tipo como escalar o no, y un campo de tamaño si no son escalares. Todos los tipos estructurados quedan representados por un apuntador y un tamaño, mientras que los tipos escalares se copian directamente.

6.9 Polimorfismo explícito.

El polimorfismo paramétrico implícito está bien para definir funciones polimórficas, pero no nos ayuda si deseamos definir estructuras de datos polimórficos.

Por ejemplo, en la siguiente declaración es imposible establecer el tipo implícito, por lo que se tendrá que escribir explícitamente el tipo que se desea.

Implícito	Explícito
Typedef struct StackNode	**Typedef struct StackNode**
{ ?? data:	**{ int data:**
Struct StackNode * Next;	**Struct StackNode * Next;**
}* Stack	**}* Stack**

C++ es un lenguaje con polimorfismo paramétrico explícito, pero sin tener asociada la verificación de tipos Standley-Milner implícita. El mecanismo que utiliza es la ***plantilla***, que puede usarse ya sea con funciones o con constructores de tipos ***Class*** o ***Struct***.

Tema 7 Control 1, Expresiones y Enunciados

Las expresiones representan el mecanismo fundamental de cómputo de los programas.

Una ***expresión*** en su forma más pura devuelve un valor y no produce efectos colaterales, es decir, no hay cambios en la memoria del programa. Son las partes del lenguaje de programación que más se parecen a las matemáticas.

Un ***enunciado*** se ejecuta por sus efectos colaterales y no devuelve ningún valor.

La forma en la que se evalúan expresiones, incluyendo el orden en el cual las subexpresiones se calculan, puede tener un efecto importante sobre su significado, algo que en ningún caso es aplicable a las expresiones matemáticas.

7.1 Expresiones.

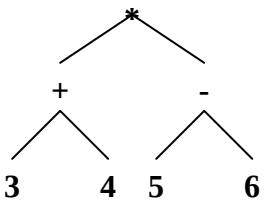
Las expresiones básicas son las literales (constantes) y los identificadores.

Expresiones más complejas se elaboran a partir de expresiones básicas mediante la aplicación de operadores y funciones.

Un ***operador unario*** es aquel que toma un operando solamente y el ***operador binario*** es el que toma dos.

Los operadores pueden escribirse de forma ***infija***, ***postfija*** o ***prefija***.

Todos los lenguajes tienen reglas para evaluar las expresiones. Una regla común es que todos los operandos son evaluados primero y después se les aplica los operadores. A esto se le conoce como ***evaluación de orden aplicativo***, o también como evaluación ***estricta***. Corresponde a una evaluación de abajo hacia arriba de los valores en los nodos del árbol sintáctico.



En el siguiente árbol sintáctico, primero se evalúan los nodos + y – a 7 y -1 respectivamente, y luego el operando * a -7.

Por ejemplo: ***Orden aplicativo***:

> Con una función como **mul(add(3, 4), sub(5, 6))**, se evalúan primero los argumentos **add(3, 4)** y **sub(5, 6),** se reemplazan por los valores obtenidos y se evalúa **mul(7, -1)**.
>
> **mul(add(3, 4), sub(5, 6))= mul(7, -1) = 7 * -1 = -7**

Si la evaluación de una expresión no provoca efectos colaterales, ésta dará el mismo resultado, independientemente del orden en que se evalúen las subexpresiones. En presencia de efectos colaterales, el orden en el que se realice la evaluación puede provocar diferencias.

Un ***operador de secuencia*** permite que se combinen varias expresiones en una sola y que se evalúen secuencialmente.

En un lenguaje de programación se puede especificar que las expresiones booleanas deben evaluarse en orden de izquierda a derecha, hasta el punto en que se conoce el valor verdadero de toda la expresión, y en ese momento la evaluación se detiene. Esto se conoce como una ***evaluación de corto circuito***.

Las expresiones booleanas no son las únicas cuyas subexpresiones pueden no evaluarse todas:

- Las expresiones *if* **nunca** evalúan todas sus subexpresiones: ***if*** **e1** (*expresión de prueba*) ***entonces*** **e2** (sentencias) ***sino*** **e3** (sentencias alternativas). Las expresiones *if* siempre evalúan primero **e1** (la expresión de prueba), pero con base en este valor, sólo se evalúa **e2** o **e3**. En ausencia de efectos colaterales, el orden de la evaluación de las expresiones no tiene importancia en relación con el valor final de la expresión.

Los lenguajes funcionales puros, carecen de efectos colaterales y las expresiones comparten la propiedad de ***transparencia referencial*** o ***regla de sustitución***: dos expresiones que tengan el mismo valor, pueden ser sustituidas la una por la otra en cualquier parte del programa. En otras palabras, sus valores siempre permanecen iguales.

Una forma más sólida de evaluación es la ***evaluación de orden normal*** (o evaluación ***perezosa*** en el lenguaje Haskell) en la que primero se evalúan las operaciones y luego se evalúan los operandos si éstos son necesarios para el cálculo.

Por ejemplo: ***Orden normal***.

Square(double(2)): Square se sustituye por **double(2)*double(2)** sin evaluar double(2), y después se reemplaza double(2) por 2 + 2.

Square(double(2))= **double(2)*double(2)** = **2+2 * 2+2 = 4 * 4 = 16**

Si hubiéramos seguido el orden aplicativo tendríamos:

Square(double(2))= **Square(2+2)** = **Square(4) = 4 * 4 = 16**

7.2 Enunciados y guardias condicionales.

Enunciados if:

El ***if cauteloso*** es aquél que tiene todas las expresiones booleanas posibles asociadas a secuencias de enunciados. Por ejemplo:

if Expresión 1 entonces secuencias 1,

Expresión 2 entonces secuencias 2,

…

Expresión n entonces secuencias n,

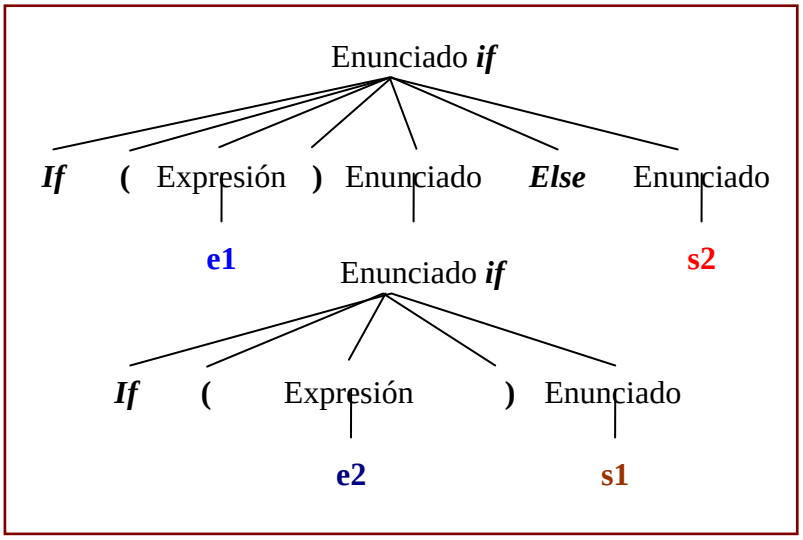
End if.

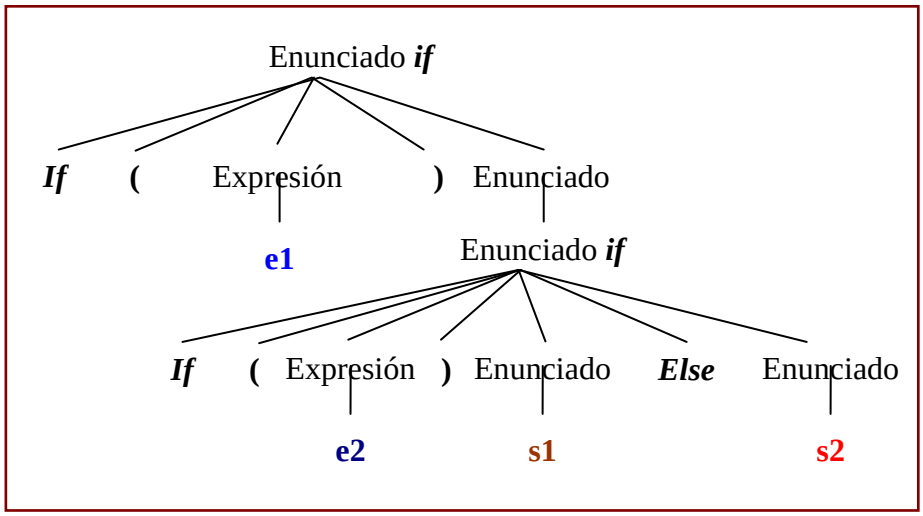
La forma básica del enunciado ***if*** es: *enunciado if* = if (expresión) enunciado, [else enunciado].

Esta forma del ***if*** es ambiguo, en el sentido sintáctico, por ejemplo: if e1 entonces if e2 entonces S1 sino S2.

Esta expresión tiene dos árboles de análisis gramatical diferentes.

Árboles gramaticales de la expresión: ***if*** (**e1**) ***if*** (**e2**) **s1** ***else*** **s2**





Esta ambigüedad se conoce como el problema del ***else ambiguo***: la sintaxis no nos indica si un else, después de dos enunciados if, debe asociarse con el primero o el segundo if.

El lenguaje C (al igual que Pascal), resuelve el problema mediante una regla para eliminar la ambigüedad: el else se asociará con el if anterior más cercano que no tenga ya una parte else. A esta regla también se le conoce como la ***regla de anidamiento más cercano***.

Existen otras maneras de resolver el problema del else ambiguo. Una forma es emplear una ***palabra clave enmarcadora***.

Enunciado if = if condición then secuencia de enunciados [else secuencia de enunciados] **end if**.

Las dos palabras clave enmarcadoras ***end if*** cierran el enunciado ***if*** y eliminan la ambigüedad.

Enunciados Case y Switch:

El enunciado ***case*** es una variación especial del if cauteloso, en el cual las guardias, en vez de ser expresiones booleanas, son valores ordinales seleccionados por una expresión ordinal.

La semántica de este enunciado es evaluar la expresión de control y transferir el control al punto del enunciado en donde está listado el valor.

7.3 Ciclos y variaciones sobre While.

El ***do cauteloso*** es de la forma:

Do Expresión 1 entonces secuencias 1,
Expresión 2 entonces secuencias 2,
…
Expresión n entonces secuencias n.
End do.

Este enunciado se repite hasta que todas las expresiones son falsas.

La forma más básica de la construcción de ciclo es el ciclo ***while***:

While **(expresión de prueba)** … secuencia de enunciados.

La expresión de prueba se evalúa primero. Si resulta verdadera o distinta de cero, entonces se ejecuta la secuencia de enunciados. Después vuelve a evaluarse la expresión de prueba y así sucesivamente.

Un enunciado alterno que asegura que el código del ciclo se ejecute por lo menos una vez es el ***do while***.

Do secuencia de enunciados ***while*** expresión de prueba.

Esta expresión del ***Do while*** se puede realizar mediante el siguiente código:

Secuencia de enunciados; ***while*** (expresión de prueba) secuencia de enunciados.

A la construcción de lenguaje que es completamente expresable en términos de otras construcciones se le llama "***azúcar sintáctico***".

Los iteradores se definen como funciones que devuelve un valor. Sin embargo, el efecto de la llamada a un iterador es:

La primera vez que se llama a un iterador, se guardan los valores de sus parámetros. El iterador entonces comienza a ejecutarse, hasta que llega a un enunciado ***yield***, donde suspende la ejecución y devuelve el valor de la expresión en el ***yield***. En llamadas subsecuentes continúa la ejecución después del ***yield***, suspendiéndose cada vez que llegue a otro ***yield***, hasta que sale, cuando el ciclo desde el cual se le llama también termina.

7.4 La controversia del GOTO.

Java tiene ***goto*** como palabra reservada, así que no puede utilizarse. Java incluye alternativas para el uso del ***goto***. Una situación como ésta es la devolución del control a un nivel de anidamiento externo.

Esto puede escribirse en Java usando un enunciado break etiquetado.

Los ***breaks etiquetados*** están severamente restringidos en Java: las etiquetas deben estar encerradas dentro de un bloque en el interior de la misma función; por lo tanto, no pueden utilizarse para saltar hacia delante o hacia una ubicación no local. Sólo pueden utilizarse para saltar hacia ***fuera*** localmente.

Java también tiene un enunciado ***continúe etiquetado*** con las mismas restricciones.

7.5 Manejo de excepciones.

Todos los mecanismos de control que hemos estudiado han sido ***explícitos***: en un punto en el que ocurre una transferencia de control, hay una indicación sintáctica de la transferencia.

Existen situaciones en donde la transferencia del control es ***implícita***. En el punto donde realmente ocurre la transferencia puede no haber ninguna indicación sintáctica.

Esta situación es el ***manejo de excepciones***: el control de condiciones de error y otros eventos no usuales durante la ejecución de un programa.

Una excepción es cualquier evento inesperado o poco frecuente.

Las excepciones no sólo se limitan a los errores: una excepción puede ser cualquier evento no usual, como el fallo en la entrada de datos y una pausa.

Un ***manejador de excepciones*** es un procedimiento o secuencia de código diseñado para ejecutarse cuando se pone de manifiesto una excepción.

Los lenguajes que no cuentan con mecanismos incorporados a veces tienen bibliotecas que los proporcionan, o cuentan con otras formas incorporadas para simularlos.

El manejo de excepciones intenta imitar en el lenguaje de programación las características de una interrupción de hardware mediante las cuales el procesador transfiere el control automáticamente a una ubicación por adelantado, de acuerdo con el tipo de error o interrupción.

Si un programa se ***aborta*** cuando se produce una excepción, dichos programas no pasan las pruebas de robustez. Un programa debe ser capaz de recuperarse de errores y continuar con la ejecución.

Los fallos de un dispositivo de hardware, problemas de asignación de memoria, problemas de comunicación, etcétera, pueden llevarnos a situaciones en las que el sistema operativo se vea obligado a tomar medidas drásticas para terminar el programa sin que éste pueda hacer algo al respecto. A estos errores se les conoce como ***excepciones asíncronas***, ya que pueden ocurrir en cualquier momento.

Los errores que el programa puede atrapar se conocen como ***excepciones síncronas***: aquellas excepciones que ocurren en respuesta directa a las acciones del programa.

Las excepciones definidas por el usuario sólo pueden ser síncronas, pero las excepciones predefinidas o de biblioteca pueden incluir varias excepciones asíncronas, ya que el entorno de ejecución del programa puede cooperar con el sistema operativo para permitir al programa atrapar algunos errores asíncronos.

Las pruebas explícitas de excepciones hacen que un programa sea más difícil de escribir, ya que el programador debe someter a prueba por adelantado todas las condiciones de excepción posibles.

Para diseñar este tipo de comportamiento, deben considerarse los siguientes puntos:

> **Excepciones**: ¿Cuáles son las excepciones predefinidas por el lenguaje?, ¿pueden desactivarse?, ¿pueden crearse excepciones definidas por el usuario?, ¿cuál es su alcance?.
>
> **Manejador de excepciones**: ¿Cómo se definen?, ¿cuál es su alcance?, ¿cuáles son los manejadores de excepción por omisión suministrados para excepciones predefinidas?, ¿pueden ser reemplazados?.
>
> **Control**: ¿Cómo se pasa el control de un manejador?, ¿a dónde se transfiere el control después de ejecutar el manejador?, ¿qué entorno de ejecución queda después de que se haya ejecutado un manejador de excepciones?.

Excepciones:

La mayoría de los lenguajes proveen valores o tipos de excepción predefinidos, por ejemplo:

Constraint error: ocurre cuando se exceden los límites de un subrango o arreglo o cuando hay desbordamiento aritmético y división por cero.

Program error: ocurren durante el procesamiento dinámico de una declaración.

Storage error: cuando falla la asignación de memoria.

Tasking error: durante el control de concurrencia.

Control de excepciones:

Una excepción predefinida o incorporada puede ser puesta de manifiesto, ya sea automáticamente por el sistema en tiempo de ejecución o bien manualmente por el programa. Las excepciones definidas por el usuario sólo pueden ser puestas de manifiesto manualmente por el programa.

Cuando se pone de manifiesto una excepción, por lo general se abandona el cálculo que se está haciendo, y el sistema en tiempo de ejecución comienza a buscar un manejador.

Esta búsqueda comienza en la sección para el manejador del bloque en el cual se atrapó la excepción. Si el programa no encuentra un manejador, se consulta la sección de manejador en el siguiente bloque, y así sucesivamente. Este proceso se conoce como ***propagación de la excepción***.

Si se llega al último bloque sin haber encontrado un manejador, la llamada se devuelve al invocador y la excepción se vuelve a poner de manifiesto en él.

El proceso de salirse de vuelta a través de las llamadas de función a los invocadores, se conoce como ***desenrollo de llamada*** o ***de pila***.

Una vez encontrado y ejecutado el manejador, todavía queda la cuestión respecto de dónde se continuará la ejecución. Una opción es regresar al punto en el que por primera vez se puso de manifiesto la excepción y comenzar la ejecución a partir de ese enunciado o expresión. Esto se conoce como el ***modelo de reanudación***. Este modelo requiere que, durante la búsqueda del manejador se conserve el entorno de la estructura de las llamadas originales y que se restablezcan antes de reanudar la ejecución.

La alternativa al modelo de reanudación es continuar la ejecución en el código inmediato siguiente al bloque o expresión en la que se encontró el manejador que fue ejecutado. Esto se conoce como ***modelo de terminación*** del manejo de excepciones.

Casi todos los lenguajes modernos utilizan el modelo de terminación para el manejo de excepciones. La terminación es más fácil de implementar que la reanudación.

El manejo de excepciones a menudo implica una carga en tiempo de ejecución. Por esta razón, es mejor evitar el uso excesivo de expresiones para implementar situaciones de control ordinarias, que podrían ser reemplazadas por pruebas simples.

Especificación de excepciones:

Una especificación de excepción es una lista de excepciones que se añade a la declaración de una función para garantizar que ésta sólo devuelva las excepciones que están en la lista y ninguna otra más.

El compilador puede utilizar la especificación de excepciones de dos formas: Primero, puede llevar a cabo ciertas optimizaciones que hacen más eficiente el manejo de excepciones. Segundo, puede volver a escribir el código de la función de tal forma que ***no pueda*** arrojar ninguna excepción diferente a las que aparecen en la lista.

El reporte de errores no es para lo único que sirve el manejo de errores; también sirve para generar información de depuración sobre el estado del programa cuando se ha encontrado un error. Una información que resulta muy útil es el ***rastro de la llamada*** (o rastro de la pila), que indica los procedimientos a los que se llamó cuando ocurrió el error.

Tema 8 Control 2, Procedimientos y Ambientes

Las funciones deben producir un único valor y no tener efectos colaterales, en tanto que los procedimientos no producen valores y operan con base en producir efectos colaterales.

Las llamadas a procedimientos son enunciados, en tanto que las llamadas a funciones son expresiones.

Los procedimientos surgieron como una forma de dividir un programa en pequeñas partes compiladas.

Los procedimientos se utilizan como un mecanismo para abstraer un grupo de operaciones relacionadas entre sí en una sola operación que puede ser utilizada de manera repetida sin necesidad de repetir el código.

8.1 Definición y activación de procedimientos.

Un ***procedimiento*** es un mecanismo en un lenguaje de programación para abstraer un grupo de acciones o de computaciones. El grupo de acciones se conoce como el ***cuerpo*** del procedimiento, que está representado por el nombre del procedimiento. Se define un procedimiento al proveer una especificación o interfaz y un cuerpo. La especificación le da nombre al procedimiento, una lista de los tipos y nombres de sus parámetros, así como el tipo de su valor devuelto si es que existe alguno.

Se llama o activa un procedimiento al enunciar su nombre, junto con los argumentos de la llamada, que corresponden a sus parámetros.

Una llamada a un procedimiento transfiere el control al principio del procedimiento llamado. Cuando la ejecución llega al final del cuerpo, el control es devuelto al llamador. En algunos lenguajes, puede devolverse el control al llamador antes de llegar al final del cuerpo utilizando un enunciado return.

Un procedimiento se comunica con el resto del programa a través de sus parámetros y también a través de sus referencias no locales.

8.2 Semántica de los procedimientos.

Un procedimiento es un bloque cuya declaración está separada de su ejecución.

El ambiente determina la asignación de la memoria y mantiene el significado de los nombres durante la ejecución.

En un lenguaje estructurado en bloques, cuando se encuentra un bloque durante la ejecución, hace que ocurra la asignación de variables locales y otros objetos correspondientes a las declaraciones del bloque. Esta memoria asignada se conoce como ***registro de activación*** o ***marco de pila***.

Para bloques que no sean procedimientos, el ambiente definidor y el ambiente invocador son siempre el mismo. Por el contrario, un procedimiento tiene ambientes invocador y de definición distintos.

El método de comunicación de un procedimiento con su ambiente invocador es a través de ***parámetros***. Se declara una lista de parámetros junto con la definición del procedimiento.

Los parámetros no asumen valores hasta que son reemplazados por los argumentos. Los parámetros a veces se conocen como parámetros formales, en tanto que los argumentos se llaman parámetros actuales.

Un procedimiento o una función no deberán nunca utilizar o cambiar una variable no local. La razón es que el uso o cambio representa una dependencia (en el caso de uso) o un efecto colateral (en el caso de cambio), invisible en una lectura de la especificación del procedimiento.

Algunos procedimientos pueden depender solamente de parámetros y de características fijas del lenguaje, y estos procedimientos se dice que están en una ***forma cerrada***, ya que no contienen dependencias no locales.

Definimos como ***Cerradura***, al código de la función junto con la representación del ambiente de definición de la misma.

8.3 Mecanismos de paso de parámetros.

Los más importantes mecanismos de paso de parámetros son 4: paso por valor, paso por referencia, paso por valor resultado y paso por nombre.

Paso por valor:

En este mecanismo, los argumentos son expresiones que se evalúan en el momento de la llamada y, sus valores se convierten en los valores de los parámetros durante la ejecución del procedimiento. Los parámetros de valor se comportan como valores constantes durante la ejecución del procedimiento.

Paso por referencia:

En vez de pasar el valor de la variable, el paso por referencia pasa la ubicación de la variable de modo que el parámetro se convierte en un ***alias*** para el argumento, y cualquier cambio que se le haga a éste lo sufre también el argumento.

Paso por valor resultado:

Es similar al del paso por referencia, excepto que en este caso no se establece un alias real. En el procedimiento, el valor del argumento es copiado y utilizado, y después, cuando el procedimiento termina, el valor final del parámetro se copia de regreso a la ubicación del argumento.

Este método es a veces conocido como ***copia al entrar***, ***copia al salir***, o como ***copia restauración***.

El paso por valor resultado sólo se distingue del paso por referencia por la presencia del alias.

Paso por nombre y evaluación retardada:

La idea del paso por nombre es que no se evalúa el argumento hasta su uso real en el procedimiento llamado. Por lo que el *nombre* del argumento reemplaza el nombre del parámetro al cual corresponde.

El argumento se evaluará siempre en el ambiente del invocador, en tanto que el procedimiento se ejecutará en su ambiente de definición.

El argumento es evaluado cada vez que se encuentra el parámetro y en el interior del procedimiento llamado. La expresión es evaluada como si todavía estuviera en el interior del procedimiento invocador.

Mecanismos de paso de parámetros frente a la especificación de los parámetros:

El paso de parámetros está íntimamente ligado a la mecánica interna del código utilizado para su implementación.

Un lenguaje que intenta dar solución a este problema es Ada. Tiene dos conceptos de comunicación de parámetros: parámetros ***in*** y parámetros ***out***.

Un parámetro ***in*** actúa como si fuera una constante y el valor de un parámetro ***out*** nunca puede ser utilizado.

Verificación de tipo de los parámetros:

En los lenguajes con tipificado fuerte, las llamadas a procedimientos deben verificarse de modo que los argumentos concuerden en su tipo y número con los parámetros del procedimiento.

En el caso del paso por referencia, los parámetros deben tener el mismo tipo, pero en el caso del paso por valor, pueden permitirse conversiones.

8.4 Ambientes, activación y asignación de procedimientos.

El ambiente para un lenguaje estructurado en bloques con un alcance léxico puede mantenerse de manera parecida a una pila, creando un registro de activación en la pila del ambiente al entrar el bloque y liberándolo cuando éste sale.

Las variables declaradas localmente en el bloque se les asignan espacio en este registro de activación.

Un ambiente totalmente basado en pilas no resulta adecuado para encarar las variables de procedimientos ni la creación dinámica de éstos, y aquellos lenguajes que tengan estos servicios, están obligados a utilizar un ambiente totalmente dinámico más complejo con recolección de basura.

Ambientes totalmente estáticos:

Toda asignación de la memoria puede llevarse a cabo en tiempo de carga, y las localizaciones de todas las variables quedan fijas durante toda la ejecución del programa.

Cada procedimiento o función tiene un registro de activación fijo. Cada registro de activación se subdivide en varias áreas: Espacio para las variables locales, espacio para los parámetros pasados, dirección de retorno y espacio temporal para evaluar la expresión.

Cuando ocurre una llamada a un procedimiento, se evalúan los parámetros y, sus ubicaciones (utilizando paso por referencia) se almacenan en el espacio para los parámetros del registro de activación del procedimiento.

Ambientes de tiempos de ejecución basados en pilas:

En un lenguaje estructurado en bloques con recursión, las activaciones de los bloques de procedimientos no pueden asignarse estáticamente, ya que el procedimiento podría ser llamado antes de haber salido de su activación anterior, y por lo tanto, debe crearse una nueva activación en cada llamada del procedimiento.

Esto puede hacerse en una forma basada en pilas, con un nuevo registro de activación creado en la pila cada vez que entra o se libera a la salida del bloque.

Para manejar un ambiente basado en pilas es necesario asignar espacio en una activación para las variables locales, espacio temporal y un apuntador de retorno.

Debe mantenerse un apuntador a la activación presente. Este apuntador a la activación presente debe conservarse en una ubicación fija, por lo general un registro, y se conoce como ***apuntador de ambiente*** o ***ep***.

La segunda información adicional que se tiene que conservar en un ambiente basado en pilas es un apuntador al registro de activación del bloque desde el que entró dicha activación. La razón es que, cuando se sale de la activación presente, es necesario retirar el registro de activación presente de la pila de los registros de activación.

El apuntador de ambiente (e p) debe restablecerse para que apunte a la activación anterior. Este apuntador hacia el registro de activación anterior se conoce como ***enlace de control***.

Los campos en cada uno de los registros de activación deben contener: el enlace de control, la dirección de retorno, los parámetros pasados, las variables locales y las variables temporales.

Las variables locales se ubican en el registro de activación presente, ya que son estáticas, cada vez que entra el bloque se asignan las mismas clases de variables en el mismo orden.

Cada variable puede ser ubicada en la misma posición en el registro de activación con relación al principio del mismo. Esta posición se conoce como ***desplazamiento*** de la variable local.

En el caso de referencias no locales no es posible anidar los procedimientos. Cualquier referencia no local se encuentra en el área global.

Con procedimientos anidados, las referencias no locales ahora pueden ser variables locales.

Una manera para lograr el alcance léxico, es que un procedimiento mantenga un enlace a su ***ambiente léxico*** o de definición. A este enlace se le conoce como ***enlace de acceso***.

Ahora cada registro de activación necesita un nuevo campo, el campo del enlace de acceso.

Cuando los bloques están anidados profundamente, para encontrar una referencia no local podría ser necesario seguir varios enlaces de acceso en vez de uno solo.

Este proceso se llama encadenamiento de accesos, y la cantidad de enlaces de acceso que deberán seguirse corresponde a la diferencia entre niveles de anidamiento, o profundidad de anidamiento, entre el ambiente de acceso y el ambiente definidor de la variable que se está accediendo.

La cerradura de un procedimiento es el código del procedimiento, junto con un mecanismo para resolver referencias no locales.

Procedimientos calculados dinámicamente y ambientes totalmente dinámicos.

El ambiente de tiempo de ejecución basado en pilas es adecuado para los lenguajes estructurados en bloques con alcance léxico. El uso de cerraduras <e p, i p> para los procedimientos hace que este ambiente sea adecuado para los lenguajes con parámetros que son procedimientos por sí mismos, siempre y cuando dichos parámetros sean de valor.

Un ambiente basado en pilas tiene sus limitaciones, cualquier procedimiento que pueda devolver un apuntador a un objeto local, dará como resultado una ***referencia pendiente*** al salir del procedimiento.

8.5 Administración de la memoria dinámica.

En un lenguaje imperativo, la asignación y desasignación automática de la memoria ocurre únicamente para los registros de activación de una pila. Se asigna espacio cuando se llama a un procedimiento y se desasigna cuando se sale de él.

Los lenguajes con necesidades significativas de almacenamiento en el montón, como Java, están mejor al dejar el almacenamiento dinámico fuera de la pila a un administrador de la memoria que incluya recolección automática de basura.

Otra razón para tener recolección de basura es que, cualquier lenguaje que no aplique restricciones al uso de procedimientos y de funciones deberá incluirlo.

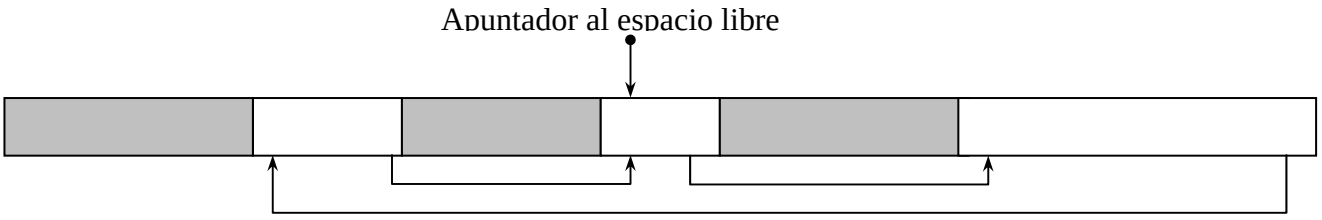
Se podría intentar resolver esto utilizando un procedimiento muy sencillo: no desasignando ninguna memoria una vez haya sido asignada.

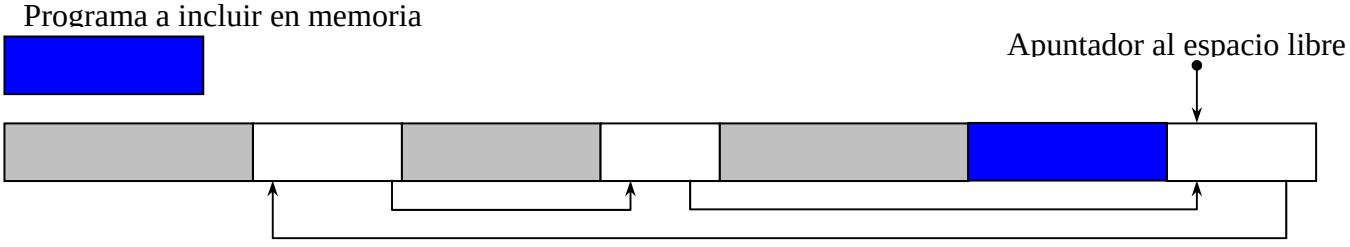
Este método es correcto y fácil de implementar, pero sólo puede funcionar en programas pequeños. Al no desasignar memoria, hace que ésta se termine muy rápidamente.

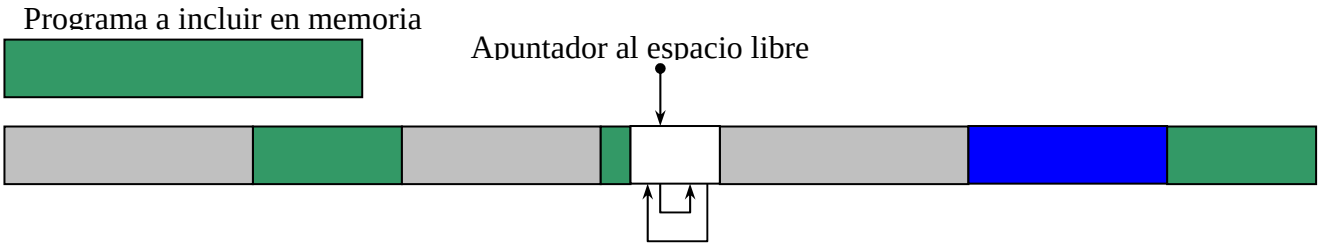
La administración de memoria se clasifica en dos categorías: ***recuperación*** de almacenamiento asignado y no utilizado (colectores de basura) y ***mantenimiento*** del espacio libre disponible para la asignación.

Mantenimiento del espacio libre:

Cuando es necesario incluir un programa en memoria, se busca un bloque libre donde quepa dicho programa. Si no existe dicho bloque toma las partes necesarias de los bloques libres hasta completar el espacio.

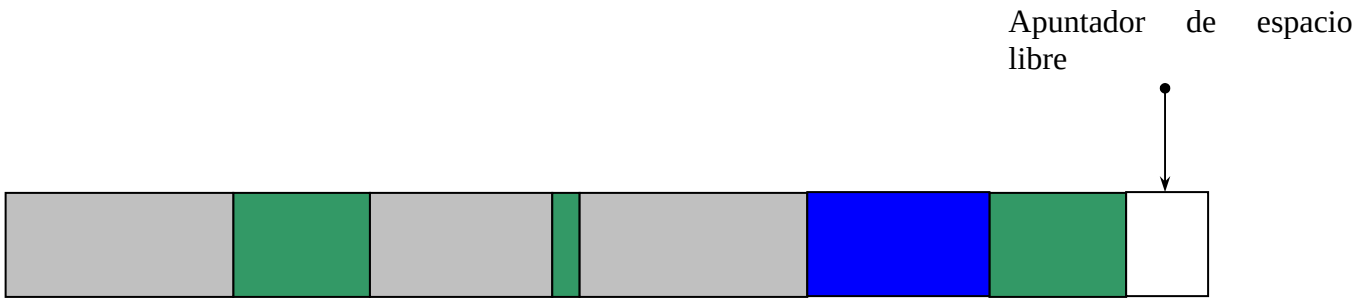






Cuando se devuelven los bloques de memoria a la lista de espacio libre, deberán unirse a bloques adyacentes, para formar el bloque contiguo más grande de memoria libre. El proceso se llama ***fusión*** o ***unión***.

Una lista puede quedar ***fragmentada***, es decir, estar formada por una cierta cantidad de bloques de pequeño tamaño. Cuando esto ocurre, es posible que falle la asignación de un bloque grande. Para evitar esto, la memoria debe ***compactarse*** moviendo todos los bloques libres para unirlos y crear un solo bloque.



Recuperación del almacenamiento:

Reconocer cuándo un bloque ya no está referenciado es una tarea mucho más difícil que el mantenimiento de la lista de memoria libre. Se han utilizado dos métodos principales: Conteo de referencias y marcar y barrer.

El conteo de referencias procura recuperar espacio tan pronto como éste deje de estar referenciado. Cada bloque contiene un campo de conteo que guarda el número de referencias. Cada vez que se cambia una referencia, debe ser actualizado. Cuando el conteo llega a cero, el bloque puede ser devuelto a la lista de memoria libre. Esto sufre de serios inconvenientes. Uno es la memoria adicional que necesita para mantener actualizados los conteos, otro mas serio es el esfuerzo de mantener el conteo.

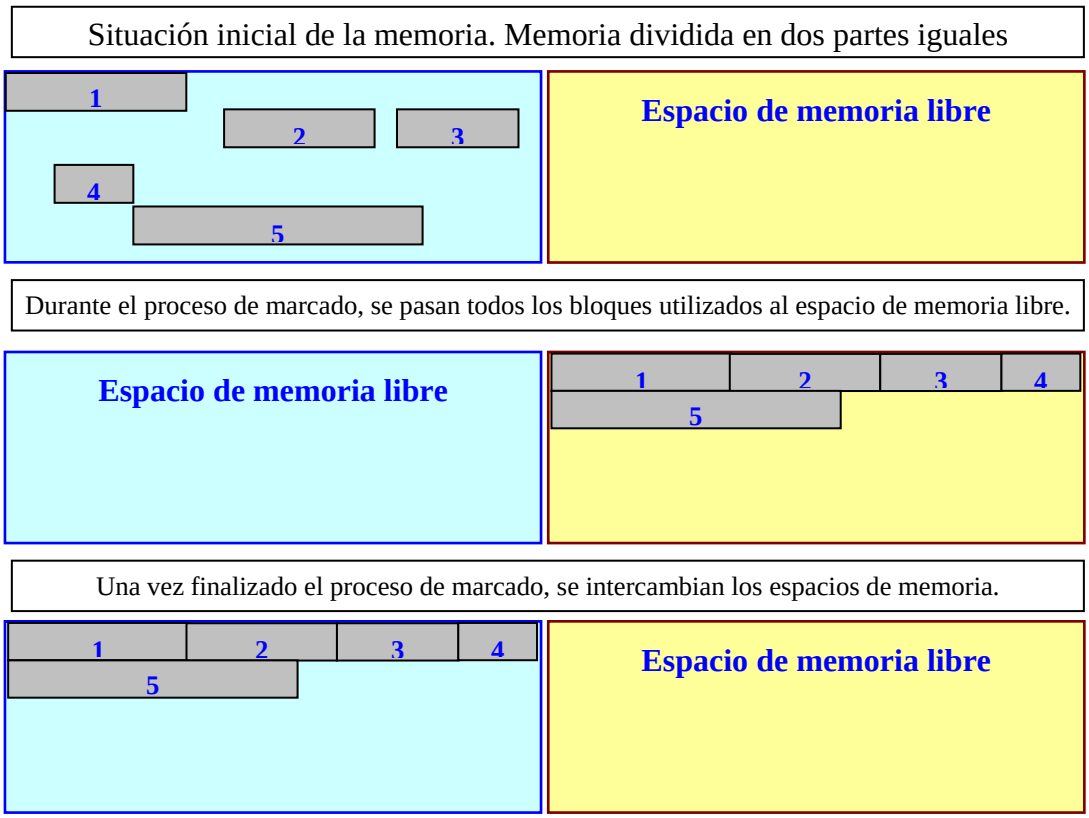
Los conteos de referencia deben ser decrementados recursivamente. Aún más serio es el hecho de que las referencias circulares pueden hacer que la memoria no referenciada jamás sea desasignada.

El método alterno estándar a los conteos de referencia es ***marcar*** y ***barrer***. Pospone la recuperación de almacenamiento hasta que el asignador se queda sin espacio, y en ese momento busca todo el almacenamiento que puede referenciarse y mueve todo el almacenamiento no referenciado de regreso a la lista libre.

Esto lo hace en dos pasadas. En la primera pasada se siguen todos los punteros de manera recursiva y marca cada bloque de almacenamiento localizado. Este proceso requiere de un bit adicional de almacenamiento para los bloques no marcados a la lista libre.

Este método no tiene dificultad al liberar bloques con referencias circulares. Sin embargo, también necesita de almacenamiento adicional y la doble pasada representa un retraso significativo.

Otra alternativa es la siguiente:



Manejo de excepciones y de ambientes:

En principio, las operaciones de poner de manifiesto y de manejar las excepciones son similares a las llamadas de procedimientos, y pueden implementarse de forma similar. Sin embargo, también existen importantes diferencias. Las principales son:

1. No puede crearse una activación en la pila en tiempo de ejecución para representar que se pone de manifiesto una excepción.
2. Debe localizarse un manejador y llamar dinámicamente, en vez de estáticamente.
3. las acciones de un manejador se basan en el tipo de la excepción, más que el valor de la excepción misma.

Tema 9 Tipos de Datos Abstractos y Módulos

En el capitulo 6 definimos los ***tipos de datos*** como un conjunto de valores con ciertas operaciones asociadas.

Estos tipos de datos fueron divididos en aquellos predefinidos por el lenguaje y los definidos por el usuario.

Los tipos predefinidos como integer o real, están diseñados para aislar al usuario del lenguaje de su implementación en la máquina.

Los tipos definidos por el usuario se construyen en base a los tipos ya definidos y utilizando los constructores de tipos del lenguaje. Sus estructuras son visibles para el usuario y no incluyen operación alguna salvo las operaciones de acceso.

Sería deseable en un lenguaje tener un mecanismo para poder construir tipos de datos con tantas características de un tipo incorporado como sea posible (con las operaciones sobre estos). Este mecanismo debería cumplir con los siguientes requisitos:

Un método para definir un tipo de datos y a la vez las operaciones sobre este.

Un procedimiento para reunir en un solo sitio los detalles de implementación y las operaciones, así como permitir su restricción.

Un tipo que satisfaga parte o la totalidad de estos criterios se le denomina un ***tipo de datos abstracto*** (o **TDA**).

Los dos criterios antes mencionados promueven tres metas del diseño que originalmente habían justificado la utilización de los tipos de datos como medio de ayuda: Capacidad de modificación, capacidad de reutilización y seguridad.

Otra alternativa para los criterios 1 y 2 para algunos autores es la ***ocultación*** y el ***encapsulado***.

El ***encapsulado*** significa reunir en una sola localización a todas las definiciones relacionadas con un tipo de datos, y restringiendo el uso del tipo a las operaciones definidas en dicha localización.

La ***ocultación*** se refiere a la separación de las definiciones de los detalles de implementación, y la supresión de dichos detalles en el uso del tipo.

En este tema utilizaremos los criterios 1 y 2 para definir los TDA.

Los mecanismos de tipos de datos abstractos no proporcionan el nivel de control activo que se espera en la verdadera programación orientada a objetos. La idea de los TDA es de hecho independiente del paradigma del lenguaje utilizado para su implementación (funcional, imperativo, orientado a objetos).

Un problema adicional es que un mecanismo de TDA a menudo se conoce como un concepto algo más general, denominado modulo, y que es una colección de servicios que pueden o no incluir un tipo o tipos de datos.

9.1 Especificación algebraica de los TDA.

En una especificación general de un tipo de datos se necesitará incluir el nombre del tipo y los nombres de las operaciones, incluyendo una definición de sus parámetros y valores devueltos.

Esta es la especificación sintáctica de un tipo de datos abstracto (conocido también como signatura del tipo).

Para una especificación independiente del lenguaje sería apropiado utilizar la notación de funciones para las operaciones del tipo de datos: $f : X \to Y$ (donde X es el dominio e Y es el rango).

Como ejemplo, los números complejos podrían tener la siguiente signatura:

Tipo complejo imports real.

Operaciones:

Suma: la suma de 2 números complejos es otro número complejo.
Resta: la resta de 2 números complejos es otro número complejo.
Producto: el producto de 2 números complejos es otro número complejo.
División: la división de 2 números complejos es otro número complejo.
Negación:la negación de un número complejo es otro número complejo.
Makecomplex: un número complejo está formado por dos números reales.
Realpart: la parte real de un número complejo es un real.
Imaginarypart: la parte imaginaria de un número complejo es un real.

Tipo complex imports real
Operaciones:
+: complex x complex -> complex
-: complex x complex -> complex
*: complex x complex -> complex
/: complex x complex -> complex
-: complex -> complex
Makecomplex: rea x real –> complex
Realpart: complex -> real
Imaginarypart: complex -> real

En matemáticas es frecuente que las propiedades semánticas de las funciones se describan mediante ***ecuaciones*** o ***axiomas***. Si se trata de operaciones aritméticas, las leyes asociativa, conmutativa y distributiva son ejemplos de axiomas.

La ***especificación algebraica*** de un tipo abstracto combina signatura, variables y axiomas ecuacionales, la cual proporciona una especificación concisa de un tipo de datos y de sus operaciones asociadas.

La ***semántica ecuacional*** indica claramente cual va a ser su comportamiento en la implementación.

Una operación que crea un nuevo tipo de datos que se está definiendo es un ***constructor***, mientras que una operación que recupera un valor anteriormente construido se llama ***inspector***.

Las operaciones de los inspectores pueden dividirse en ***predicados***, que devuelven un valor booleano, y ***selectores*** que devuelven otros valores no booleanos.

En general se necesita un axioma por cada combinación de un inspector con un constructor.

9.2 Mecanismos de TDA y Módulos.

Mecanismos de Tipos de Datos Abstractos:

Algunos lenguajes tienen un mecanismo específico para expresar los tipos de datos abstractos.

Dicho mecanismo deberá tener alguna forma de separar la especificación o signatura del TDA (el nombre del tipo que se esta definiendo, y los nombres y tipos de las operaciones) de su implementación.

También debe garantizar que cualquier código fuera de la definición del TDA no puede utilizar detalles de la implementación, pero puede operar sobre un valor del tipo definido sólo a través de las operaciones proveídas.

Módulos:

Un módulo es una unidad de programación con una interfaz pública y una implementación privada; todos los servicios disponibles de un módulo quedan descritos en su interfaz pública y se exportan a otros módulos, así como todos los servicios requeridos deben importarse de otros módulos.

Se pueden agrupar una serie de funciones que estén íntimamente relacionadas, encapsular las definiciones e implementaciones y ocultar los detalles de implementación. Sin embargo, un paquete no está directamente relacionado con un tipo de datos y por tanto, dicho paquete no se ajusta a la definición de TDA.

Los módulos son una herramienta esencial en la descomposición, control de complejidad y la creación de bibliotecas de un programa, para compartir y reutilizar el código.

Los módulos ayudan en la proliferación de nombres. Una interpretación de un módulo es que su propósito principal es proporcionar alcances controlados, exportando únicamente aquellos nombres que requieren su interfaz y manteniendo ocultos los demás.

9.7 Problemas que se presentan con los mecanismos de TDA.

Los mecanismos de TDA en los lenguajes de programación utilizan servicios de compilación por separado para llenar los requisitos de protección y de independencia en la implementación.

La especificación del TDA se utiliza como interfaz.

Los módulos no son tipos: Un módulo debe exportar un tipo así como operaciones.

Los módulos son entidades estáticas: Los módulos son por naturaleza estáticos. Una posibilidad para la implementación de un TDA es no revelar un tipo, evitando que los clientes dependan de los detalles de la implementación, impidiendo cualquier mal uso.

Los módulos que exportan tipos no controlan adecuadamente las operaciones sobre variables de dichos tipos: El módulo exportador no puede garantizar que la asignación y la prueba de igualdad sean llamados antes de que se utilicen las variables.

Los módulos no siempre representan adecuadamente la forma en la que dependen de los tipos importados: A menudo, los módulos dependen de la existencia de ciertas operaciones sobre los parámetros de tipo y pueden también llamar a funciones cuya existencia no se ha hecho explícita en la especificación del módulo.

Las definiciones de los módulos no incluyen especificación de la semántica de las operaciones incluidas: En la especificación algebraica de un tipo abstracto, se dan ecuaciones que especifican el comportamiento de las operaciones. Y sin embargo no se requiere especificación del comportamiento de las operaciones disponibles.

9.8 Las matemáticas de los TDA.

En la especificación algebraica de un TDA, se especifica un conjunto de operaciones y un conjunto de axiomas en forma ecuacional. No se dice nada sobre la existencia actual de este tipo, que sigue siendo hipotético hasta que se construya un tipo que cumpla con todos los requisitos.

Tipo existencial: Afirma la existencia de un tipo que cumple sus requisitos.

Modelo: Un conjunto y operaciones que cumplen la especificación. Dada una especificación algebraica puede que exista uno o varios modelos.

Género: Es el nombre de un tipo que no está todavía asociado con ningún conjunto real de valores.

Signatura: Es el nombre y tipo de operación o conjunto de operaciones que existe sólo en teoría, sin tener todavía ninguna operación real asociada.

Un modelo es, por tanto, la actualización de un género y de su signatura y se conoce como un ***álgebra***.

Álgebra inicial: Cuando los únicos términos que son iguales son aquellos que pueden comprobarse que efectivamente lo son a partir de los axiomas. El resultado del Álgebra inicial es la semántica inicial.

Álgebra final: Es cuando dos valores de datos cualquiera en los que la operación de inspección no pueda encontrar diferencias, entonces deben ser iguales. El resultado del Álgebra final es la semántica final.

Principio de extensionalidad: dos cosas son iguales cuando todos sus componentes son iguales.

Un sistema de axiomas debe ser ***consistente*** y ***completo***:

Consistencia: Los axiomas no deben identificar términos que deben ser distintos. Por ejemplo: **empty (create q) = true** y **empty (create q) = false** son axiomas inconsistentes. La consistencia del sistema de axiomas requiere que el álgebra inicial no sea "***demasiado pequeña***".

Completo: Lo completo de un sistema de axiomas declara que el álgebra inicial no es "***demasiado grande***". Al agregar axiomas se identifican más elementos, haciendo que el álgebra inicial sea menor, en tanto que eliminar axiomas lo hace más grande.

Tema 10 Programación orientada a objetos

Un objeto es una entidad con ciertas propiedades, y con una cierta capacidad de reacción ante eventos o sucesos.

La programación orientada a objetos es un mecanismo efectivo para la reutilización y la modificación de código.

Posee ciertos inconvenientes como pueden ser su capacidad de expresar tipos de datos abstractos.

10.1 Reutilización e independencia del software.

Existen cinco formas básicas para modificar un componente de software de modo que pueda reutilizarse:

Extensión de los datos y/o operaciones.

Restricción de los datos y/o operaciones: En esencia, esta operación es lo contrario que la anterior.

Redefinición de una o más de las operaciones. Incluso si las operaciones se conservan esencialmente igual, pudiera ser necesario redefinir algunas para que acepten un nuevo comportamiento.

Abstracción o la reunión de operaciones similares de dos componentes diferentes en uno nuevo.

Polimorfización, o la extensión del tipo de datos a los cuales se pueden aplicar las operaciones. Las operaciones comunes provenientes de diferentes tipos son abstraídas y reunidas en una sola.

El diseño para la reutilización no es el único objetivo de los lenguajes de programación orientados a objetos; otro es la restricción del acceso a los detalles internos de los componentes del software. Para asegurar que los clientes utilicen los componentes de manera tal que resulten independientes de los detalles de implementación de éstos y que cualquier cambio en la implementación de uno de ellos sólo tenga un efecto local.

10.2 Java: Objetos, clases y métodos.

El estado de un objeto en un lenguaje orientado a objetos es principalmente de índole ***interna***, es decir, ***local*** para el objeto mismo. Es decir, el estado de un objeto está representado por variables locales declaradas como parte del objeto e inaccesibles a componentes externos al objeto.

Métodos:Cada objeto incluye un conjunto de funciones y procedimientos a través de los cuales puede tenerse acceso y puede modificarse el estado local del objeto. Éstos se conocen como métodos.

Clases: Los objetos pueden declararse al crear un patrón para el estado y los métodos locales. Este patrón se conoce como clase, y esencialmente es justo como un tipo de datos.

Objetos: Un objeto es una ***instancia de una clase***. Las variables locales que representan el estado de un objeto se conocen como ***variables de instancia***.

Los conceptos de clase y objeto pueden visualizarse como la generalización de la idea de tipo de registro o estructura y de variables respectivamente.

Constructores. Son igual que los métodos pero no devuelven ningún valor y el nombre del método es el mismo que el de la clase.

La mayoría de los lenguajes orientados a objetos requieren de recolección automática de basura.

Algunos lenguajes orientados a objetos permiten ***multimétodos***, en los cuales más de un objeto puede ser el objetivo de una llamada de método.

El concepto de clase puede visualizarse como la generalización de la idea de tipo de registro o estructura y el objeto, como la generalización de una variable.

10.3 Herencia.

La herencia permite compartir datos y operaciones entre clases, así como redefinir operaciones sin necesidad de modificar el código existente.

La herencia que obedece al principio del subtipo expresa la relación ***es-un***: si B es una subclase de A, entonces todos los objetos pertenecientes a B también pertenecen a A.

"downcasting": Es la conversión forzada de tipos. En caso de no ser correcta dicha conversión, se produce un error en tiempo de ejecución y no en tiempo de compilación.

Ejemplo: q = (Deque) p; el tipo de p se fuerza a que sea del tipo Deque.

El método ***equals*** está diseñado para probar la ***igualdad de valor*** de los objetos, en contraposición a la igualdad de referencia o de puntero realizada mediante = =.

Ejemplo:

String s = "Hola";

String t = new String ("Hola");

/* Ahora s = = t es erróneo pero s.equals (t) es correcto */.

Un ***método abstracto*** es un método que está siempre para los objetos de la clase, pero que se le dan diferentes implementaciones para las diferentes subclases.

Ejemplo:

```
public abstract class FiguraCerrada
{       public abstract double área ();     Método no definido cuya implementación queda diferida.
        private Point centro;               Variable de instancia del tipo Point llamada centro.
        public FiguraCerrada (Point c)      Constructor al que se le pasa como parámetro una variable c del tipo Point.
        {       Centro = c;
                //…
        }
}
```

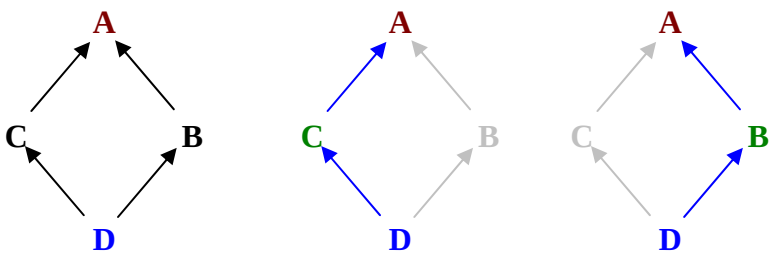
Los métodos abstractos a veces se conocen como ***diferidos***, y una clase que tiene un método diferido se conoce como una clase ***diferida***. La ventaja de definir métodos abstractos o diferidos no es sólo para consolidar el código, sino también para que sea un requisito para que cualquier objeto que instancie a dicha clase tenga las funciones definidas en ella.

La palabra clave ***Super*** se utiliza para indicar operaciones en el objeto presente interpretado como un miembro de la clase padre, en tanto que ***this*** se refiere al objeto presente como miembro de su clase definidora.

La herencia establece una relación de dependencia padre-hijo entre las superclases y las subclases.

La herencia proporciona muchas oportunidades de polimorfismo ya que cualquier método aplicable a una clase A, puede ser aplicado a las subclases de A, donde además dichos métodos pueden ser redefinidos. A este tipo de polimorfismo se le llama ***polimorfismo de subtipo*** y tiene un efecto similar a la sobrecarga.

La gráfica de herencia es un árbol. Esta situación se conoce como ***herencia sencilla***, ya que cada subclase sólo hereda de una superclase. También es posible tener ***herencia múltiple***, en la que una clase hereda de dos o más superclases.En un lenguaje con herencia múltiple, las gráficas de herencia pueden ser gráficas acíclicas dirigidas en vez de árboles. Esto significa que los métodos pueden heredarse de más de una manera.



Java tiene un mecanismo para permitir la herencia múltiple llamado ***interfaz***.

Una interfaz es parecida a una clase, pero sólo especifica los nombres y los tipos de los métodos (signatura), sin proporcionar la implementación, las variables de instancia ni los constructores.

10.4 Ligadura dinámica.

Una de las características que distinguen a las clases de los módulos o paquetes es la naturaleza dinámica de las clases en contraste con la naturaleza estática de los módulos.

Es posible ver esto en el hecho de que a los objetos de una clase se les asigna almacenamiento de forma totalmente dinámica por lo general en un montón.

La asignación de memoria puede ser asignada por el usuario, estar totalmente automatizada o bien ser un hibrido.

Java por ejemplo, es un sistema hibrido. La asignación (new) y la inicialización las realiza el usuario, mientras que la eliminación es automática (no existe un delete).

En los lenguajes Orientados a Objetos es posible ofrecer al mismo tiempo la ligadura estática y la dinámica de los métodos.

Si un método es declarado como virtual, se utilizará la ligadura dinámica. En C++ todos los métodos son estáticos por defecto y para hacerlo dinámico debemos utilizar la palabra clave ***virtual***.

En java, por el contrario, los métodos son dinámicos, y para hacerlos estáticos utilizamos la palabra clave ***super***.

Los lenguajes orientados a objetos implementan las ligaduras dinámicas como parte del entorno de ejecución, ya que las llamadas de métodos dinámicos no pueden resolverse en tiempo de compilación.

```
class A
{       void p()
        {       System.out.println("A.p");}
        void q()
        {       System.out.println("A.q");}
        void f()
        {       p(); q();}
}class B extends A
{       void p()
        {System.out.println("B.p");}
        void q()
        { System.out.println("B.q");
                super.q();}
}public class VirtualExample
{       public static void main(String[] args)
        {
        A a = new A();
        a.f();
        a = new B();
        a.f();
        }}
```

Salida:

A.p
A.q
B.p
B.q
A.q

La llamada **Super.q()** tiene el mismo efecto que una llamada no ligada dinámicamente.

10.5 C++.

Las variables de instancia se conocen como ***miembros de datos*** y los métodos como ***funciones miembro***.

A las subclases se les denomina ***clases derivadas*** y a las superclases ***clases base***.

Los objetos no son automáticamente apuntadores o referencias.

El tipo de datos ***Class*** en C++ es en esencia idéntico al tipo de datos ***Struct***.

En C++ se incluyen tres niveles de protección: ***público***, ***privado*** y ***protegido***.

Los miembros ***públicos*** son accesibles por el código del cliente y por las clases derivadas.

Los ***protegidos*** no son accesibles por el código del cliente pero sí por las clases derivadas.

Los ***privados*** no son accesibles ni por el código del cliente ni por las clases derivadas.

En C++, la inicialización de los objetos se efectúa como en java mediante ***constructores*** que tienen el mismo nombre que el de la clase.

C++ no tiene un recolector de basura, por lo que tiene ***destructores***, que al igual que los constructores, utilizan el mismo nombre que la clase pero con una tilde delante "~".

Los destructores son llamados automáticamente al desasignar un objeto, ya sea por haberse salido del alcance o por uso del operador ***delete***.

Para que exista la ligadura dinámica en las funciones de C++, dicha función debe definirse utilizando la palabra clave ***virtual***.

Por ejemplo:

```
class Rectángulo {
    public:
        virtual double area()
        {return ancho * alto}
    private:
    …
}
```

Ejemplo:

```
class A{
        public:
                void p(){printf( "A::p\n");}
                virtual void q() {printf( " A::q\n");}
                void f(){   p(); q(); }};
class B : public A
{
        public:
                void p() {printf("B::p\n");}
                void q() {printf("B::q\n");}};
int main()
{       A a;
        B b;
        a.f();
        b.f();
        a=b;
        a.f();}
```

```
int main()
{       a.f()=
        void f(){
                void p(){printf( "A::p\n");};
                 virtual void q() {printf( " A::q\n");};
};
        b.f()=
                void p(){printf( "A::p\n");}
                void q() {printf("B::q\n");}
        a=b;
        a.f()=
                void p(){printf( "A::p\n");}
                virtual void q() {printf( "A::q\n");}
}
        A::p    A::q    A::p    B::q    A::p    A::q
```

```
class A{
        public:
                void p(){printf( "A::p\n");}
                virtual void q() {printf( " A::q\n");}
                void f(){    p(); q(); }};
class B : public A
{
        public:
                void p() {printf("B::p\n");}
                void q() {printf("B::q\n");}};
int main()
{       A* a=new A; B* b=new B;
        a->f();
        b->f();
        a=b;
        a->f();}
```

```
int main()
{       a->f()=
        void f(){
                void p(){printf( "A::p\n");};
                 virtual void q() {printf( " A::q\n");};
};
        b->f()=
                void p(){printf( "A::p\n");}
                void q() {printf("B::q\n");}
        a=b;
        a->f()=
                void p(){printf( "A::p\n");}
                void q() {printf("B::q\n");}
}
        A::p    A::q    A::p    B::q    A::p    B::q
```

La herencia múltiple en C++ generalmente crea copias por separado de cada clase. Por ejemplo, las declaraciones:

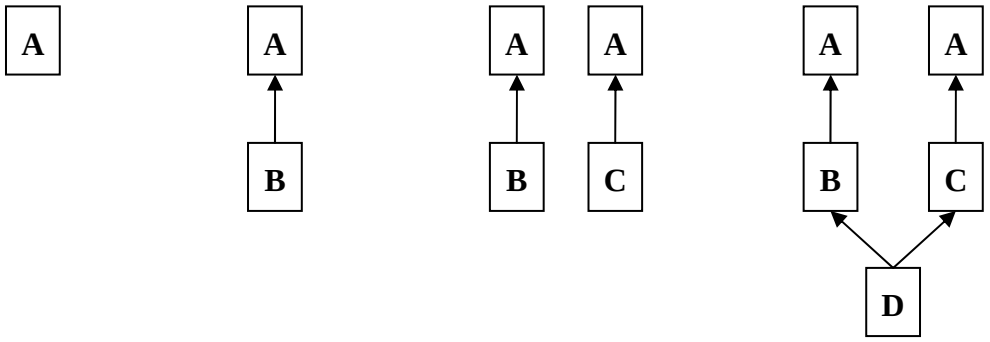
Class A {…};

Class B : public A {…};

Class C : public A {…};

Class D : public B : public C {…};

Proporcionan cualquier objeto de la clase D con dos copias por separado de los objetos de la clase A, y crea la siguiente gráfica de herencia:



Esto es herencia repetida.

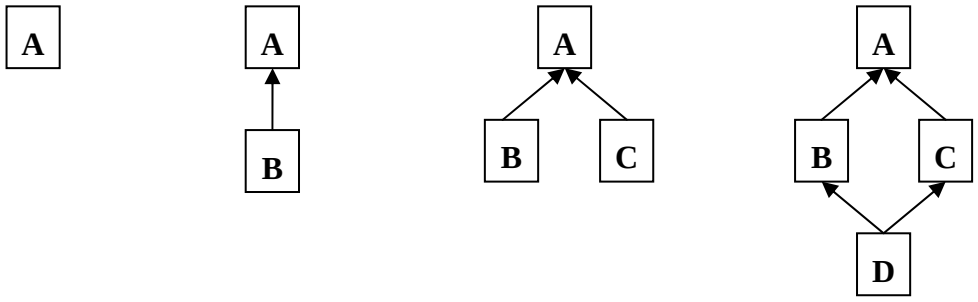
Para obtener sólo una copia de una clase A en la clase D, debe declararse la herencia usando la palabra clave ***virtual***.

Class A {…};

Class B : ***virtual*** public A {…};

Class C : ***virtual*** public A {…};

Class D : public B : public C {…};



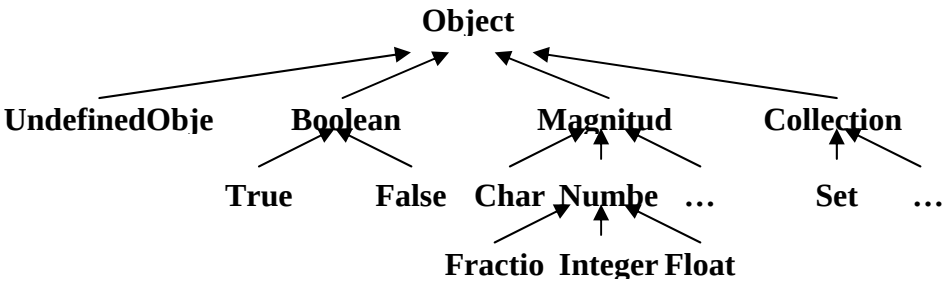
Esto se conoce como herencia ***compartida***.

10.6 Smalltalk.

En Smalltalk toda entidad es un objeto, incluyendo las constantes, los números y los caracteres.

Smalltalk incluye clases preexistentes, y cada clase nueva debe ser una subclase de una clase existente.

Los nombres de las clases deben empezar por mayúsculas, en tanto que los nombres de los objetos y de los



métodos empiezan por minúsculas.

En Smalltalk no hay exportación ni control de privacidad de los métodos. Sin embargo, el acceso a las variables de instancia queda limitado a los métodos: los clientes no pueden acceder directamente a las variables de instancia.

El estado de un objeto puede modificarse sólo invocando un método propio del objeto.

En Smalltalk los nombres de los métodos se conocen como ***selectores***, y están divididos en categorías: selectores ***unarios*** que corresponden a operaciones sin parámetros, y selectores de ***palabras clave***, que corresponden a operaciones con parámetros.

Un mensaje, o una invocación a un método, está representado por el nombre seguido del nombre del método y de los parámetros.

Por ejemplo: lo que en C sería **q.enqueue(x)**, en Smalltalk es **q enqueue : x**.

Los ***selectores binarios***, se utilizan para operaciones binarias y que deben estar formados por uno o dos caracteres no alfabéticos.

2+3 en Smaltalk significa lo mismo que 2. + (3) y es un mensaje al objeto 2 solicitando que le sume 3 a sí mismo.

Las variables de instancia no reciben tipos, ya que Smalltalk tiene tipificación dinámica.

Las variables locales se declaran entre barras verticales.

El punto "." Se utiliza para separar enunciados, en vez del punto y coma.

El operador "==" prueba la identidad de los objetos.

Un bloque en Smalltalk es una secuencia de código rodeada por corchetes.

El símbolo ↑ significa "devuelve ese valor" o return. Si no se especifica un valor, devuelve el objeto presente.

10.7 Cuestiones de diseño en lenguajes orientados a objetos.

Las características Orientadas a Objetos representan capacidades dinámicas más que estáticas, por lo que un aspecto del diseño es introducir características de modo que se reduzca la penalización en tiempo de ejecución.

Otro problema que se presenta con relación a un diseño eficiente de lenguajes Orientados a Objetos incluye la adecuada organización del entorno de ejecución.

Clases en contraposición con tipos.

Las clases deben incorporarse de alguna manera en el sistema de tipos. Existen varias posibilidades:

Las clases podrían específicamente excluirse de la verificación de tipos. Este es el método más sencillo.

Las clases podrían convertirse en constructores de tipos. Si X es una subclase de la clase Y, entonces la verificación de tipos debe permitir la asignación Y=X, pero deberá indicar un error en la igualdad X=Y.

Las clases pueden convertirse simplemente en el sistema de tipos. Esto es, todos los demás tipos estructurados quedan excluidos del lenguaje. El uso de las clases como la base del sistema de tipos significa que, bajo la tipificación estática, puede verificarse la validez de todas las asignaciones de objetos antes de la ejecución a pesar de no conocer la membresía exacta de clase sino hasta el tiempo de ejecución.

Clases en contraposición con módulos.

Las clases incluyen un mecanismo versátil para organizar código.

Proporcionan un mecanismo para la especificación de interfaces, para la localización de código que se aplica a colecciones específicas de datos y para la protección de la implementación a través del uso de datos privados.

Las clases no permiten una separación limpia de la implementación con relación a la interfaz, ni tampoco protegen la implementación de la exposición al código del cliente.

Las clases sólo son adecuadas para controlar la importación y exportación de nombres. Puede utilizarse una clase como un espacio de nombres o un alcance para una colección de servicios.

Sin embargo, no es posible importar nombres de otras clases y reexportar una selección de los mismos sin un cierto esfuerzo, por lo que las clases no funcionan bien como espacio de nombres o módulos estáticos.

Por esta razón, los lenguajes incluyen mecanismos que son independientes de los mecanismos orientados a objetos.

Herencia en contraposición con polimorfismo.

Existen tres tipos de polimorfismo:

Polimorfismo paramétrico: donde los parámetros pueden quedarse sin especificar en las declaraciones.

Sobrecarga o polimorfismo ad-hoc: Diferentes funciones o declaraciones de método comparten el mismo nombre y se elimina la ambigüedad mediante el uso de los tipos de los parámetros.

Polimorfismo de subtipo: donde todas las operaciones de un tipo pueden aplicarse a otro tipo.

La Herencia es una clase de polimorfismo de subtipo. También puede visualizarse como una especie de sobrecarga sobre el tipo del parámetro de objeto implícito (el apuntador ***this***).

El problema que se presenta con la mezcla de ***herencia*** y ***sobrecarga*** es que no toma en consideración métodos binarios que pudieran necesitar sobrecarga basados en la membresía de clase en dos o más parámetros. Este se conoce como el problema de ***doble despacho*** o ***multidespacho***.

Por ejemplo:

> Si ***x*** es del tipo **complex** e ***y*** es un entero o un real, el método ***x.add(y)*** puede resolverse convirtiendo ***y*** en un complex y realizando la suma definida en el tipo complex.
>
> Pero no podríamos llevar a cabo la suma ***y + x***, puesto que ***y*** es un entero o un real, y ***x*** es un complex y el operador + para enteros o reales no contempla la conversión de complex a enteros o reales.

Multimétodos: métodos que pueden pertenecer a más de una clase y cuyo despacho de sobrecarga puede basarse en la membresía de clase de varios parámetros.

Aunque la sobrecarga y la herencia son conceptos relacionados, el polimorfismo paramétrico es más difícil de integrar.

10.8 Cuestiones de implementación en lenguajes orientados a objetos.

Analizamos algunos de los problemas que se encuentran al implementar un lenguaje orientado a objetos, y métodos que permiten la implementación de las características orientadas a objetos de manera eficiente.

Implementación de objetos y métodos:

Los objetos se implementan exactamente igual que las estructuras de registro en C o en Ada, con variables de instancia que representan los campos de datos de la estructura.

Class Point
{ …
Public void moverA (double dx, double dy);
{ x += dx;
y += dy;
}
…
Private double ***x***, ***y***;
}
 	
Class ColoredPoint extends Point
{…
Private Color ***color***;
}
	Struct
	{ double x;
	double y;
	};

La asignación sería de la siguiente forma:

Espacio para ***x***.
Espacio para ***y***.
Espacio para ***color***.

Los métodos difieren de las funciones de dos formas fundamentales:

La primera es que un método puede tener acceso directo a los datos del objeto presente en la clase.

La segunda es un problema con la ligadura dinámica de los métodos durante la ejecución.

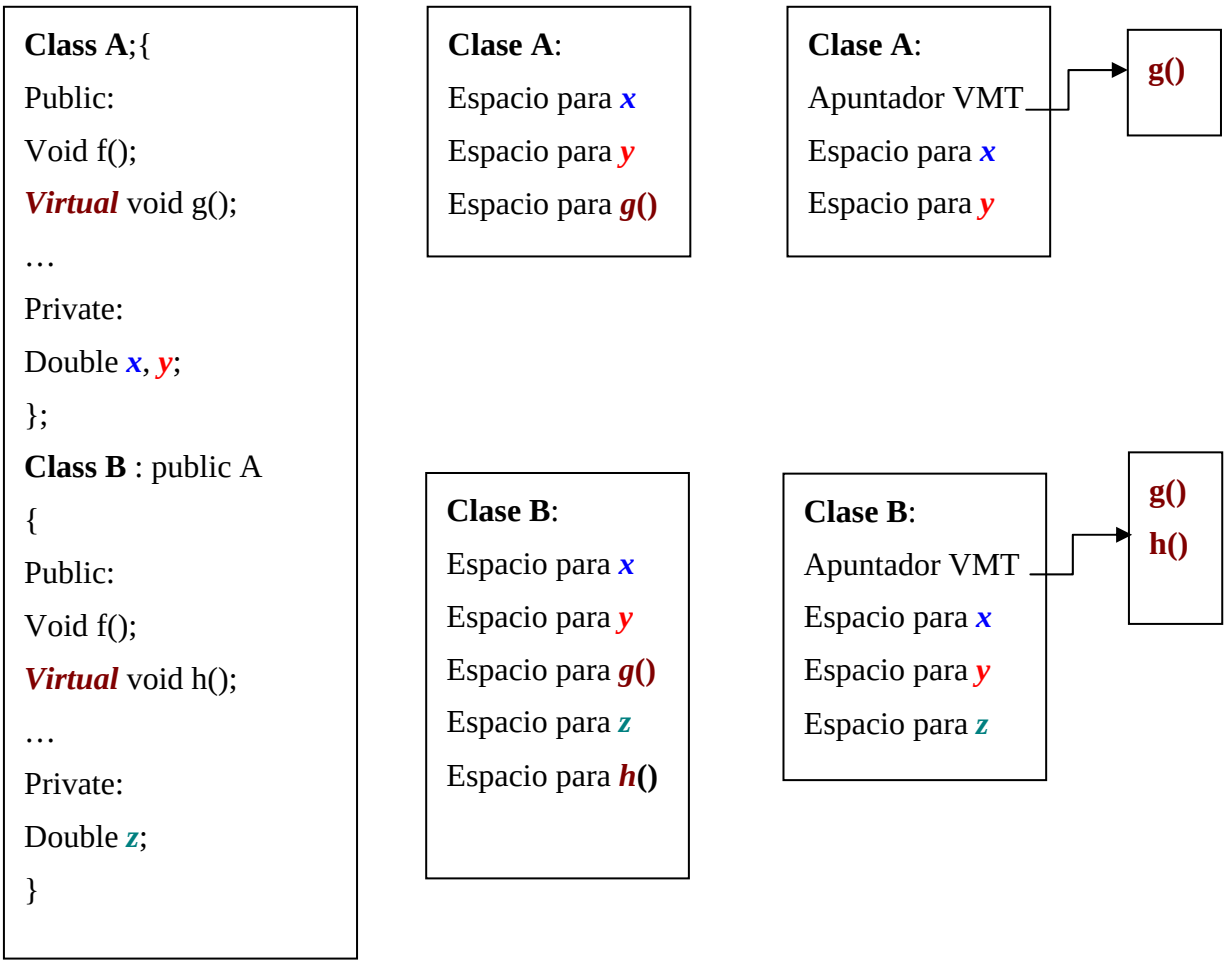
Herencia y Ligaduras Dinámicas:

Se presenta un problema cuando se usa la ligadura dinámica para los métodos, ya que el método no es conocido excepto durante la ejecución. Una solución posible sería conservar todos los métodos ligados dinámicamente como campos adicionales en las estructuras asignadas para cada objeto.

El problema es que cada estructura debe mantener una lista de todas las funciones virtuales disponibles en ese momento, y dicha lista podría resultar muy grande.

Un procedimiento alterno para esta estrategia sería mantener una tabla de métodos virtuales para cada clase.

Estas tablas se conocen como ***tabla de método virtual*** o VMT.



Asignación e inicialización:

Los lenguajes orientados a objetos pueden mantener un entorno de ejecución de la manera tradicional de pilas y montículos.

En el caso de la asignación a la pila, el compilador es el responsable de la generación automática de llamadas al constructor a la entrada del bloque y de las llamadas al destructor a la salida del bloque.

En el caso de la asignación de los apuntadores en el montículo, son las llamadas explícitas a ***new*** y ***delete*** quienes llaman a los constructores y destructores.

Java no tiene rutinas explícitas de desasignación, por lo que requiere del uso de un recolector de basura, lo que supone una carga para la ejecución y una mayor complejidad.

Durante la inicialización, es necesario ejecutar el código de inicialización de una clase anterior antes de llevar a cabo las inicializaciones actuales.

Donde la herencia múltiple puede crear varias trayectorias de una clase descendente, el código de inicialización se ejecuta de acuerdo con el orden topológico determinado por el orden en el que las clases padre aparecen listadas en cada declaración de clase derivada.

Tema 11 Programación funcional

La programación funcional tiene varias ventajas sobre la programación imperativa.

Incluyen la concepción de programas como funciones, el trato de las funciones como si fueran datos, la limitación de efectos colaterales y el uso de administración automática de la memoria.

Un lenguaje de programación funcional tiene gran flexibilidad, es conciso en su notación y su semántica es sencilla.

El inconveniente ha sido la ineficiencia en la ejecución. Debido a su naturaleza dinámica, estos lenguajes han sido interpretados más que compilados, resultando en una pérdida sustancial en velocidad de ejecución.

Programas como funciones.

Una ***función*** es una regla que asocia a cada ***x*** de algún conjunto de valores una única ***y*** de otro conjunto de valores.

Podemos referirnos a ***x*** como “entrada” y a ***y*** como “salida”. Por lo tanto, el punto de vista funcional de la programación no hace ninguna distinción entre un programa, un procedimiento y una función. Hace siempre una distinción entre valores de entrada y de salida.

Debemos distinguir entre definición de función y aplicación de función: la primera es la declaración que describe la forma en la que debe calcularse una función utilizando los parámetros formales, en tanto que la aplicación de la función es una llamada a una función declarada utilizando parámetros reales, es decir, argumentos.

La propiedad de una función de que su valor dependa sólo de los valores de sus argumentos se conoce como ***transparencia referencial***.

Una función transparente referencialmente sin parámetros deberá siempre devolver el mismo valor, y por lo tanto, no tiene ninguna diferencia de una constante.

La carencia de asignación y la transparencia referencial hacen que la semántica de los programas funcionales resulte simple.

El ambiente de ejecución asocia nombres sólo a valores (y no a localizaciones de memoria), y una vez introducido un nombre en el ambiente, su valor no puede cambiar jamás. Esta idea de la semántica a veces se conoce como ***semántica de valor***.

Las funciones deben considerarse como valores que pueden ser calculados por otras funciones y que también pueden ser parámetros de funciones. Expresamos esta generalidad diciendo que las funciones son ***valores de primera clase***.

Funciones que tienen parámetros que a su vez son funciones, o producen un resultado que es una función, o ambos, se llaman funciones de ***primer orden***.

Cualidades de los lenguajes de programación funcional y de programas funcionales:

Todos los procedimientos son funciones y distinguen claramente los valores de entrada (parámetros) de los de salida (resultados).

No existen variables ni asignaciones. Las variables han sido reemplazadas por los parámetros.

No existen ciclos. Estos han sido reemplazados por llamadas recursivas.

El valor de una función depende solo del valor de sus parámetros y no del orden de evaluación o de la trayectoria de ejecución que llevó a la llamada.

Las funciones son valores de primera clase.

Programación funcional en un lenguaje imperativo.

El requisito básico para la programación funcional en cualquier lenguaje es la disponibilidad de la recursión y un mecanismo general de funciones adecuado.

Los lenguajes imperativos contienen varias restricciones:

Los valores estructurados como los arreglos y los registros no pueden ser valores devueltos de las funciones.

No existe forma de construir un valor de un tipo estructurado de manera directa.

Las funciones no son valores de primera clase, por lo que no es posible escribir funciones de orden superior.

TEMA 12 – PROGRAMACIÓN LÓGICA

12.1 Lógica y programas lógicos.

El tipo de lógica que se utiliza en la programación lógica es el cálculo de predicados de primer orden.

El cálculo de predicados de primer orden clasifica las distintas pares de estos enunciados de la siguiente forma:

Constantes: números o nombres (átomos)

Predicados: nombres de funciones que son verdaderas o falsas.

Funciones: el cálculo de predicados distingue entre funciones que son verdaderas o falsas (predicados) y todas las demás funciones (representan valores no booleanos)

Variables: representan cantidades todavía no especificadas.

Conectores: incluyen operaciones AND, OR, implicación ->, y equivalencia <->.

Cuantificadores: son operaciones que introducen variables (cuantificador universal y existencial)

Símbolos de puntuación: incluyen paréntesis izquierdo y derecho, la coma y el punto.

Un lenguaje de programación lógico es un sistema de notación para escribir enunciados lógicos junto con algoritmos especificados para implementar las reglas de inferencia.

El conjunto de enunciados lógicos que se toman como axiomas pueden considerarse como el programa lógico.

En ocasiones los sistemas de programación lógica se conocen como bases de datos deductivas.

12.2 Cláusulas Horn.

Es un enunciado de la forma: a_1 y a_2 y a_3 ... y a_n -> b, donde los a_i solo se les permite ser enunciados simples sin involucrar conectores.

En las cláusulas Horn no existen conectores "o" ni cuantificadores.

La cláusula Horn dice que b es verdadero si todos los a_i son verdaderos.

A b se le llama cabeza de la cláusula y a a_1 y a_2 y a_3 ... y a_n cuerpo de la cláusula, la cantidad de a puede ser 0.

-> b , una cláusula como ésta significa que b siempre es verdadero, a estas cláusulas se les llama hechos.

Las cláusulas Horn pueden utilizarse para expresar la mayoría, pero no la totalidad de los enunciados lógicos.

12.3. Resolución y unificación.

La resolución es una regla de inferencia para cláusulas Horn que tiene una eficiencia especial. Si tenemos dos cláusulas Horn, y podemos parear la cabeza de la primera cláusula Horn con uno de los enunciados en el cuerpo de la segunda cláusula, entonces pueden utilizarse la primer cláusula para reemplazar su cabeza en la segunda utilizando su cuerpo.

b <- a y c <- b, lo que nos da b,c <- a,b y cancelando b c <- a

12.4. El lenguaje Prolog.

Prolog es el lenguaje de programación lógica más utilizado. Prolog utiliza cláusulas Horn e implementa la resolución utilizando una estrategia "primero en profundidad" estrictamente lineal y un algoritmo de unificación.

Notación y estructuras de datos.

Prolog utiliza casi una notación idéntica a la desarrollada anteriormente para las cláusulas Horn, excepto que la flecha de implicación "←" se reemplaza con dos puntos seguidos de un guión :-

Las variables se escriben con mayúsculas y las constantes y los nombres con minúsculas.

Las estructuras básicas de datos son términos como `padre(X,Z),` o bien `sucesor(sucesor(0)).` Prolog también incluye listas como una estructura básica de datos.

Ejecución en Prolog.

Existen compiladores para Prolog, pero la mayoría de los sistemas se ejecutan como intérpretes. Un programa Prolog incluye un juego de cláusulas Horn en la sintaxis de Prolog que generalmente se toma de un archivo y se almacena en una base de datos de cláusulas mantenida dinámicamente.

Aritmética.

Prolog tiene operaciones aritméticas incorporadas así como un evaluador aritmético.

Para obligar a la evaluación de un término aritmético se requiere de una nueva operación : el predicado incorporado is: `X is 3+5, write (X).`

Unificación.

La unificación es el proceso mediante el cual se hace la instanciación de las variables o se les asigna memoria y valores asignados, de manera que coincidan los patrones durante la resolución. La unificación es el proceso para hacer en cierto sentido que dos términos se conviertan en "el mismo".

La expresión básica es la igualdad: `s = t`

Estrategia de búsqueda.

Prolog aplica la resolución de una forma estrictamente lineal, reemplazando metas de izquierda a derecha y considerando las cláusulas en la base de datos en orden descendente, de arriba abajo.

Ciclos y estructuras de control.

Para que Prolog ejecute ciclos y búsquedas repetitivas podemos utilizar búsqueda primero en profundidad con retroceso. Lo que debemos lograr es forzar el retroceso incluso cuando se haya encontrado una solución.

12.5. Problemas que se presentan con la programación lógica.

La meta original de la programación lógica era hacer de la programación una actividad de especificación.

Los lenguajes de programación lógica y Prolog en particular, sólo han llenado esta meta de manera parcial.

El problema de verificación de ocurrencias en la unificación.

El algoritmo de unificación utilizado por Prolog es de hecho erróneo: al unificar una variable con un término. Problema de verificación de ocurrencias.

Negación como fracaso.

Todos los sistemas de programación lógica tienen la propiedad básica de que algo que no pueda ser probado como verdadero se supone que es falso.

Esto se conoce como la suposición del mundo cerrado.

Las cláusula Horn no expresan toda la lógica.

No todo enunciado lógico puede ser convertido a una cláusula Horn. Los enunciados que involucran cuantificadores posiblemente no podrán expresarse en este formato.

Información de control en la programación lógica.

Cut es un mecanismo explícito de control útil, incluso esencial en Prolog.

Pero en razón de su estrategia de búsqueda de primero en profundidad, y de su procesamiento lineal de metas y enunciados, los programas Prolog también contienen información implícita de control que fácilmente puede ser responsable de que fracasen los programas.

12.6. Extensión de la programación lógica: programación lógica con restricciones y sistemas basados en ecuaciones.

Programación lógica con restricciones.

Una de las fallas más importantes de Prolog es su tratamiento especial de la aritmética: no se aplican las operaciones sino hasta que se obligan mediante la operación "is", y el uso de esta operación hace que el programa se comporte más como un programa imperativo que como uno lógico. Por ejemplo, no puede ser ejecutado hacia atrás.

Podría lograrse una gran mejoría en las propiedades y en el comportamiento de los programas que involucran este tipo de cálculos si estos requisitos de instanciación y el operador "is" fueran eliminados totalmente.

Existen un número creciente de lenguajes lógicos que hacen exactamente eso: se conocen como lenguajes de programación lógica con restricciones, lenguajes CLP.

Ej. Prolog IV, CLP(R) y CHIP

Convierten la aritmética y las desigualdades en restricciones que se van acumulando conforme avanza la resolución del programa, y que entonces se resuelven mediante un revolvedor de restricciones.

Sistemas basados en ecuaciones.

La programación lógica puede utilizarse para escribir especificaciones ejecutables para un sistema de programación. Estas especificaciones pueden ser probadas para verificar que no sólo son correctas, sino que representan de manera adecuada los requisitos del usuario.

Los lenguajes de programación lógica pueden ser utilizados como sistemas de prototipazo rápido para el desarrollo de software.