Universidad Tecnológica Nacional
Facultad Regional Córdoba
Dpto de Ingeniería en Sistemas de Información
Cátedra Paradigmas de Programación

UNIDAD Nº6

ELEMENTOS CONSTITUTIVOS DE LENGUAJES Y PARADIGMAS

2021

EL PRESENTE CAPÍTULO HA SIDO ELABORADO, CORREGIDO Y COMPILADO POR:

MGTR. ING. CORSO, CYNTHIA
ESP. ING.GUZMAN, ANALIA
ESP. ING. LIGORRIA, LAURA
DR. ING. MARCISZACK, MARCELO
ESP. ING. TYMOSCHUK, JORGE

ÚLTIMA REVISIÓN: MAYO 2021

ÍNDICE

Objetivos de la Unidad

Que el alumno reconozca los elementos constitutivos, mecanismos y formas de implementación de las características de los lenguajes de programación dentro del Paradigma de programación correspondiente vistos en las unidades temáticas de la asignatura.

Contenidos Abordados

Conceptos lógicos y transversales.
Abstracción: definición y mecanismos de implementación.
Tipos de datos: Teoría, clasificación, verificación, sistema de tipos, conversión y ejemplos en los diferentes lenguajes de programación.
Mecanismos de control de flujo: Organización y ejemplos en los diferentes lenguajes de programación.

1. Conceptos Utilizados: lógicos y transversales

De manera de comprender acabadamente, las estrategias y principios utilizados dentro de los lenguajes de programación, haremos un repaso de los conceptos utilizados como unidades fundamentales dentro de los principios de construcción.

1.1 Abstracción

Es la operación intelectual que ignora selectivamente partes de un todo para facilitar su comprensión. En la resolución de problemas: implica ignorar los detalles específicos buscando generalidades que ofrezcan una perspectiva distinta, más favorable a su resolución. El proceso de abstraerse progresivamente de los detalles, permite manejar **niveles de abstracción.**

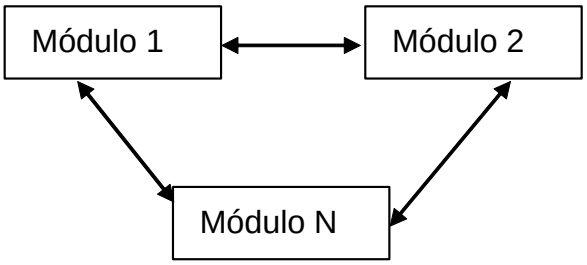
Por ejemplo, en un programa que usa un archivo: En un máximo nivel de abstracción podemos definir:

- Apertura del archivo
- Procesamiento
- Cierre del archivo

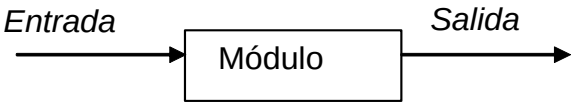
1.2 Modularización, encapsulamiento y delegación

Una estrategia central de la programación es buscar la manera de organizar y distribuir la funcionalidad de un sistema complejo en **unidades más pequeñas de software con se responsabilizan**

de tareas específicas y que interactúan entre ellas. Estas unidades reciben nombres diferentes según cada lenguaje de programación, como rutinas, funciones, procedimientos, métodos, predicados, subprogramas, bloques, entidades, siendo "módulos" una de las más frecuentes y que dan origen al término.



La clave de **cada módulo es que no conoce el funcionamiento interno de los demás módulos** con los que interactúa **sino sólo suinterfaz**, es decir, la forma en que debe enviarle información adicional en forma de parámetros y cómo va a recibir las respuestas. Esta propiedad recibe diversos nombres, como el de **encapsulamiento**, ocultación de información o "caja negra". Ante la modificación de una funcionalidad en particular del sistema, en la medida que su implementación esté encapsulada en un módulo, el impacto que produce su cambio no afectará a los otros módulos que interactúan con él. Facilita el manejode la complejidad, sólo se conoce el comportamiento pero no los detalles internos, nos interesa conocer que hace pero no cómo lo hace.



En concordancia con la distribución de responsabilidades entre las diferentes unidades de software, la **delegación** consiste en la invocación que desde un módulo se efectúa a otro módulo, de manera que el que invoca explicita qué es lo que pretende y el que es invocado se ocupa de todo lo necesario para realizarlo. Puede realizarse de numerosas maneras, variando el criterio de distribución de responsabilidades, el modo de evaluación, la forma de paso de parámetros, los tipos de datos que utiliza, de acuerdo a las posibilidades y restricciones de cada lenguaje y paradigma.

1.3 DECLARATIVIDAD

La declaratividad, en términos generales, se basa en la separación del conocimiento sobre la definición del problema con la forma de buscar su solución, una **separación entre la lógica y el control**.

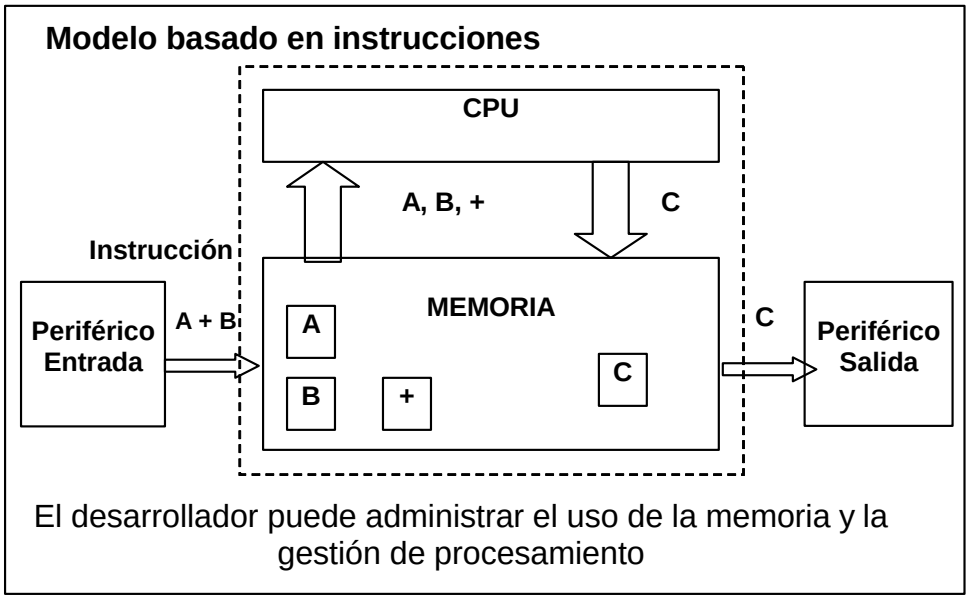
En un programa declarativo se especifican un **conjunto de declaraciones**, que pueden ser proposiciones, condiciones, restricciones, afirmaciones, o ecuaciones, que caracterizan al problema y **describen su solución**. A partir de esta información el sistema utiliza **mecanismos internos de control,** comúnmente llamado **"motores"**,que evalúan y relacionan adecuadamente dichas especificaciones, de manera de obtener la solución. De esta manera, en vez de ser una secuencia de órdenes, un programa es un conjunto de definiciones sobre el dominio del problema.

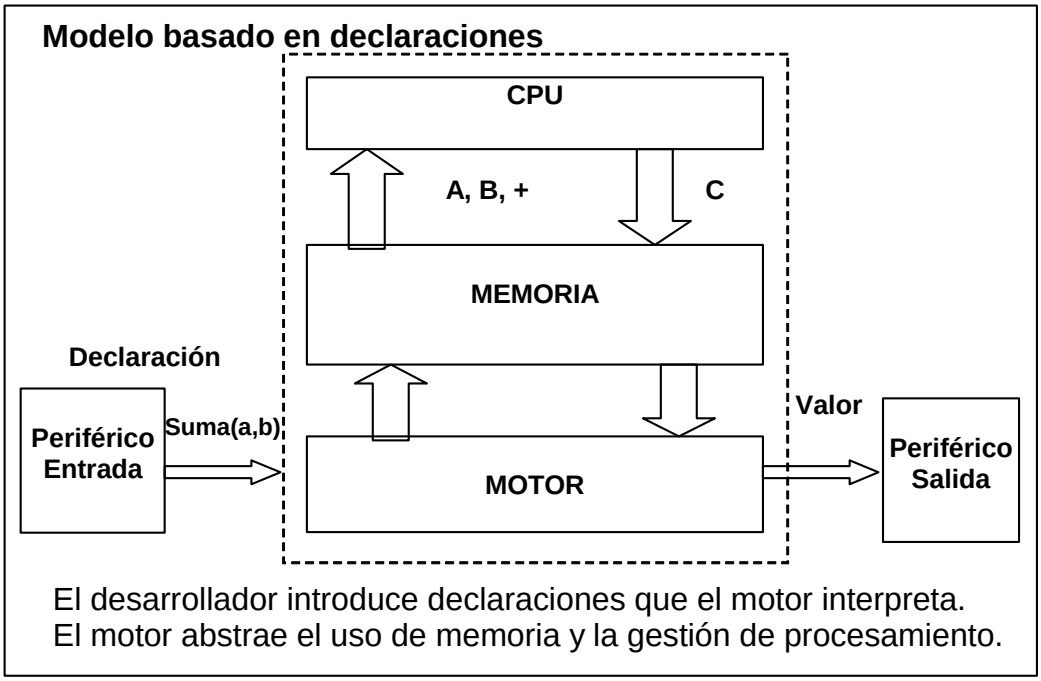
Basándose en la noción de **delegación**, la declaratividad plantea como criterio para distribuir las responsabilidades, separar las relacionadas con modelar o definir el conocimiento del problema de aquellas de manipular ese conocimiento para alcanzar un objetivo concreto. En otras palabras, distinguir el "qué" del "cómo".

La declaratividad brinda la posibilidad de usar una misma descripción en múltiples contextos en forma independiente de los motores que se utilicen. Permite focalizar por un lado en las cuestiones algorítmicas del motor, por ejemplo para trabajar en forma unificada sobre eficiencia, y por otro en la definición del dominio del problema y la funcionalidad de la aplicación en sí.

La noción opuesta, aunque en cierta medida complementaria, de la declaratividad, puede denominarse "**proceduralidad**". Los programas procedurales se construyen indicando explícitamente la secuencia de ejecución en la que se procesan los datos y obtienen los resultados. Para ello se detalla un **conjunto de sentencias**, ordenadas mediante estructuras de control como decisiones, iteraciones y secuencias, que conforman "algoritmos".

En un sistema complejo, no se puede hablar de declaratividad o proceduralidad como conceptos excluyentes o totalizantes, sino que coexisten y se relacionan en una permanente tensión. Dependiendo de los lenguajes y de las herramientas que se utilicen, y en particular del diseño del sistema, habrá partes del sistema que por su sentido o ubicación dentro del sistema global serán más declarativas o procedurales que otras, de manera de aprovechar las ventajas respectivas.





1.4 TIPOS DE DATOS

Un **tipo de dato**, o como también es llamado, un tipo abstracto de dato, es un **conjunto de valores** y de **operaciones asociadas** a ellos.

Los Lenguajes de Programación, dentro de cada Paradigma utilizan diversos tipos de datos, lo cual permite la agrupación o clasificación del gran volumen y variedad de valores que es necesario representar y operar en un sistema, según sus semejanzas y diferencias.

Tomando como criterio las similitudes en cuanto al contenido de lo que representan, una primera condición es la existencia de un conjunto de valores o entidades **homogéneos en su representación**. Otra condición es que los elementos del mencionado conjunto se **comporten en forma uniforme** respecto a una serie de operaciones.

Cada paradigma de programación, y en particular cada lenguaje, tiene su forma de determinar tanto la conformación de cada tipo de dato, con sus valores y operaciones, como la forma en que se relacionan entre sí conformando un **sistema de tipo de datos**.

En un lenguaje **fuertemente tipado**, toda variable y parámetro deben ser definidos de un tipo de dato en particular que se mantiene sin cambios durante la ejecución del programa, mientras que en uno **débilmente tipado**no, sino que pueden asumir valores y tipos de datos diferentes durante la ejecución del programa.

El tipo de dato al que pertenece una entidad determina la operatoria que se puede realizar con ello. Una tarea que realizan muchos de los lenguajes de programación como forma de su mecanismo interno es el **chequeo del tipo de dato** de las entidades del programa. Esta acción se realiza de diferentes maneras, con mayor o menor flexibilidad, y en diferentes momentos, como la compilación o la ejecución del programa, y en otros no se realiza.

De todas maneras, el tipo de datos permite entender qué entidades tienen sentido en un contexto, independientemente de la forma de chequeo o si el tipado es débil o fuerte. Por ejemplo, a un bloque de software que recibe como argumento una variable, conocer de qué tipo de dato es le permite saber qué puede hacer con ella.

1.5 ESTRUCTURAS DE DATOS

Los valores atómicos, es decir, aquellos que no pueden ser descompuestos en otros valores, se representan mediante **tipos de datos simples**.

En contrapartida, en un **tipo de dato compuesto** los valores están compuestos a su vez por otros valores, de manera que conforma una **estructura de datos**. Cada uno de los valores que forman la estructura de datos corresponde a algún tipo de dato que puede ser tanto simple como compuesto. Su utilidad consiste en que se pueden procesar en su conjunto como una unidad o se pueden descomponer en sus partes y tratarlas en forma independiente. En otras palabras, las estructuras de datos son conjuntos de valores.

1.6 POLIMORFISMO Y SOFTWARE GENÉRICO

El polimorfismo, como se vio durante el desarrollo de la asignatura, es un concepto para el que se pueden encontrar numerosos y diferentes definiciones. En un sentido amplio, más allá de las especificidades de cada paradigma y del alcance que se le dé a la definición del concepto desde la perspectiva teórica desde la que se lo aborde, el objetivo general del **polimorfismo** es favorecer a la construcción de piezas de software genéricas que trabajen indistintamente con diferentes tipos de entidades, para otra entidad que requiere interactuar con ellas; en otras palabras, que dichas entidades puedan ser intercambiables. Mirando con mayor detenimiento los mecanismos que se activan y sus consecuencias para el desarrollo de sistemas, se pueden distinguir dos grandes situaciones.

Hay polimorfismo cuando, ante la existencia de dos o más bloques de software con una misma interfaz, otro bloque de software cualquiera puede trabajar indistintamente con ellos. Esta noción rescata la existencia de tantas implementaciones como diferentes tipos de entidades con las que se interactúe.

Una entidad emisora puede interactuar con cualquiera de las otras entidades de acuerdo a las características de la interfaz común, y la entidad receptora realizará la tarea solicitada de acuerdo a la propia implementación que tenga definida, independientemente de las otras implementaciones que tengan las otras entidades. Consistentemente con la noción de delegación, la entidad emisora se desentiende de la forma en que las otras entidades implementaron sus respuestas, ya sea que fuera igual, parecida o totalmente diferente.

Analizando el concepto desde el punto de vista de las entidades que responden a la invocación, el proceso de desarrollo debe contemplar la variedad y especificidad de cada una para responder adecuadamente a lo que se les solicita. Desde el punto de vista de la entidad que invoca, el proceso es transparente, y es en definitiva ésta, la que se ve beneficiada por el uso del concepto.

Otra forma de obtener un bloque de software genérico es ante el caso en que las entidades con las que el bloque quiere comunicarse, en vez de tener diferentes implementaciones, aun siendo de diferente tipo, sean lo suficientemente similares para compartir una misma y única implementación. Desde el punto de vista de la entidad que invoca, el proceso continúa siendo transparente y sigue aprovechando los beneficios de haber delegado la tarea y haberse despreocupado de qué tipo de entidad es la receptora. En este caso, el concepto de polimorfismo se relaciona o se basa en la coexistencia de herencia, de variables de tipo de dato o en las formas débiles de declaración y chequeo de tipos de datos, dependiendo de las diferentes herramientas de cada paradigma.

En la unidad del paradigma con orientación a objetos, se estudiaron las colecciones de SmallTalk, las cuales son polimórficas ya que se definen a través de una jerarquía de clases y de operaciones genéricas asociadas, que pueden almacenar objetos de cualquier clase y que presentan un protocolo unificado, por lo que todas las diferentes formas de colecciones provistas por el entorno responden a un mismo conjunto de mensajes básicos, lo que facilita su aprendizaje y el cambio de un tipo de colección a otra.

1.7 TRANSPARENCIA REFERENCIAL Y EFECTO DE LADO

La **transparencia referencial**como se vio dentro del desarrollo del Paradigma Funcional, consiste en que el valor de una expresión depende únicamente del valor de sus componentes, de manera que siempre que se evalúa el mismo bloque de software con los mismos parámetros, se obtiene el mismo resultado, sin importar el comportamiento interno de dicho bloque. Su evaluación no produce un cambio en el estado de información del sistema que pueda afectar una posterior evaluación del mismo u otro bloque.

Entre las ventajas, la transparencia referencial da robustez, ya que se garantiza que el agregado de nuevas unidades de software no interfiere con las existentes generando algún efecto colateral. Los tests son más confiables, ya que toda consulta o evaluación, responde siempre de la misma forma. El conjunto de variables o condiciones iniciales a tener en cuenta está acotado a la misma consulta. Los errores tienden a ser más locales y a no propagarse por todo el sistema. Aporta independencia entre las unidades de software, ya que trabajan sobre valores diferenciados.

Ejemplo: Función con Transparencia Referencial

Definición:

```
int sumaUno(int x)
{
   return (x + 1);
}
```

Invocación:

```
sumaUno(6) = 7;
sumaUno (4) = 5;
```

El **efecto de lado** (*sideeffect*), también traducido como **efecto colateral**, se produce cuando el resultado de la evaluación de un bloque de software depende de otros valores o condiciones del ambiente más allá de sus parámetros. Su evaluación incluye acciones que afectan un estado de información que sobrevive a la evaluación del bloque, por lo que influye en una posterior evaluación del mismo u otro bloque.

La utilización del efecto de lado representa mejor la dinámica del sistema, ya que cuando hay un **estado** de información que mantener actualizado, **es un efecto deseado que el funcionamiento del sistema provoque cambios en la información**. Refleja la interacción de diferentes partes del sistema, ya que una unidad de software puede realizar cambios en la información que incidan en el funcionamiento de otra unidad.

Ejemplo: Función con Efecto de Lado

Definición:

```
int var_global = 0;

int sumaValor(int x)
{
  var_global = var_global + 1;
  return (x + var_global);
}
```

Invocación:

```
sumaValor (6) = 7;
sumaValor (4) = 6;
var_global = 9;
sumaValor (2) =12;
```

1.8 ASIGNACIÓN Y UNIFICACIÓN

La **asignación destructiva** es una operación que consiste en cambiar la información representada por una **variable**, de forma tal que, si se consulta su valor antes y después de dicha operación, se obtiene un resultado distinto. Estas asignaciones se realizan repetitivamente sobre la misma celda de memoria, remplazando los valores anteriores. La asignación determina el **estado** de una variable, que consiste en el valor que contiene en un momento en particular. La asignación destructiva es la forma más usual de provocar efecto de lado, pero no la única, ya que, dependiendo de los lenguajes, hay otro tipo de instrucciones que también permiten modificar el estado de información de un sistema.

La **unificación** es un mecanismo por el cual una variable que no tiene valor, asume un valor. Una vez unificada, o "ligada", como también se le dice, una variable no cambia su valor, por lo que no existe la noción de estado. La duración de la unificación está condicionada por el alcance que tienen las variables en cada lenguaje en particular, pudiendo una variable unificarse con varios valores alternativos en diferentes momentos de la ejecución del bloque de software en el que se encuentran. Se suele denominar como "indeterminada" a una variable sin ligar o no unificada. La unificación se encuentra íntimamente relacionado con el mecanismo de "**encaje de patrones**" (*patternmatching*)**.**

1.9 MODO DE EVALUACIÓN

Los parámetros que se utilizan en la invocación de cualquier bloque de software pueden ser evaluados en diferentes momentos de acuerdo al modo de evaluación que utilice el lenguaje de programación.

La **evaluación ansiosa** consiste en que los argumentos son evaluados antes de invocar al bloque de software y es responsabilidad de la entidad que invoca.

La **evaluación diferida** plantea que la evaluación de los argumentos es responsabilidad del bloque de software invocado, quien decide el momento en que lo hará. Provoca que se difiera la evaluación de una expresión, permitiendo que en algunos casos, de acuerdo a cómo sea la implementación, no sea necesario evaluarla nunca, con el consiguiente beneficio en términos de eficiencia.

La evaluación diferida es utilizada en cierto tipo de situaciones que serían consideradas erróneas o imposibles de resolver con la evaluación ansiosa, como por ejemplo los bucles o las listas infinitas.

1.10 ORDEN SUPERIOR

El poder que brinda esta generalidad puede abrir posibilidades en diferentes contextos. La idea general se explica con la siguiente definición: Los programas pueden tener como argumento Programas y producir como resultado otros Programas.

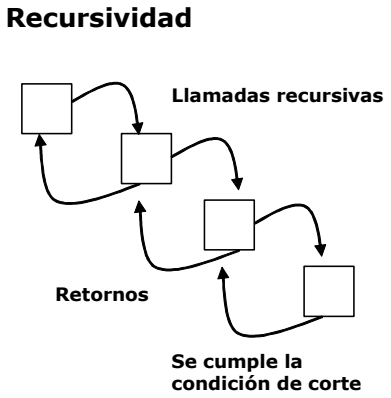
Según el paradigma, también podemos definirlo de las siguientes formas:

- **En la programación imperativa**: Los procedimientos son tratados como datos y en consecuencia pueden ser utilizados como parámetros, representados en variables, devueltas como resultados. Se denomina de orden superior a los procedimientos que reciben como argumentos otros procedimientos.

- **En la programación Funcional**: el concepto de funciones de orden superior se refiere al uso de funciones como si de un valor cualquiera se tratara, posibilitando pasar funciones como parámetros de otras funciones o devolver funciones como valor de retorno.

1.11 RECURSIVIDAD

La recursividad, entendida como iteración con asignación no destructiva, está relacionada con el principio de **inducción**. En general, un bloque de software recursivo se define con al menos un término recursivo, en el que se vuelve a invocar el bloque que se está definiendo, y algún término no recursivo como caso base para detener la recursividad.



2. ABSTRACCIÓN

Este concepto fundamental, tal como ha sido abordado, es la operación intelectual que ignora selectivamente partes de un todo para facilitar su comprensión. En la resolución de problemas: implica ignorar los detalles específicos buscando generalidades que ofrezcan una perspectiva distinta, más favorable a su resolución.

El proceso de abstraerse progresivamente de los detalles y así manejar **niveles de abstracción**, es el que permite construir sistemas complejos. Las unidades de software que realizan la funcionalidad del sistema requieren también de convenciones y criterios de ordenamiento y articulación interna tanto para un funcionamiento confiable y eficiente del sistema, como para que el proceso de construcción y mantenimiento del software sea de la forma lo más simple posible.

2.1 EL PROGRESO DE LA ABSTRACCIÓN

Todos los lenguajes de programación proporcionan abstracciones. Puede argumentarse que la complejidad de los problemas que sea capaz de resolver está directamente relacionada con el tipo (clase) y la calidad de las abstracciones, entendiendo por "clase", "¿qué es lo que se va a abstraer?". El lenguaje ensamblador es una pequeña abstracción de la máquina subyacente.
Muchos de los lenguajes denominados "imperativos" que le siguieron (como FORTRAN, BASIC y C) fueron abstracciones del lenguaje ensamblador. Estos lenguajes constituyen grandes mejoras sobre el lenguaje ensamblador, pero su principal abstracción requiere que se piense en términos de la estructura de la computadora en lugar de la estructura del problema que se está intentado resolver. El programador debe establecer la asociación entre el modelo de la máquina (en el "espacio de la solución", que es donde se va a implementar dicha solución, como puede ser una computadora) y el

modelo del problema que es lo que realmente se quiere resolver (en el "espacio del problema", que es el lugar donde existe el problema, como por ejemplo en un negocio). El esfuerzo que se requiere para establecer esta correspondencia y el hecho de que sea extrínseco al lenguaje de programación, da lugar a programas que son difíciles de escribir y caros de mantener, además del efecto colateral de toda una industria de "métodos de programación".

La alternativa a modelar la maquina es modelar el problema que se está intentado solucionar. Los primeros lenguajes como LISP y APL eligen vistas parciales del mundo ("todos los problemas pueden reducirse a listas" o "todos los problemas son algorítmicos", respectivamente). Prolog convierte todos los problemas en cadenas de decisión.

Cada uno de estos métodos puede ser una buena solución para resolver la clase de problema concreto para el que están diseñados, pero cuando se aplican en otro dominio resultan inadecuados.

El enfoque orientado a objetos trata de ir un paso más allá proporcionando herramientas al programador para representar los elementos en el espacio del Problema. Esta representación es tan general que el programador no está restringido a ningún tipo de problema en particular. Se hace referencia a los elementos en el espacio del problema denominando "objetos" a sus representaciones en el espacio de la solución (también se necesitarán otros objetos que no tendrán análogos en el espacio del problema). La idea es que el programa pueda adaptarse por sí sólo a la jerga del problema añadiendo nuevos tipos de objetos, de modo que cuando se lea el código que describe la solución, se estén leyendo palabras que también expresen el problema. Ésta es una abstracción del lenguaje más flexible y potente que cualquiera de las que se hayan hecho anteriormente'. Por tanto, la programación orientada a objetos permite describir el problema en términos del problema en lugar de en términos de la computadora en la que se ejecutará la solución.

Alan Kay resumió las cinco características básicas del Smalltalk, el primer lenguaje orientado a objetos que tuvo éxito y uno de los lenguajes en los que se basa Java. Estas características representan un enfoque puro de la programación orientada a objetos.

Todo es un objeto, piense en un objeto como en una variable: almacena datos, permite que se le "planteen solicitudes", pidiéndole que realice operaciones sobre sí mismo. En teoría, puede tomarse cualquier componente conceptual del problema que se está intentado resolver (personas, edificios servicios, etc.) y representarse como un objeto del programa.

Un programa está formado por muchos objetos que se dicen entre sí lo que tienen que hacer enviándose mensajes. Para hacer una solicitud a un objeto, hay que enviar un mensaje a dicho objeto. Más concretamente, puede pensar en que un mensaje es una solicitud para llamar a un método que pertenece a un determinado objeto.

Cada objeto tiene su propia memoria formada por otros objetos, un objeto se puede crear a partir de objetos existentes y puede contener otros objetos, por lo tanto, se puede incrementar la complejidad de un programa ocultándola tras la simplicidad de los objetos.

Todo objeto tiene un tipo asociado. Como se dice popularmente, cada objeto es una instancia de una clase, siendo "clase" sinónimo de "tipo". La característica distintiva más importante de una clase es "el conjunto de mensajes que se le pueden enviar".

Todos los objetos de un tipo particular pueden recibir los mismos mensajes. Como veremos más adelante, esta afirmación es realmente importante. Puesto que un objeto de tipo "Circulo" también es un objeto de tipo "Figura", puede garantizarse que un círculo aceptará los mensajes de Figura. Esto quiere decir que se puede escribir código para comunicarse con objetos de tipo Figura y controlar automáticamente cualquier cosa que se ajuste a la descripción de una Figura. Esta capacidad de suplantaciónes uno de los conceptos más importantes de la programación orientada a objetos.

Booch ofrece una descripción aún más sucinta de objeto:

Un objeto tiene estado, comportamiento e identidad.

Un objeto puede tener datos internos (lo que le proporciona el estado), métodos (para proporcionar un comportamiento) y que cada objeto puede ser diferenciado de forma unívoca de cualquier otro objeto; es decir, cada objeto tiene una dirección de memoria exclusiva.

Todo objeto tiene una interfaz

Aristóteles fue probablemente el primero en estudiar cuidadosamente el concepto de tipo; hablaba de "la clase de peces y de la clase de pájaros". La idea de que todos los objetos, aún siendo únicos, son también parte de una clase de objetos que tienen características y comportamientos comunes ya se empleó en el primer lenguaje orientado a objetos, el Simula-67, que usaba su palabra clave fundamental class y que permite introducir un nuevo tipo en un programa.

Simula, como su nombre implica, se creó para desarrollar simulaciones como la clásica del "problema del cajero de un banco". En esta simulación, se tienen muchos cajeros, clientes, cuentas, transacciones y unidades monetarias, muchísimos "objetos". Los objetos, que son idénticos excepto por su estado durante la ejecución de un programa, se agrupan en "clases de objetos", que es de donde procede la palabra clave class. La creación de tipos de datos abstractos (clases) es un concepto fundamental en la programación orientada a objetos. Los tipos de datos abstractos funcionan casi exactamente como tipos predefinidos: pueden crearse variables de un tipo (llamadas objetos u instancias en la jerga de la POO) y manipular dichas variables (mediante el envío de mensajes o solicitudes, se envía un mensaje y el objeto sabe lo que tiene que hacer con él).

Los miembros (elementos) de cada clase comparten algunos rasgos comunes. Cada cuenta tiene asociado un saldo, cada cajero puede aceptar un depósito, etc. Además, cada miembro tiene su propio estado. Cada cuenta tiene un saldo diferente y cada cajero tiene un nombre. Por tanto, los cajeros, clientes, cuentas, transacciones, etc., pueden representarse mediante una entidad unívoca en el programa informático. Esta entidad es el objeto y cada objeto pertenece a una determinada clase que define sus características y comportamientos.

Por tanto, aunque en la programación orientada a objetos lo que realmente se hace es crear nuevos tipos de datos, en la práctica, todos los lenguajes de programación orientada a objetos utilizan la palabra clave "class". Cuando vea la palabra "type" (tipo) piense en "class" (clase), y viceversa'

Dado que una clase describe un conjunto de objetos que tienen características (elementos de datos) y comportamientos (funcionalidad) idénticos, una clase realmente es un tipo de datos porque, por ejemplo, un número en coma flotante también tiene un conjunto de características y comportamientos. La diferencia está en que el programador define un clase para adaptar un problema en lugar de forzar el uso de un tipo de datos existente que fue diseñado para representar una unidad de almacenamiento en una máquina. Se puede ampliar el lenguaje de programación añadiendo nuevos tipos de datos específicos que se adapten a sus necesidades. El sistema de programación admite las nuevas clases y proporciona a todas ellas las comprobaciones de tipo que proporciona a los tipos predefinidos.

Una vez que se ha definido una clase, se pueden crear tantos objetos de dicha clase como se desee y dichos objetos pueden manipularse como si fueran los elementos del problema que se está intentado resolver. Realmente, uno de los retos de la programación orientada a objetos es crear una correspondencia uno-a-uno entre los elementos del espacio del problema y los objetos del espacio de la solución.

Por ejemplo:

Vida real	clase	objetos

Necesito conocer la condición de regularidad de un alumno. **Como sería el proceso de abstracción?**

Alumno

Propiedades
legajo
asignatura
nota1
nota2

Operaciones
conocerLegajo....
conocerNota1
conocerNota2
conocerPromedio

A continuación podemos ejemplificar de manera general, como se produce la abstracción en los distintos paradigmas:

- En el paradigma imperativo la principal abstracción requiere que se piense en términos de la estructura de la computadora, en como traducir un problema en instrucciones de computadora.
- Tanto en el paradigma Orientado a objetos como en los declarativos se permiten describir el problema en términos del problema en lugar de términos de la computadora. La abstracción se realiza:
    - En POO con objetos o entidades.
    - En P. Funcional con funciones matemáticas.
    - En P. Lógico con proposiciones lógicas.

2.2 TIPOS DE ABSTRACCIÓN

Entendiendo un sistema como una **abstracción de la realidad**, los **datos** son las entidades que representan cada uno de los aspectos de la realidad que son significativos para el funcionamiento del sistema. Para que tenga sentido como abstracción de la realidad, cada dato implica un determinado **valor** y requiere de una convención que permita representarlo sin ambigüedad y procesarlo de una manera confiable.

En la misma línea, la **lógica del procesamiento** de los datos es también una abstracción de los procesos que suceden en la realidad que conforma el dominio de la aplicación.

De estas ideas surgen tres tipos de abstracciones:

- **Abstracción de Datos (TDA**): Tenemos un conjunto de datos y un conjunto de operaciones que caracterizan el comportamiento del conjunto. Las operaciones están vinculadas a los datos del tipo.
- **Abstracción Procedimental**: Definimos un conjunto de operaciones (procedimiento) que se comportan como una operación.
- **Abstracción de Iteración:** Abstracción que permite trabajar sobre colecciones de objetos sin tener que preocuparse por la forma concreta en que se organizan.

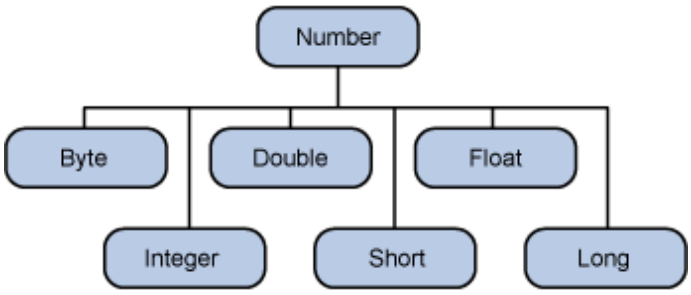
2.2.1 ABSTRACCIÓNDE DATOS

La mayoría de los lenguajes de programación enfatizan las acciones. En estos lenguajes, los datos existen para apoyar las acciones que los programas deben realizar. Los datos son "menos interesantes" que las acciones. Los datos son "crudos". Sólo existen unos cuantos tipos primitivos, y es difícil para los programadores crear sus propios tipos. Java y el estilo orientado a objetos de programación elevan la importancia de los datos. Las principales actividades de la programación orientada a objetos en Java son la creación de tipos (por ejemplo, clases) y la expresión de las interacciones entre objetos de esos tipos. Para crear lenguajes que enfaticen los datos, la comunidad de lenguajes de programación necesitaba formalizar ciertas nociones sobre los datos.

La formalización que consideramos aquí es la noción de tipos de datos abstractos (ADTs), los cuales mejoran el proceso de desarrollo de software.

Ejemplos con el lenguaje Java

Los programadores en Java utilizan clases para implementar tipos de datos abstractos. La plataforma java provee clases para cada uno de los tipos de datos primitivos. Estas clases encapsulan las operaciones posibles de los tipos primitivos en un objeto. Todas las clases numéricas son subclases de la clase abstracta Number:



SUBCLASES DE LA CLASE ABSTRACTA NUMBER

Hay tres razones por las que podría usar un objeto Number en lugar de un primitivo:

1. Como un argumento de un método que espera un objeto (a menudo usado cuando se manipulan colecciones de números).
2. Para usar las constantes definidas por la clase, tales como MIN_VALUE y MAX_VALUE, que proveen límites superiores e inferiores del tipo de datos.
3. Para usar métodos de clase para conservar valores hacia y desde otros tipos de datos primitivos, para convertir hacia y desde cadenas, y para convertir entre sistemas de numeración (decimal, octal, hexadecimal, binario).

3. Tipos de Datos

Los lenguajes de programación, para poder cumplir su cometido, que esen definitiva, el de facilitar las herramientas para modelar e implementar una solución a un problema planteado, necesita imperiosamente ordenar y sistematizar la información. Esto se logra a través de lo que se denomina como Teoría de Tipos de Datos, que en si es un complejo sistema en donde se definen como serán manipulados y operados los datos y variables de nuestro programa.

Los Paradigmas y lenguajes vistos en unidades anteriores, se apoyan en diferentes construcciones de los mismos, los cuales los formalizaremos a continuación.

3.1 TEORÍA DE VALORES Y TIPOS

Los valores o datos organizados proveen información, todos los sistemas de información tienen como objetivo general el de hacer algo requerido por los usuarios con esta información. Por lo tanto, el estudio de los valores o datos es imprescindible dentro del área de las ciencias de la computación.

3.1.1 VALOR

Nuestro mundo gira en torno a valores que adquieren sentido cuando están organizados, por ejemplo: ``María'', 20, 23452, ``Rojas'', 19, 545632, son valores distintos que al parecer organizados cobrarían sentido. A continuación, se definirá este término dentro del contexto de nuestro estudio.
Definición: *Un valor es cualquier entidad que puede ser evaluada, almacenada en una estructura de datos, pasada como un argumento a una función o procedimiento, retornado como un resultado de una función, etc. Cualquier entidad que existe durante una computación.*
Todos los lenguajes de programación involucran el uso de valores. Porejemploen Haskell:

- Valores primitivos (valores de verdad, enteros, reales, etc.).
- Valores compuestos (tuplas, construcciones, cadenas, listas y abstracciones de funciones).

3.1.2 TIPO

Es útil agrupar los valores en tipos. Por ejemplo: los valores de verdad se distinguen claramente de los valores enteros, puesto que existen operaciones propias para los valores de verdad que no son aplicables a los valores enteros y también viceversa.
Definición: Un *tipo es un conjunto de valores que tienen características en común y exhiben un comportamiento uniforme bajo ciertas operaciones.*
Es importante recalcar que todo el conjunto de valores en un tipo exhibe un comportamiento uniforme bajo las operaciones asociadas con el tipo. Así el conjunto {13,'e', true}no es un tipo; pero {false, true}es un tipo porque sus valores exhiben un comportamiento uniforme bajo las operaciones de negación, conjunción y disyunción.

El tipo de una expresión nos indica los valores que ésta puede representar y las operaciones que pueden aplicarse a ellas. Es posible sumar los enteros, pero no los valores booleanos verdadero (true) y falso (false). De este modo, el operador + puede aplicarse a dos expresiones de tipo entero, pero no a dos expresiones de tipo booleano.
Un principio del diseño de lenguajes de uso muy extendido es que toda expresión debe tener un tipo único. Según este principio, los tipos constituyen un mecanismo para clasificar expresiones.

En ciencias de la computación, un **sistema de tipos** define como un lenguaje de programación clasifica los valores y las expresiones en **tipos**, cómo se pueden manipular estos tipos y cómo interactúan. Un tipo indica un conjunto de valores que tienen el mismo significado genérico o propósito (aunque algunos tipos, como los tipos de datos abstractos y tipos de datos función, tal vez no representen valores en el programa que se está ejecutando). Los sistemas de tipificación varían significativamente entre lenguajes, siendo quizás las más importantes variaciones las que estén en sus implementaciones de la sintáctica en tiempo de compilación y la operativa en tiempo de ejecución.

Un compilador puede usar el tipo estático de un valor para optimizar el almacenamiento que necesita y la elección de los algoritmos para las operaciones sobre ese valor. Por ejemplo, para tipo float el lenguaje C usa operaciones específicas de coma flotante sobre estos valores (suma de coma flotante, multiplicación, etc.).

El rango del tipo de dato limita y la forma de su evaluación afecta en el "tipado" del lenguaje. Además, un lenguaje de programación puede asociar una operación concreta con diferentes algoritmos para cada tipo de dato en el caso del polimorfismo. La teoría de tipos de datos es el estudio de los sistemas de tipificación, aunque los sistemas de tipos de datos concretos de los lenguajes de programación se originaron a partir de los problemas técnicos de las arquitecturas del ordenador, implementación del compilador y diseño del lenguaje.

3.2 CLASIFICACIÓN DE LOS TIPOS

Todo lenguaje de programación tiene tipos primitivos cuyos valores son atómicos y tipos compuestos cuyos valores están compuestos a partir de otros valores más simples.

Tipos primitivos

Un tipo primitivo es aquel cuyos valores son atómicos y por lo tanto no pueden ser descompuestos en valores más simples.

Lenguaje	Sintaxis	Conjunto de Valores	Notación
Java, C++	Boolean	{false, true}	Valor- Verdad
Haskell	Bool	{false, true}	Valor- Verdad
C++	Int	{... -2, -1, 0, 1, 2 ..}	Entero
Haskell	Int, Integer	{... -2, -1, 0, 1, 2 ...}	Entero
Java, C++	Float	{..,-1.0,.. 0.0, .. ,..1.0, ...}	Real

EJEMPLOS DE TIPOS PRIMITIVOS EN DISTINTOS LENGUAJES

Las opciones de tipos primitivos en un lenguaje de programación nos dicen mucho acerca del área de aplicación predeterminada por el lenguaje. Un lenguaje predeterminado para el área de comercio (por ejemplo Cobol) provee de tipos primitivos cuyos valores son cadenas de longitud fija y los números de punto fijo. Un lenguaje predeterminado para la computación numérica (por ejemplo Fortran) es probable que tenga tipos primitivos cuyos valores sean números reales (con opción de precisión) y tal vez números complejos.

Tipos primitivos similares frecuentemente ofrecen lo mismo pero con distintos nombres en los distintos lenguajes de programación. Por lo tanto, se utilizara una notación común (lenguaje natural en español) para referirse a estos tipos, ver el cuadro.

En algunos lenguajes el programador puede definir completamente un tipo primitivo nombrando sus valores. Tal tipo es llamado tipo *enumerado*, sus valores que se denominan y se listan de manera explícita.

Ejemplo: *Considerar la definición de tipos enumerados en C:*
enumMes {ene, feb, mar, abr, may, jun ,jul, ago, sep, oct, nov, dic}

Un tipo *primitivo discreto* es un tipo cuyos valores tienen relaciones uno-a-uno con un rango de enteros. Esto es un concepto importante en Pascal y Ada, en los cuales los valores de cualquier tipo primitivo discreto pueden ser usados en una variedad de operaciones, tales como conteo, selección de casos e indexado de arreglos.

Tipos compuestos

Un tipo compuesto (o tipo estructurado de datos) es un tipo cuyos valores son compuestos o estructurados a partir de otros valores simples. Los lenguajes de programación soportan una variedad amplia de estructuras de datos, por ejemplo: tuplas, registros, variantes, uniones, arreglos, conjuntos, cadenas, listas, árboles, archivos seriales, archivos directos, relaciones, etc.

Caso particular: Cadenas

Una cadena es una secuencia de caracteres. Las cadenas son soportadas por todos los lenguajes de programación, sin embargo, no hay un consenso sobre cuál sería la clasificación de estas. Los puntos de discusión son los siguientes:

1. ¿Deberían las cadenas ser valores primitivos o compuestos?
2. ¿Qué operaciones deberían proveerse? Típicamente las operaciones de cadena son prueba de igualdad, concatenación, selección de caracteres o subcadenas y el ordenamiento lexicográfico. La forma como se llevaran a cabo estas operaciones dependerán si las cadenas con consideradas como: arreglos, listas o incluso clases.

Una posibilidad es hacer que la cadena sea un valor primitivo como en ML o Snobol, con valores que son cadenas de cualquier longitud.

Otra propuesta es definir a las cadenas como arreglos de caracteres, entonces todas las operaciones sobre los arreglos están automáticamente permitidas para las cadenas, con todas las consecuencias que esto conlleva, por ejemplo en Pascal.

En Java las cadenas son clases, donde las operaciones permisibles sobre las cadenas son métodos.

En otros lenguajes las cadenas son secuencias de caracteres (listas) como en Prolog y Haskell.

3.3 VERIFICACIÓN DE TIPOS

El conocimiento de la clasificación de los valores en tipos permite al programador utilizar a los datos efectivamente. Una disciplina en la asignación de los tipos evita la ejecución de operaciones sin sentido en los programas, por ejemplo: multiplicar un valor de verdad con un carácter.

Para asegurarse que las operaciones no deseadas sean prevenidas, la implementación del lenguaje debe hacer la verificación de tipos sobre los operandos. Por ejemplo, antes que una multiplicación sea realizada, ambos operandos deben ser verificados para asegurarse que son números.

En el caso de tipos compuestos, se deben considerar tanto la forma de la composición y los tipos de los componentes individuales.

Las reglas de un sistema de tipos especifican el uso apropiado de cada operador del lenguaje. La verificación de tipos asegura que las operaciones de un programa se apliquen de manera apropiada.

El propósito de la verificación de tipos es prevenir errores. Durante la ejecución de un programa, puede ocurrir un error si se aplica una operación de manera incorrecta, por ejemplo si un entero se interpreta erróneamente como cualquier otra cosa. De manera más precisa, ocurre un error de tipo si una función f de tipo S -> T se aplica a alguna a que no sea de tipo S. Se dice que un programa que ejecuta sin errores de tipo tiene seguridad de tipos.

En la implementación de los lenguajes, existe un grado importante de libertad en la verificación de tipos, esta verificación puede ser realizada en tiempo de compilación o en tiempo de ejecución.

Un compilador debe comprobar si el programa fuente sigue tanto las convenciones sintácticas como las semánticas del lenguaje fuente. Esta comprobación, llamada comprobación estática (para distinguirla de la comprobación dinámica que se realiza durante la ejecución del programa objeto), garantiza la detección y comunicación de algunas clases de errores de programación. Los ejemplos de comprobación estática incluyen:

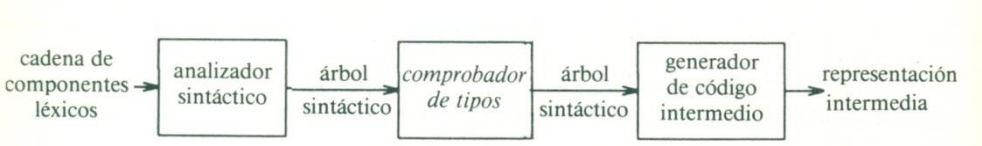
Comprobaciones de tipos. Un compilador debe informar de un error si se aplica un operador a un operando incompatible; por ejemplo, si se suman una variable tipo matriz y una variable de función.

Comprobaciones del flujo del control. Las proposiciones que hacen que el flujo del control abandone una construcción deben tener algún lugar a dónde transferir el flujo de control. Por ejemplo, una proposición break en C o Java hace que el control abandone la proposición que la engloba, while, for o switch más cercana; si dicha proposición englobadora no existe, ocurre un error.

Comprobaciones de unicidad. Hay situaciones en que se debe definir un objeto una vez exactamente. Por ejemplo, en Pascal, un identificador debe declararse de forma única, las etiquetas en una proposición case deben ser diferentes y no se pueden repetir los elementos en un tipo escalar.

Comprobaciones relacionadas con nombres. En ocasiones, el mismo nombre debe aparecer dos o más veces. Por ejemplo, en Ada, un lazo o bloque puede tener un nombre que aparezca al principio y al final de la construcción. El compilador debe comprobar que se utilice el mismo nombre en ambos sitios.

En este capítulo, se trata principalmente la comprobación de tipos. La mayoría de las otras comprobaciones estáticas son rutinarias y se pueden aplicar utilizando las técnicas dellos conceptos vistos en este capítulo. Algunas de ellas pueden formar parte de otras actividades. Por ejemplo, conforme se introduce información acerca de un nombre en una tabla de símbolos, se puede comprobar que el nombre esté declarado de una única manera. Muchos compiladores de Pascal combinan la comprobación estática y la generación de código intermedio con el análisis sintáctico. En construcciones más complejas, como las de Ada, puede ser conveniente tener una pasada de comprobación de tipos independiente entre el análisis sintáctico y la generación de código intermedio, como se indica en la figura siguiente



UBICACIÓN DE UN CONTROLADOR DE TIPOS

Un comprobador de tipos se asegura de que el tipo de una construcción coincida con el previsto en su contexto. Por ejemplo, el operador aritmético predefinido moden Pascal exige operandos de tipo entero, de modo que un comprobador de tipos debe asegurarse de que los operandos de mod tengan tipo entero. De igual manera, el comprobador de tipos debe asegurarse de que la desreferenciación se aplique sólo a un apuntador, de que la indización se haga sólo sobre una matriz, de que una función definida por el usuario se aplique al número y tipo correctos de argumentos, etcétera. Un poco más adelante veremos la especificación de un comprobador de tipos simple. Luego vemos la representación de los tipos y la cuestión de cuándo concuerdan dos tipos.

Puede necesitarse la información sobre los tipos reunida por un comprobador de tipos cuando se genera el código. Por ejemplo, los operadores aritméticos como + normalmente se aplican tanto a enteros como a reales, tal vez a otros tipos, y se debe examinar el contexto de + para determinar el sentido que se pretende dar. Se dice que un símbolo que puede representar diferentes operaciones en diferentes contextos está "sobrecargado". La sobrecarga puede ir acompañada de coacción de tipos, donde un compilador proporciona un operador para convertir un operando en el tipo esperado por el contexto.

Lenguajes de tipificación estática

En un lenguaje de tipificación estática, cada variable y parámetro tiene un tipo fijo que es elegido por el programador, así el tipo de cada expresión y cada operación puede ser deducido con la verificación de tipos en tiempo de compilación.

Los programas se verifican estáticamente, hasta donde es posible, sólo una vez durante la traducción del texto fuente. Por ejemplo, un compilador de Fortran puede ver en el texto fuente que + se aplicará a un par de enteros cada vez que la expresión I+J se evalúe.

Ejemplos de lenguajes que usan tipado estático son C, C++, Java y Haskell. Comparado con el tipado dinámico, el estático permite que los errores de programación sean detectados antes, y que la ejecución del programa sea más eficiente.

Lenguajes de tipificación dinámica

En un lenguaje de tipificación dinámica, solo los valores tienen tipos fijos. Una variable o parámetro no tiene un tipo designado, pero puede tomar valores de diferentes tipos en tiempo de ejecución. Esto implica que la verificación de tipos de los operandos debe hacerse inmediatamente antes de realizar una operación en tiempo de corrida. Lisp, Smalltalk, Clipper, todos los xBase son ejemplos de lenguajes dinámicamente tipados.

Más ejemplos de lenguajes que usan tipado dinámico son Perl, Python y Lisp.

Tipificación estática frente a Tipificación dinámica

Debido a la verificación implícita en tiempo de corrida, la tipificación dinámica conlleva una ejecución más lenta de los programas. Además, para hacer posible la verificación de tipos es necesario que cada valor deba ser etiquetado con su tipo, lo que supone el uso de espacio de memoria adicional.

El tiempo y el espacio adicional son evitados en los lenguajes de verificación de tipos estática, porque toda la verificación de tipos se la hace en tiempo de compilación. Una ventaja muy importante del tipificado estático es la seguridad que garantiza el compilador al detectar los errores de tipos. Esto es importante porque muchos de los errores de tipos en la programación son una proporción importante de errores en la programación.

La ventaja de la verificación de tipos dinámica es la flexibilidad, es decir, la re-utilización de código, característica muy importante en estos días.

La verificación estática es tan efectiva y la dinámica tan cara que las implantaciones de lenguaje a menudo verifican sólo aquellas propiedades que pueden verificarse estáticamente en el texto fuente. Las propiedades que depende de valores calculados en tiempo de ejecución, como la división entre cero o los índices de arreglo que se encuentran dentro de los límites, se verifican muy rara vez.

Los términos estricto y no estricto se refieren a la efectividad con la cual un sistema de tipos evita errores. Un sistema de tipos es estricto si acepta sólo expresiones seguras.

Algunos lenguajes estáticamente tipados tienen una "puerta trasera" en el lenguaje que permite a los programadores escribir código que no es chequeado estáticamente. Por ejemplo, los lenguajes como Java y los parecidos al C tienen una "conversión de tipos de datos forzada (cast)"; estas operaciones pueden ser inseguras en tiempo de ejecución, por que pueden causar comportamientos indeseados cuando el programa se ejecuta.

La presencia de un tipado estático en un lenguaje de programación no implica necesariamente la ausencia de mecanismos de tipado dinámico. Por ejemplo, Java usa tipado estático, pero ciertas operaciones requieren el soporte de test de tipos de datos en tiempo de ejecución, que es una forma de tipado dinámico.

Chequeo de tipificación estático y dinámico en la práctica

La elección entre sistemas de tipificación dinámico y estático requiere algunas contra prestaciones.

El tipado estático busca errores en los tipos de datos en tiempo de compilación. Esto debería incrementar la fiabilidad de los programas procesados. Sin embargo, los programadores, normalmente, están en desacuerdo en cómo los errores de tipos de datos más comunes ocurren, y en qué proporción de estos errores que se han escrito podrían haberse cazado con un tipado estático. El tipado estático aboga por la creencia de que los programas son más fiables cuando son chequeados sus tipos de datos, mientras que el tipado dinámico apunta al código distribuido que se ha probado que es fiable y un conjunto pequeño de errores. El valor del tipado estático, entonces, es que se incrementa a la par que se endurece el sistema de tipificación. Los defensores de los lenguajes fuertemente tipados como ML y Haskell han sugerido que casi todos los errores pueden ser considerados errores de los tipos de datos, si los tipos de datos usados en un programa están suficientemente bien declarados por el programador o inferidos por el compilador.

El tipado estático resulta, normalmente, en un código compilado que se ejecuta más rápidamente. Cuando el compilador conoce los tipos de datos exactos que están en uso, puede producir código máquina optimizado. Además, los compiladores en los lenguajes de tipado estático pueden encontrar atajos más fácilmente. Algunos lenguajes de tipificación dinámica como el lisp permiten declaraciones

de tipos de datos opcionales para la optimización por esta misma razón. El tipado estático generaliza este uso.

En contraste, el tipado dinámico permite a los compiladores e intérpretes ejecutarse más rápidamente, debido a que los cambios en el código fuente en los lenguajes dinámicamente tipados puede resultar en un menor chequeo y menos código que revisar. Esto también reduce el ciclo editar-compilar-comprobar-depurar.

3.4 SISTEMAS DE TIPOS

Los lenguajes de programación clásicos como Pascal y C tienen sistema de tipos muy simples.

Cada constante, variable, resultado de función, y parámetro formal debe ser declarado con un tipo específico. Un tipo de sistema como este es llamado monomórfico donde la verificación de tipos es estática.

Desafortunadamente, la experiencia nos muestra que un sistema de tipos monomórfico no es satisfactorio, especialmente para escribir código re-usable. Muchos algoritmos son naturalmente genéricos, como por ejemplo un algoritmo de ordenamiento de un grupo de elementos.

Asignar tipos de datos (tipificar) da significado a colecciones de bits. Los tipos de datos normalmente tienen asociaciones tanto con valores en la memoria o con objetos como con variables. Como cualquier valor simplemente consiste en un conjunto de bits de un ordenador, el hardware no hace distinción entre dirección de memoria, código de instrucción, caracteres, enteros y números en coma flotante. Los tipos de datos informan a los programas y programadores cómo deben ser tratados esos bits.

Las principales funciones que los sistemas de tipificación ofrecen son:

- **Seguridad** - El uso de tipos de datos puede permitir a un compilador detectar incoherencias en el significado o código probablemente inválido. Por ejemplo, podemos identificar una expresión `3 / "Hello, World"` como inválida porque no se puede dividir (de forma normal) un entero por una cadena de caracteres. Un sistema de tipado fuerte ofrece más seguridad, pero no garantiza, necesariamente una seguridad completa.
- **Optimización** - Un sistema de tipado estático puede dar información muy útil al compilador. Por ejemplo, si un tipo de dato dice que un valor debe alinearse en múltiplos de 4, el compilador puede usar de forma más eficiente las instrucciones máquina.
- **Documentación** - En sistemas de tipificación más expresivos, los tipos de datos pueden servir como una forma de documentación, porque pueden ilustrar la intención del programador. Por ejemplo, los ítem de tiempos pueden ser un subtipo de un entero, pero si un programador declara una función como que devuelve ítems de tiempo en lugar de un simple entero, esto documenta parte del significado de la función.
- **Abstracción** (o **modularidad**) - Los tipos de datos permiten a los programadores pensar en los programas a un alto nivel, sin tener que preocuparse con el bajo nivel de la implementación. Por ejemplo, los programadores pueden pensar en una cadena de caracteres como un valor en lugar de un simple array de bytes. O los tipos de datos pueden permitir a los programadores expresar la Interfaz entre dos subsistemas. Esto localiza las definiciones requeridas para la interoperabilidad de los subsistemas y previene de inconsistencias cuando estos subsistemas se comuniquen.

Un programa normalmente asocia cada valor con un tipo de dato determinado (aunque un tipo de dato puede tener más de un subtipo). Otras entidades, como los objetos, bibliotecas, canales de comunicación, dependencias o, incluso, los propios tipos de datos, pueden ser asociados con un tipo de dato. Por ejemplo:

- Tipo de dato - un tipo de dato de un valor
- clase - un tipo de dato de un objeto

Un *sistema de tipado*, especificado en cada lenguaje de programación, estipula las formas en que los programas pueden ser escritos y hace ilegal cualquier comportamiento fuera de estas reglas.

3.5 CONVERSIÓN DE TIPOS

Debido a la necesidad de convertir valores de un tipo a valores de otro tipo, algunos lenguajes asocian la conversión en su sistema de tipos de forma que las conversiones se las haga en forma automática, en algunos casos el programador debe hacer explicitas estas conversiones.

Coerción

Es la conversión implícita o automática de un tipo a otro. Por ejemplo: Sea la sqrt una función que calcula la raíz cuadrada de un número real. La llamada sqrt(5) presenta como argumento el numero entero 5, este valor es automáticamente convertido al valor real 5.0, sin que el programador se percate de ello para que pueda efectuarse la operación.

La mayoría de los lenguajes de programación maneja la expresión 2*3.142 como si fuera 2.0*3.142. La coerción es la conversión de tipo en otro, realizada automáticamente por el lenguaje de programación. En 2 * 3.142, se hace que el entero 2 sea real antes de que se lleve a cabo la multiplicación.

Ejemplo: Considere el System.out.println de Java

```
class Imprime{
                public static void main(Strings args[]) {
                   inti=2; double x=8.1; String p="palabra";
                   System.out.println(i+x);
                   System.out.println(p+i*x);
                }
            }
```

Conversión explicita

En la conversión explicita el programador se encarga de hacer las conversiones directamente en código (conocida como casting). La forma de escritura de esta conversión dependerá del lenguaje, ver el siguiente ejemplo en código C y C++.

Ejemplo en C o Java:

```
...
int x;
...
x = (int) (1.5 + (double) (x/2));
...
En C++:
...
int x;
...
x = int(1.5 + double(x/2));
...
```

3.6 TIPOS DE DATOS EN DIFERENTES LENGUAJES

Se presentan a continuación y de manera de visualizar las diferentes posturas asumidas en los principios en la construcción, que definieron a cada uno de los programas ya utilizados dentro de la asignatura.

3.6.1 TIPOS DE DATOS EN EL LENGUAJE FUNCIONAL HASKELL

Haskell es un lenguaje de programación estandarizado multi-propósito puramente funcional con semánticas no estrictas y fuerte tipificación estática. En Haskell, "una función es un ciudadano de primera clase" del lenguaje de programación. Como lenguaje de programación funcional, el constructor de controles primario es la función. Las características más interesantes de Haskell incluyen el soporte para tipos de datos y funciones recursivas, listas, tuplas, guardas y calce de patrones.

- **Tipos Simples Predefinidos:**
  - Bool, Int, Integer, Float, Double, Char
- **Tipos Compuestos:**
  - Tuplas: ('a',True,3)
  - Listas: [1,2,3,4] ó (1:(2:(3:(4:[]))))
- **Tipos definidos por el usuario:**
  - Tipos sinónimo, producto, polimórficos, recursivos

3.6.2 TIPOS DE DATOS EN EL LENGUAJE LÓGICO PROLOG

En Prolog los tipos disponibles son:

- **Tipos predefinidos:**
  - symbol, string, integer, real, char
- **Tipos Compuestos:**
  - Tuplas o functores
  - Listas

3.6.3 TIPOS DE DATOS EN JAVA

En Java los tipos disponibles son:

- **Tipos simples:**
  - byte, short, int, long, float, double, char y boolean.
- **Tipos Compuestos:**Se basan en los tipos simples, e incluyen:arreglos, cadenas, matrices y tanto las clases como las interfaces, en general. Por ejemplo, la librería de clases de Java provee clases de envoltorio o wrap, que permiten manejar los datos equivalentes de los tipos primitivos, poseen una gran cantidad de métodos para el manejo de los mismos, tales como modificaciones, conversiones, comparaciones, etc. Las principales son:
  - Boolean.

- Double
- Float
- Integer

3.6.4 TIPOS DE DATOSEN SMALLTALK

Smalltalk implementa los tipos de datos como clases de Smalltalk, y las operaciones sobre datos como métodos en estas clases. Cada dato es una instancia de una de estas clases.

Smalltalk tiene cinco clases de datos: String (cadena), Number (número), Carácter (caracter), Symbol (símbolo) y Array (vector). Las subclases de Number: Integer (entero), float (real), y Fraction (fracción) son parte de los literales también.

Las operaciones sobre datos, incluyendo las comparaciones lógicas, las operaciones aritméticas, las operaciones lógicas usando "and" y "or", los condicionales lógicos, y las iteraciones, son implementadas como métodos en sus respectivas clases.

3.6.5 TIPOS DE DATOS EN C

Los tipos básicos son:

char	caracteres
int	enteros
unsignedint	enteros, con aritmética módulo 2^n
float	reales
double	reales de doble precisión

no se incluyen booleanos. En la evaluación de condiciones, un valor diferente de cero se reconoce como true y un cero como false.

4. MECANISMOS DE CONTROL DE FLUJO

Otra cuestión no menos importante, es como las estrategias para la solución de los problemas prevé cada lenguaje. Esto es de que elementos disponemos para la implementación de un algoritmo para la solución de un problema determinado.

El Control de Flujo es fundamental para la mayoría de los modelos de cómputo, pues establece el orden en que debe ejecutarse el programa.

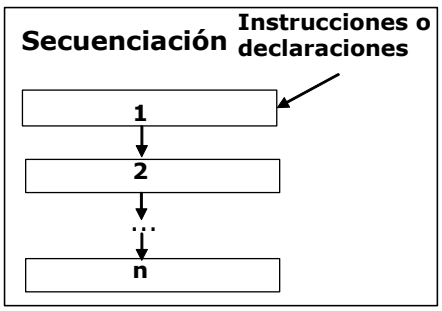
El flujo de control hace referencia al orden en que las llamadas a funciones, instrucciones y declaraciones son ejecutadas o evaluadas

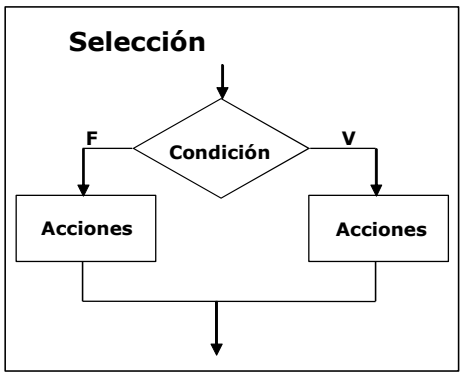
4.1.1 ORGANIZACIÓN DE LOS MECANISMOS DE CONTROL

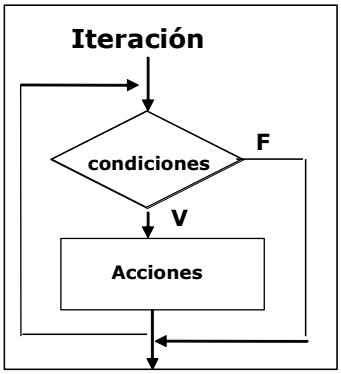
Los mecanismos de control se pueden organizar según los siguientes criterios:

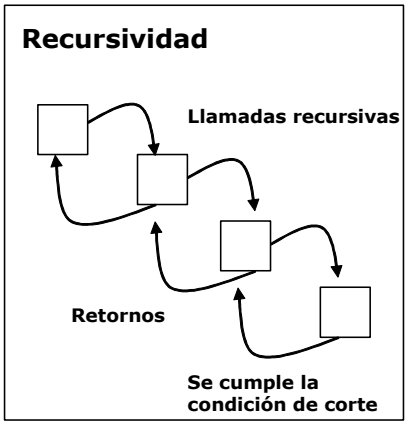
- **Secuenciación**: orden específico, usualmente el de aparición en el programa.

- **Selección**: se escoge entre dos o más instrucciones según alguna condición a tiempo de ejecución.
- **Iteración**: un fragmento de código se ejecuta de manera repetida bien sea un número de veces o hasta cumplirse determinada condición a tiempo de ejecución.
- **Recursión**: una expresión se define en términos de si misma directa o indirectamente.
- **Abstracción procedimental**: una colección de construcciones de control se encapsula como una unidad, sujeta a parametrización.
- **No determinismo**: el orden o escogencia de instrucciones y expresiones no es especificado deliberadamente.
- **Concurrencia**: dos o más fragmentos de programa se ejecutan/evalúan al mismo tiempo.









Abstracción Procedimental

El mensaje de Smalltalk do: unBloque : es una iteración general que recorre cada elemento de la colección y ejecuta un bloque de código.

```
"Calcula el total de la suma de los números en numeros"
  |numeros suma|
  numeros := #(1 2 3 4 5).
  suma := 0.
  numeros do: [ :num | suma := suma + num].
  ^suma
```

4.2 EJEMPLOS DE FLUJO DE CONTROL EN DIFERENTES LENGUAJES

4.2.1 FLUJO DE CONTROL EN C Y JAVA

- La selección se resuelve con las siguientes estructuras:

La estructura if, permite elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición. Si la condición es verdadera, se ejecuta las instrucciones 1 y si es falsa, se ejecuta las instrucciones 2.

```
if (condicion)
{
      //instrucciones 1
}
else
{
      //instrucciones 2
}
```

La estructura switch, (decisión múltiple) evaluará una expresión que podrá tomar n valores distintos: 1, 2, 3,..., n. Según que elija uno de estos valores en la condición, se realizará una de las n acciones, o lo que es igual, el flujo del algoritmo seguirá sin determinado camino entre los n posibles.

```
switch (expresión_entera) {
      case (valor1) : instrucciones_1; [break;]
      case (valor2) : instrucciones_2; [break;]
      ...
      case (valorN) :instrucciones_N; [break;]
      default: instrucciones_por_defecto;
}
```

- La iteración se resuelve con las siguientes estructuras:

La estructura while: Permite repetir un ciclo de instrucciones mientras se cumple una determinada condición. Cuando se ejecuta la instrucción mientras, lo primero que sucede es que se evalúa la condición (una expresión booleana). Si se evalúa falsa, ninguna acción se toma y el programa prosigue en la instrucción siguiente al cuerpo del bucle. Si la expresión booleana es *verdadera,* entonces se ejecuta el cuerpo del bucle, después de lo cual se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez **mientras** la expresión **(condición)** sea verdadera.

```
while (condicion)
{
      //instrucciones
}
```

La estructura do… while: Permite ejecutar un conjunto de instrucciones y repetirlas mientras se cumple una determinada condición.

```
do
{
      //instrucciones
} while (condicion);
```

La estructura for: Permite ejecutar las instrucciones del ciclo un número especificado de veces usando una o más variables de control de ciclo.La estructura está formada por cuatro partes:

1. Inicialización: Inicializa la o las variables de control.

2. Condición: Condición (expresión booleana) del ciclo, generalmente esta directamente relacionada con la o las variables de control.

3. Incremento: Incrementa la o las variable de control, los valores van evolucionando mientras el ciclo se repita.

4. Cuerpo: Sentencia simple o compuesta que se repite de acuerdo a si se cumple la condición del ciclo o no.

```
for (inicializacion; condicion; incremento)
{
      //instrucciones
}
```

Control General del Flujo

Cuando se necesita interrumpir el control del flujo se puede utilizar:

- **break**: Permite salir de la estructura alternativa múltiple o repetitiva.
- **continue**: Permite saltar al principio de una ejecución repetitiva.
- **return**expr : retorna desde una función, una expresión o no.

4.2.2 FLUJO DE CONTROL EN HASKELL

Haskell es un **lenguaje de programación puramente funcional**. En los lenguajes imperativos obtenemos resultados dándole al computador una secuencia de tareas que luego éste ejecutará. Mientras las ejecuta, puede cambiar de estado. Por ejemplo, establecemos la variable aa 5, realizamos algunas tareas y luego cambiamos el valor de la variable anterior. Estos lenguajes poseen estructuras de control de flujo para realizar ciertas acciones varias veces (for, while...). Con la programación puramente funcional no decimos al computador lo que tiene que hacer, sino más bien, decimos como son las cosas. El factorial de un número es el producto de todos los números desde el 1 hasta ese número, la suma de una lista de números es el primer número más la suma del resto de la lista, etc. Expresamos la forma de las funciones. Lo único que puede hacer una función es calcular y devolver algo como resultado. Al principio esto puede parecer una limitación pero en realidad tiene algunas buenas consecuencias: si una

función es llamada dos veces con los mismos parámetros, obtendremos siempre el mismo resultado. A esto lo llamamos ***transparencia referencial*** y no solo permite al compilador razonar acerca de el comportamiento de un programa, sino que también nos permite deducir fácilmente (e incluso demostrar) que una función es correcta y así poder construir funciones más complejas uniendo funciones simples.

- Las selección se resuelve con declaraciones **guarded e if…then..else**.

guarded:

```
| x >= y && x>=z =x
| y >= x && y>=z =y
| otherwise =z
```

if..then..else:

```
if x >= y && x>=z then x
else if y >= x && y>=z then y
else z
```

- Utiliza la **recursividad**, (una alternativa para las iteraciones).

4.2.3 FLUJO DE CONTROL EN PROLOG

En el caso de Prolog, prácticamente no tiene sentido hablar de flujo de control. Se dice que procede en forma no determinista. En todo caso, es mucho menos lo que podemos hacer que con el lenguaje Haskell.

En Prolog, especificamos los hechos y las reglas que los relacionan (Sección clauses) y luego formulamos consultas que son las metas u objetivos a obtener. Por **control** se entiende la forma en que el lenguaje busca las respuestas a los objetivos.

Cuando se ha escrito una consulta (Query), el intérprete busca un hecho o regla que coincida con la primera meta de la consulta (y trata de probar esta meta antes de tratar de probar una segunda meta, si la hay).

Si la meta (U objetivo) coincide con un hecho, se ha probado, y la tarea del intérprete con la meta termina allí.

Si la meta coincide con la cabeza de una regla, cada uno de los términos de su cuerpo se convierte en un nuevo objetivo que debe ser satisfecho. El intérprete no termina hasta que todos los objetivos se han cumplido con un hecho. Si uno de los objetivos no se cumple, toda la consulta falla.

Hay casos en los que el intérprete podría tener que tomar una decisión sobre cómo hacer frente a una consulta que no tiene nada que ver con la lógica del programa. Esto es, si ocurre cuando hay más de una cabeza que coincide con un objetivo (queremos los pares de personas que son cuñados), y cuando el cuerpo de una regla consiste en más de una cláusula. A nivel de ejemplo considere el siguiente par de reglas donde declaramos de dos formas la relación cuñado:

```
cuñado(X, Y): -
      casado(X,Z),
      hermano(Z,Y)

cuñado(X,Y)
      hermano(X,Z),
      casado(Z, Y).
```

```
X es cuñado de Y porque:
  X está casado con Z y además
  Z es hermano de Y
            Ó bien
X es cuñado de Y porque:
  X es hermano de Z y además
Z estácasado con Y
```

El objetivo se obtiene mediante un mecanismo conocido como **Backtracking**(Vuelta atrás). Este mecanismo funciona recorriendo el archivo creado a partir de los hechos y reglas. El orden en que ellas

se detallan tiene importancia. Además, en la formulación de metas existe una regla (!) que introduce algo de determinismo en su forma de proceder. Vemos todo esto en detalle en su respectiva unidad.

En resumen:

- La selección se podría comparar con las reglas que son afirmaciones condicionales, por ejemplo:
```
alumno(juan)      es un hecho o afirmación incondicional
Regular(X) if
            alumno(X)              es una implicación que
            and aproboPrimerParcial(X) será cierta si se cumplen
            and    aproboSegundoParcial(X)    las    proposiciones
dependientes
```

- Utiliza el mecanismo de recursividad, por ejemplo, para recorrer las listas.

4.2.4 FLUJO DE CONTROL EN SMALLTALK

Smalltalk, utiliza abstracciones prodecimentales, tanto para manejar la selección como la iteración:

- **Selección: Se forma enviando mensajes a objetos booleanos y evaluando bloques si las condiciones se cumplen.Ejemplos:**

```
( a>b )  ifTrue: [ c:= a + b ]. Ejecuta el bloque si la condición
                                es verdadera
( a=b ) ifFalse: [ c:= a + b].  Ejecuta el bloque si la condición
                                es falsa.
( a>=b )                        Ejecuta el bloque [ c:= a + b ] si
ifTrue:  [ c:= a + b ].  la condición es verdadera;
ifFalse: [ c:= a  - b ]. en caso contrario ejecuta el otro.
```

- **Iteraciones:** Hay cuatro tipos de iteraciones:

  - Evaluar un bloque n veces. El mensaje de palabra clave es **timesRepeat:**.
  - Evaluar un bloque hasta que se encuentre en una condición false. El mensaje de palabra clave es **whileFalse:**.
  - Evaluar un bloque hasta que se encuentre con una condición true**.** El mensaje de palabra clave es **whileTrue:.**
  - Evaluar un bloque usando un índice. El mensaje de palabra clave es **to:do:.**

  Ejemplo: Itera de 1 hasta 10, imprimiendo los sucesivos números en la ventana Transcript.
```
1 to: 10
do: [ :x | Transcript show: x; cr].
```

5. BIBLIOGRAFÍA

- David A. Watt, William Findlay - 2004 - Programming Language Desing Concepts
- Robert Sebesta - 2011 - Concepts of Programming Languages
- Spigariol, Lucas – 2008 – Apunte de Paradigmas de Programación – UTN, FRBA
- J. Baltasar García Perez-Schofield - Curso de Doctorado: Tecnologías de
- Objetos - http://webs.uvigo.es/jbgarcia/
- Apunte de la Cátedra P.P.R -UTN FRC Tymoschuk/Serra/Castillo Ed. Educa
- Lenguajes de Programación - Conceptos y Constructores RaviSethy Ed. Addison Wesley
- Compiladores. Análisis semántico y chequeo de tipos. Universidad Galileo