UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

AUTOR: Ismael Israel Perea Camarillo

PROGRAMACIÓN		Clave: 1369
Plan: 2006		Créditos: 8
Licenciatura: Informática		Semestre: 3
Área: Informática		Hrs. Asesoría: 4
Requisitos: Ninguno		Hrs. Por semana: 4
Tipo de asignatura:	Obligatoria (X)	Optativa ( )

Objetivo general de la asignatura

Al finalizar el curso el alumno conocerá la evolución de los lenguajes de programación, así como las diferentes filosofías (paradigmas) que emplean para describir modelos de la realidad.

Temario oficial (68 horas sugeridas)

1. Definición de un lenguaje de programación (18 h.)
2. Paradigma Imperativo (20 h.)
3. Paradigma orientado a objetos (10 h.)
4. Paradigma funcional (10 h.)
5. Paradigma lógico (10 h.)

Introducción

Todas las computadoras necesitan ser programadas, es decir, almacenar en memoria la información sobre las tareas que van a ejecutar.

"Una de las definiciones comúnmente presentadas de lenguaje de programación es *notación para comunicarle a una computadora lo que deseamos que haga*. Esta definición es inadecuada, ya que antes de los años 40 las computadoras se programaban mediante cableado e interruptores"[1], y el cablear es todo menos una notación.

"En los 40 se dio un adelanto importante en el diseño de las computadoras, cuando John von Neumann (se pronuncia *"Noiman"*) tuvo la idea de que una computadora no debería estar "cableada" para ejecutar algo en particular, sino que podría lograrse que una serie de códigos almacenados como datos determinarán las acciones ejecutadas por una unidad de procesamiento central"[2]. Desarrolló un modelo que lleva su nombre, para describir el concepto de *"programa almacenado en memoria"*.

[1] Kenneth C. Louden, *Programming Languages: Principles and Practice*, 2a ed., Brooks Cole, 2003, p. 2.
[2] loc. cit.

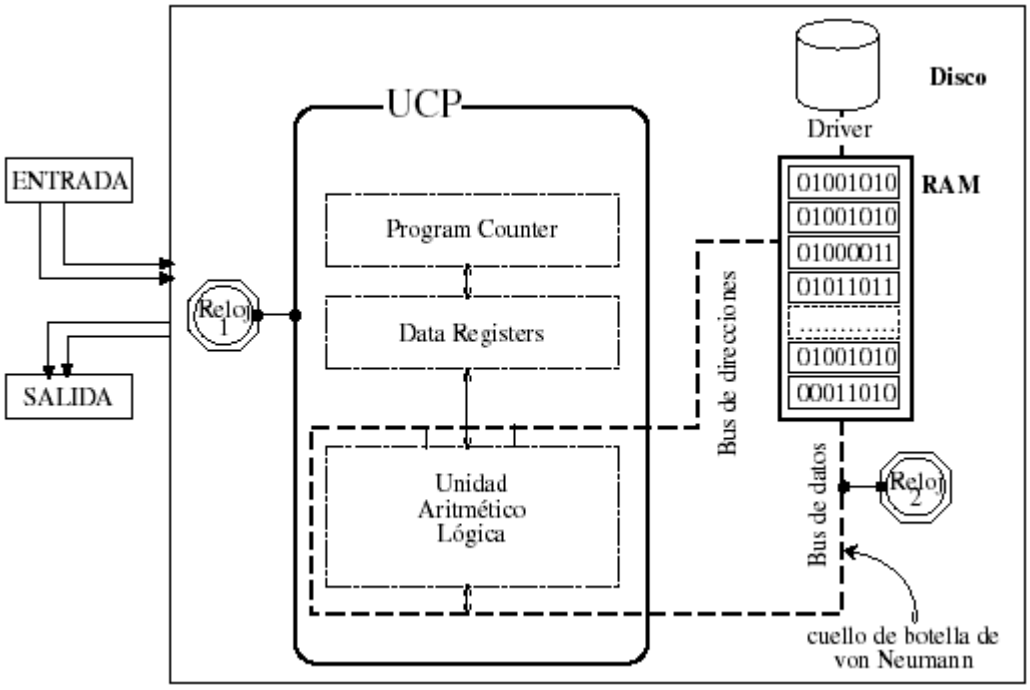


Figura 1.1. Modelo de Von Neumann[3]

Tema1. Definición de un lenguaje de programación

Objetivo particular

Al término de este tema, el alumno:

Definirá qué es un lenguaje de programación y podrá explicar las clasificaciones que se hacen de éste, sus niveles de abstracción, los paradigmas que los implementan, su evolución e historia, su definición formal (a través de la sintaxis y semántica) y el proceso de traducción.

---

[3] Fuente: http://informatica.utem.cl/~mcast/CCOMPUTACION/Introduccion/ModelosComputacionales.pdf [Consulta: 30 de enero de 2009]

Temario detallado

1.1 ¿Qué es un lenguaje de programación?

1.2 Clasificación de los lenguajes de programación

1.3 Abstracción en los lenguajes de programación

- 1.3.1De datos
- 1.3.2 De control

1.4 Historia y evolución de los lenguajes de programación

1.5 Definición de un lenguaje de programación

- 1.5.1 Sintaxis
  - 1.5.1.1 Léxico
  - 1.5.1.2 Metalenguajes y notación BNF
- 1.5.2 Semántica

1.6 Traducción de los lenguajes de programación

- 1.6.1 Compilación
  - 1.6.1.1 Programa fuente
  - 1.6.1.2 Análisis léxico
  - 1.6.1.3 Análisis sintáctico
  - 1.6.1.4 Análisis semántico
  - 1.6.1.5 Generación de código intermedio
  - 1.6.1.6 Programa objeto
  - 1.6.1.7 Optimizador de código
  - 1.6.1.8 Tabla se símbolos
- 1.6.2 Interpretación

Introducción

Un lenguaje de programación es el resultado de querer hacer que una computadora o dispositivo programable ejecute cosas que nosotros le indiquemos. Pero al tener su propio lenguaje (ceros y unos), inicialmente no era tan sencillo programar debido a que una sola operación de suma implicaba varias líneas de código máquina, lo que hacía lento el proceso de construcción de programas. La solución: crear programas a partir de abstracciones que nosotros aplicamos en la vida real y que se han ido implementando en los lenguajes posteriores al lenguaje máquina. Así surgió el lenguaje ensamblador como el primer intento de cambiar los 0s y 1s por nemotecnias como ADD, MOV, LOAD, etc., las cuales son más fáciles de utilizar. Ya en la tercer y cuarta generación las nemotecnias se transforman en estructuras muy semejantes al lenguaje natural, con una sintaxis y una semántica que nos permite trasladar nuestra lógica de resolver los problemas a la lógica de una computadora. Veamos a continuación cuáles son estas abstracciones y sus tipos, además de los dos elementos que nos permiten definir a un lenguaje de programación como tal: la sintaxis y la semántica.

1.1 ¿Qué es un lenguaje de programación?

"Un lenguaje de programación es un sistema notacional que puede ser utilizado para controlar el comportamiento de una máquina, particularmente una computadora, y que describe operaciones computacionales en una forma legible tanto para la computadora como para el ser humano".[4] Es un conjunto de símbolos (lexemas) que se agrupan en categorías (tokens), contiene reglas sintácticas y semánticas que le dan estructura y significado a sus elementos y expresiones.

[4] Kenneth C. Louden, op. cit., p. 3.

No sólo existen lenguajes que sirven para programar, también hay otro tipo de lenguajes, como los lenguajes de marcado (HTML por ejemplo) y los metalenguajes (como XML o la notación BNF), que permiten crear otros lenguajes (llamados lenguajes objeto).

Los lenguajes de programación intentan parecerse a los lenguajes naturales, pero esto lleva a un problema: nuestro lenguaje es ambiguo, y un lenguaje de programación puede ser todo, menos ambiguo.

1.2 Clasificación de los lenguajes de programación

Los lenguajes de programación se clasifican según su nivel de abstracción, la forma en que se ejecutan en la computadora y por el paradigma o filosofía que implementen:

Según su nivel de abstracción

Lenguajes de bajo nivel: se programa en 0s y 1s, conocido como lenguaje máquina.

Lenguajes de medio nivel: utiliza nemotecnias para programar, se les conoce como lenguaje ensamblador.

Lenguajes de alto nivel: están formados por palabras que se usan en los idiomas o lenguajes naturales, como el inglés. Ejemplos de este tipo de lenguajes están: C, C++, Java, Basic, Pascal, Ruby y PHP.

Según la forma de ejecución

Con base en lo anterior, las computadoras trabajan ya se con 0 y 1s, ensamblador o alto nivel. Cuando se usan los niveles medio y alto de abstracción, los programas se ejecutan de dos maneras:

- Un programa llamado intérprete que va ejecutando cada una de las instrucciones que va leyendo de un programa. A este proceso se le llama *interpretar*.

- Un programa llamado compilador que traduce cada una de las instrucciones de un programa a su equivalente en lenguaje de 0 y 1s (lenguaje máquina). A ese proceso se le llama *compilación*.

Según el paradigma de programación

Un paradigma de programación es un conjunto de reglas y conceptos que dirigen la elaboración de programas que a su vez constituyen software o programas de aplicación.

Existen cuatro paradigmas principales de programación:

- Paradigma imperativo
- Paradigma orientado a objetos
- Paradigma funcional
- Paradigma lógico

1.3 Abstracciones en los lenguajes de programación

Existen dos tipos de abstracciones que se implementan en los lenguajes de programación:

- De datos: Resumen las propiedades de los datos como cadenas de caracteres, números o árboles de búsqueda.

- De control: Resumen propiedades de la transferencia de control, o sea, de la modificación de la trayectoria de ejecución de un programa con base en una situación determinada. Ejemplos: bucles, sentencias condicionadas, llamadas de procedimiento, etc.

Todas las abstracciones tienen niveles, que miden la cantidad de información contenida en la abstracción. Existen tres niveles:

- Básicas: Reúnen la información de máquina más localizada.
- Estructuradas: Reúnen información más global sobre la estructura del programa.
- Unitarias: Reúnen información sobre una parte completamente funcional de un programa.

1.3.1 Abstracciones de datos

Básicas: resumen la representación interna de valores de datos comunes en una computadora. Ejemplo: a menudo los valores enteros de datos se almacenan en una computadora utilizando una representación de complemento a dos. Las localizaciones en la memoria que contienen valores de datos se abstraen dándoles un nombre: se conocen como identificadores o variables. El tipo de valor de datos también recibe un nombre y se conoce como tipo. A las variables se les dan nombres y tipos de datos mediante una declaración.

```
var x: integer;
int x;
```

Estructuradas: La estructura de los datos es el método principal para la abstracción de colecciones de valores de datos relacionados entre sí. Una estructura típica es el arreglo: reúne datos en una secuencia de elementos de indización individual. A las variables se les puede dar una estructura de datos.

```
int a[10];
INTEGER a(10);
```

Las estructuras de datos también pueden ser nuevos tipos de datos:

```
typedef int intArray[10];
```

Unitarias: Es la reunión de códigos relacionados entre sí en localizaciones específicas dentro de un programa, ya sea en forma de archivos por separado o como estructuras de lenguaje por separado dentro de un archivo. Incluyen acuerdos convencionales y restricciones de acceso, que se conocen como encapsulado de datos y ocultamiento de la información. Ejemplos: los módulos, los paquetes, las clases (intermedia entre el nivel estructurado y unitario), componentes, librerías (bibliotecas), etc.

1.3.2 Abstracciones de control

Básicas: Son aquellos enunciados o sentencias que combinan unas cuantas instrucciones de máquina en una sentencia abstracta más comprensible.
Ejemplo: el enunciado de asignación, que resume el cómputo y almacenamiento de un valor en la localización dada por una variable.

```
x = x + 3;  // enunciado de asignación
```

Este enunciado de asignación representa la recuperación del valor de la variable x, agregando el entero 3 a la misma y almacenando el resultado en la localización de x.

Estructuradas: Dividen un programa en grupos de instrucciones que están anidadas dentro de pruebas que gobiernan su ejecución. Ejemplo: Enunciados de selección: if, case, switch. Un mecanismo adicional para estructurar el control es el procedimiento (subprograma o subrutina).

```
procedure gcd (u, v: in integer; x: out integer) is
…
end gcd;
```

Este procedimiento puede ser llamado simplemente con nombrarlo y proporcionarle los parámetros apropiados o argumentos reales:

```
gcd (8, 18, d);
```

Unitarias: Sirven para incluir una colección de procedimientos que proporcionan servicios relacionados lógicamente con otras partes del programa y que forman una parte unitaria o independiente. Es esencialmente lo mismo que una abstracción unitaria de datos, y se implementa de igual forma a través de módulos, paquetes, clases, etc. La diferencia consiste en el enfoque: en las abstracciones de control unitarias el enfoque está en las operaciones y en los servicios que proporcione, más que en los datos, pero las metas de reutilización y formación de bibliotecas se conservan.

1.4 Historia y evolución de los lenguajes de programación

¿Cuál fue el primer lenguaje de programación? Hasta la fecha, ¿cuántos lenguajes de programación han existido? ¿Cómo han evolucionado?

- Para responder estas preguntas, qué mejor que una *timeline*[5] de los lenguajes de programación, la cuál se anexa en formato PDF. En este sitio encontrarás un trabajo de investigación muy completo sobre la historia y evolución de los lenguajes de programación desde 1954 con el surgimiento de Fortran, hasta 2008 y Phyton (y hasta donde se acumule). Otros lenguajes que se muestran son: BASIC, C, Perl, Pascal, PHP, C++, C#,

[5] Fuente: Éric Lévénez: “Computer Languages History”, en línea, disponible en: http://www.levenez.com/lang/ [consulta: 15 de agosto de 2008].

VB.NET, Delphi, Eiffel, Java, JavaScript, Oz, PHP, Python, Ruby, Smalltalk, Haskell, Miranda, Lisp, Scheme, Ocaml, ML, Prolog, etc.

1.5 Definición de un lenguaje de programación

Antes, los lenguajes de programación eran solo descripciones informales en inglés. Un lenguaje de programación necesita una descripción precisa y completa que lo defina formalmente. Esta definición se da con:

- La sintaxis
- La semántica

1.5.1 Sintaxis

Es la estructura de un lenguaje. Son las reglas que indican cómo realizar las construcciones del lenguaje de programación. Es como la *gramática* de un lenguaje natural: describe las maneras en que las diferentes partes del lenguaje pueden ser combinadas para formar otras partes. Representa la estructura superficial del lenguaje. Por ejemplo, en C:

x = x + 3; // secuencia de símbolos válida

x 3 x + =; // secuencia no válida

La sintaxis proporciona la información significativa necesaria para entender un programa y proporciona mucha información para la traducción del programa fuente en un programa objeto. Ejemplo:

La expresión 2 + 3 * 4 = 14 es interpretada como:

2 + (3 * 4) = 14

y no como:

(2 + 3) * 4 = 20

1.5.1.1 El léxico

Ayuda a la especificación de la sintaxis y estructura de un lenguaje de programación. Es similar a la ortografía de un lenguaje natural. Constituye el conjunto de símbolos permitidos o vocabulario. El léxico se conforma de:

- Lexemas
- Tokens
- Sentencias

Lexema: son las unidades sintácticas de más bajo nivel. Incluyen: identificadores, operadores y palabras especiales.
Token: es una categoría de lexemas (el lexema es un atributo del token).
Sentencia: los lenguajes de programación utilizan conjuntos de cadenas de caracteres pertenecientes a algún alfabeto. Una sentencia es cada una de estas cadenas. Dicho de otra forma, son cadenas de lexemas.

Algunas categorías de lexemas (o tokens) son:

- Conjunto de caracteres ASCII (abcdefgABCDEFG123456[]{}/+-*)
- Identificadores (cualquier nombre de una variable, constante, tipo de dato)
- Símbolos operadores (+*/-=<>!)
- Palabras clave y palabras reservadas (if, else, case, this, include)
- Literales y constantes ($_GET, $_POST, cualquier valor de un tipo de dato)
- Comentarios (// Este es un comentario, */ Este es otro comentario*/)
- Espacios y delimitadores (|,_,-)
- Formatos de libre campo y campo fijo (posición en columnas de una instrucción)
- Expresiones (cualquier expresión regular)
- Sentencias o enunciados (declaraciones o asignaciones)

Los tokens de un lenguaje de programación pueden describirse empleando expresiones regulares:

Expresión regular	Token
letra (letra | dígito | '_')*	**Identificador**
('a'..'z') U ('A'..'Z')	**letra**
'0'..'9'	**dígito**

1.5.1.2 Metalenguajes y notación BNF

Un metalenguaje es usado para definir a otros lenguajes. A estos se les llama lenguajes objeto. En lingüística la sintaxis que describe la gramática de un lenguaje es un ejemplo de metalenguaje.

Por ejemplo, *bisílaba* es toda aquella palabra que tiene dos sílabas. Pero en sí misma la palabra *"bisílaba"* no es bisílaba por definición.

Otro ejemplo del uso de los metalenguajes es en las matemáticas: los metalenguajes sirven para resolver paradojas matemáticas. ¿Cuál de las siguientes proposiciones dice la verdad?

A: El enunciado B dice la verdad.
B: El enunciado A dice mentiras.

Necesitamos una proposición escrita en metalenguaje matemático para poder calificar a las otras dos proposiciones, las cuales estarán definidas en lenguaje objeto.

En informática, XML es un ejemplo de metalenguaje de marcado, ya que a partir de él se describen otros lenguajes de marcado objeto como XLink, XMath, XSL, etc. Otro ejemplo de metalenguaje informático es la notación BNF.

La notación BNF

La notación BNF se usa para describir lenguajes de programación. Es una nomenclatura que sirve para describir la sintaxis de un lenguaje usando ciertos símbolos y reglas.

Se manejan dos tipos de elementos: los elementos o **símbolos terminales**, que pertenecen al vocabulario y que se escriben tal cual, y los elementos o **símbolos no terminales** que se escriben entre los símbolos <>.

Los símbolos BNF son:

Sintaxis	Significado
::=	se define como
<>	se usa para delimitar el nombre de una categoría
|	se usa para separar opciones de una categoría

Ejemplo:

<digito>::= 0|1|2|3|4|5|6|7|8|9
<operador>::= =|-|*|+|/
<letra> ::= a|A|b|B|c|C|…|z|Z

1.5.2 Semántica

Son las reglas que determinan el significado de un lenguaje.
En C la sintaxis del `if`:

```
<enunciado if>::= if (<expresión>) <enunciado>
                        [else <enunciado>]
```

En C la semántica de `if`:

Un enunciado if es ejecutado, primero, evaluando su expresión, misma que debe tener tipo aritmético o apuntador, incluyendo todos los efectos colaterales, y si se compara diferente de 0, el enunciado que sigue a la expresión es ejecutado. Si existe un parte else, y la expresión es 0, el enunciado que sigue al "else" es ejecutado.

1.6 Traducción de los lenguajes de programación

Cualquier código fuente debe ser traducido a código binario para que las instrucciones que pusimos en él puedan ser entendidas y ejecutadas por la máquina (las computadoras no entienden directamente los lexemas "`if(valor==0){instrucción 1} else {instrucción 2}`", sino que éstos deben pasarse a 0y1s).

Hay programas encargados de realizar esta traducción, se llaman **traductores**. Hay dos tipos: compiladores e intérpretes.

1.6.1 Compilación

Se atribuye el término compilador a Grace Murray Hopper. Los primeros compiladores se escribieron en los años 50. FORTRAN es el primer lenguaje compilado con éxito.

Un compilador es un programa que traduce un programa fuente escrito en lenguaje fuente y da como resultado un programa objeto en lenguaje objeto.

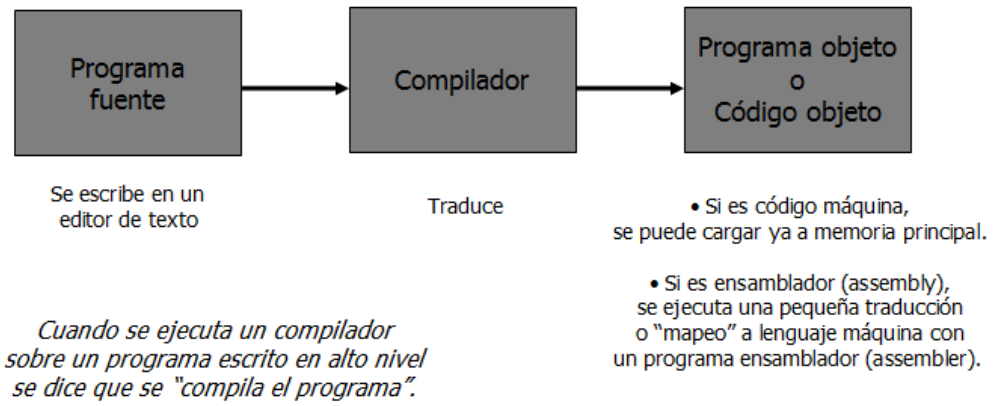


Figura 1.2. Entrada y salida en el proceso de compilación[6]

El proceso de compilación tiene el siguiente flujo:

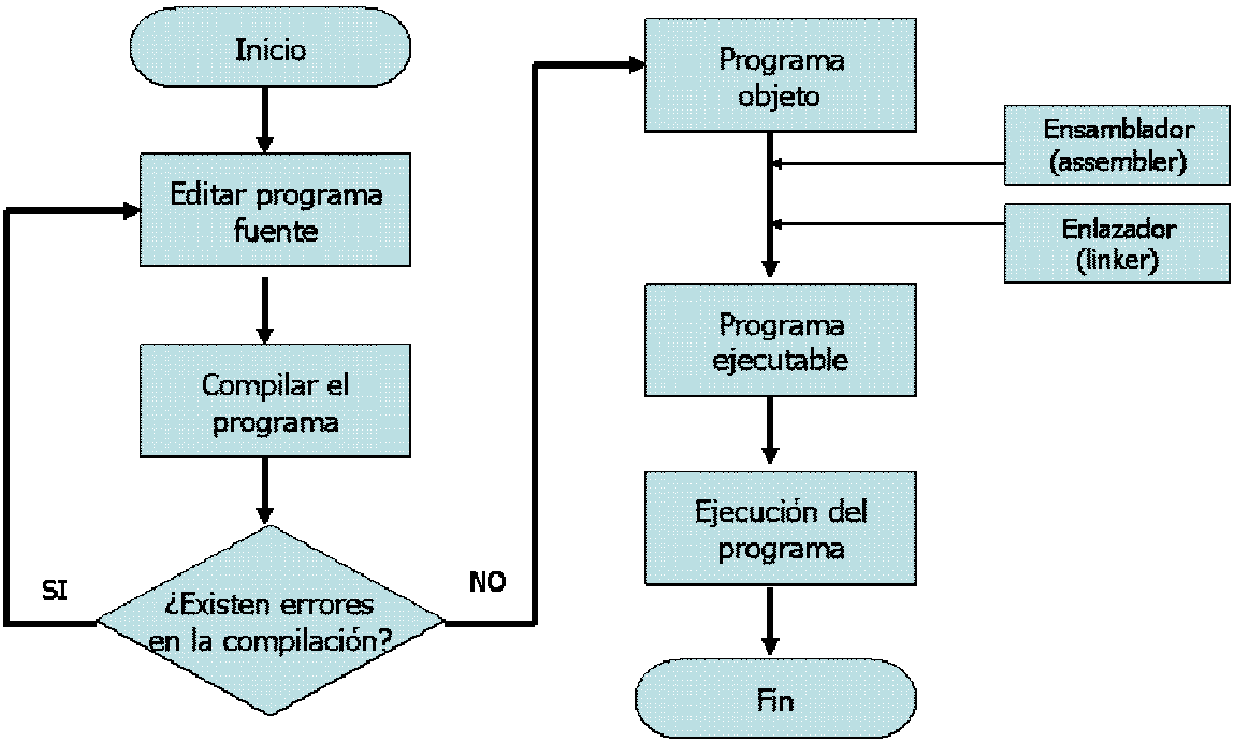


Figura 1.3. Proceso de compilación en flujo de datos[7]

[6] Fuente: Perea, Ismael: *Apuntes de programación para tercer semestre de la licenciatura en informática*, 2008.
[7] Fuente: Perea, Ismael: *Apuntes de programación para tercer semestre de la licenciatura en informática*, 2008.

Este proceso se puede dividir en una serie de fases, que pueden llevarse a cabo simultáneamente o consecutivamente y cada una de las cuales transforma el programa fuente de una representación a otra. Se pueden agrupar las fases en etapas:[8]

- Etapa de análisis o front-end
- Etapa intermedia o middle-end
- Etapa de síntesis o back-end

Veamos el detalle del proceso y sus fases:

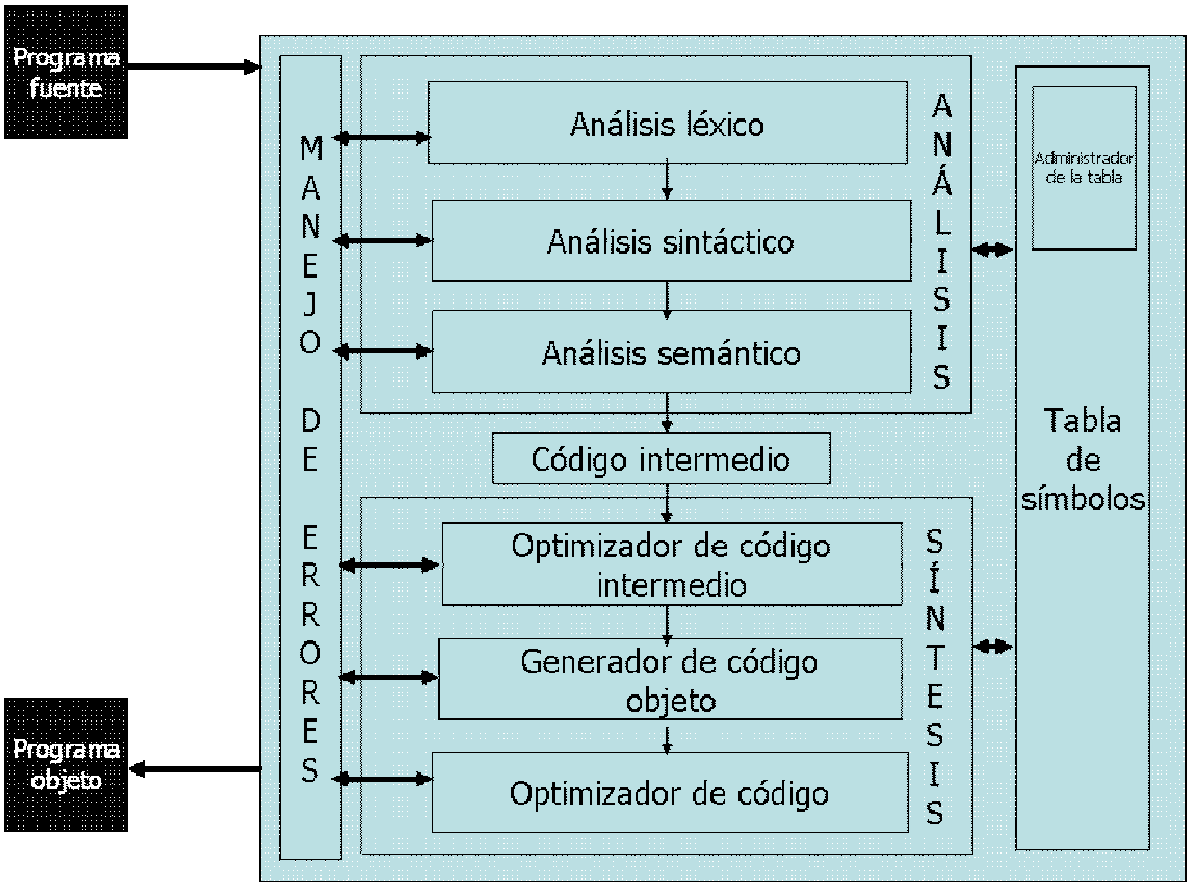


Figura1.4. El proceso de compilación, sus etapas y fases[9]

[8] César Ignacio García Osorio: *Introducción a los compiladores*, material en línea, disponible en: http://pisuerga.inf.ubu.es/cgosorio/ALeF/UD1/compilacion.pdf, consultado el 28/01/09.
[9] Basada en: César Ignacio García Osorio: *Introducción a los compiladores*, material en línea, disponible en: http://pisuerga.inf.ubu.es/cgosorio/ALeF/UD1/compilacion.pdf, consultado el 28/01/09.

1.6.1.1 Programa fuente

Un programa fuente o código fuente o simplemente "fuente" es el conjunto de instrucciones válidas para un lenguaje de programación en particular que indican a una computadora lo que debe realizar. Se crean en cualquier editor de texto o en línea de comandos (ejemplo: el shell de Unix).

- La etapa de análisis (front-end o etapa inicial) agrupa aquellas fases que dependen principalmente del lenguaje fuente[10] (java, C, C++, etc.). Las fases son:

1.6.1.2 Análisis léxico

Analizador léxico (también llamado scanner):

Agrupa los caracteres individuales en entidades lógicas léxicas[11] (lexemas) e identifica el token con el que se corresponden. Realiza las entradas oportunas en la tabla de símbolos. La salida es una secuencia de tokens, provoca una simplificación de la entrada.

1.6.1.3 Análisis sintáctico

Analizador sintáctico (también llamado parser):

Analiza la estructura general de todo el programa, agrupando las entidades simples identificadas por el scanner en construcciones mayores, como sentencias, bucles, rutinas, etc., que componen el programa completo. Normalmente se utiliza la representación de árboles sintácticos para reflejar

---

[10] César Ignacio García Osorio: *Introducción a los compiladores*, material en línea, disponible en: http://pisuerga.inf.ubu.es/cgosorio/ALeF/UD1/compilacion.pdf, consultado el 28/01/09.
[11] César Ignacio García Osorio: *Introducción a los compiladores*, material en línea, disponible en: http://pisuerga.inf.ubu.es/cgosorio/ALeF/UD1/compilacion.pdf, consultado el 28/01/09.

dicha estructura.[12] Aplica una gramática para construir el árbol sintáctico para un programa a partir de la secuencia de tokens.

1.6.1.4 Análisis semántico

Una vez determinada la estructura del programa se puede analizar su significado (semántica) mediante un análisis en el que se determina qué variables almacenarán enteros, cuáles números de punto flotante, si el acceso a los arrays cae dentro del rango fijado en su definición, etc.[13] También, realiza un estudio de los aspectos contextuales del lenguaje:

- Comprobación de tipos
- Comprobación de unicidad (un objeto se define una sola vez)
- Comprobación de nombres (todas las variables que se usan deben estar declaradas)
- Se completa la información almacenada en la tabla de símbolos a la vez que se consulta para realizar las comprobaciones oportunas

1.6.1.5 Generación de código intermedio

Es el código generado por la etapa de análisis que recibe como entrada la etapa de síntesis. Un tipo de código intermedio es el AST (Abstract Syntax Trees) que es una forma condensada de árboles de análisis, con sólo nodos semánticos y sin nodos para símbolos terminales (se supone que el programa es sintácticamente correcto). Otros tipos de códigos intermedios son: DAG (Directed Acyclic Graphs), TAC (Three Address Code) y RTL (Register Transfer Language).

---

[12] Véase, César Ignacio García Osorio: *Introducción a los compiladores*, material en línea, disponible en: http://pisuerga.inf.ubu.es/cgosorio/ALeF/UD1/compilacion.pdf, consultado el 28/01/09.

[13] Véase, César Ignacio García Osorio: *Introducción a los compiladores*, material en línea, disponible en: http://pisuerga.inf.ubu.es/cgosorio/ALeF/UD1/compilacion.pdf, consultado el 28/01/09.

- La etapa de síntesis (back-end o etapa final) agrupa aquellas fases que dependen principalmente[14] de la arquitectura de la computadora (i860, 88000, OpenRISC, SPARC, 8086, 8088, x86-64, etc.). Una de sus fases involucra al:

Optimizador de código intermedio

Transforma la representación intermedia a otra equivalente pero más eficiente.[15]

1.6.1.6 Programa objeto

Es el código generado por el proceso de compilación en lenguaje máquina o bytecode (en el caso de Java) y se distribuye en varios archivos que corresponden a cada código fuente compilado. Después se obtiene un programa ejecutable para lo cual se han enlazado todos los archivos de código fuente con un programa llamado enlazador o linker.

Generador de código objeto

Genera un programa binario equivalente al código fuente para su ejecución en la computadora de arquitectura específica, añadiéndole posiblemente rutinas de bibliotecas y códigos de inicialización.[16]

1.6.1.7 Optimizador de código

Finalmente puede haber un optimizador de código para mejorar aún más el código generado haciéndolo de menor tamaño y más rápido.[17]

---

[14] Véase, César Ignacio García Osorio: *Introducción a los compiladores*, material en línea, disponible en: http://pisuerga.inf.ubu.es/cgosorio/ALeF/UD1/compilacion.pdf, consultado el 28/01/09.
[15] loc. cit.
[16] loc. cit.
[17] loc. cit.

1.6.1.8 Tabla de símbolos

Un compilador debe tener un determinado comportamiento ante programas erróneos. Este proceso se agrupa en la fase de manejo de errores. Cada una de las fases anteriores interactúa con este manejador.[18]

Otro elemento de la compilación es la tabla de símbolos a la que accede cada una de las fases a través del administrador de la tabla de símbolos. En esta tabla se almacena información sobre variables, constantes, funciones y procedimientos, tipos de datos, memoria asignada, ámbito, alcance, etc.[19] Es una estructura de datos que contiene un registro por cada identificador con sus atributos. Es como un diccionario en donde las fases anteriores introducen esta información sobre los identificadores y después la utilizan de varias formas.

1.6.2 Interpretación

Hay dos formas de ejecutar un programa escrito en alto nivel:

- Compilación: traducir todo el programa fuente a otro programa equivalente en código máquina. Entonces se ejecuta el programa obtenido.
- Interpretación: interpretar las instrucciones del programa fuente y ejecutarlas una por una.

Hay lenguajes de programación que utilizan un programa intérprete traductor, el cual analiza directamente la descripción simbólica del programa fuente y realiza las instrucciones dadas.[20]

---

[18] loc. cit.
[19] loc. cit.
[20] Cf. Lenguajes de programación: “Lenguajes de programación”, material en línea, disponible en: http://www.lenguajes-de-programacion.com/lenguajes-de-programacion.shtml, consultado el 28/01/09.

El intérprete en los lenguajes de programación simula una máquina virtual, donde el lenguaje de máquina es similar al lenguaje fuente.[21]

La ventaja del proceso intérprete es que no necesita de dos fases para ejecutar el programa (la fase de compilación y la fase de carga de datos que son propias de los lenguajes compilados), sin embargo su inconveniente es que la velocidad de ejecución es más lenta ya que debe analizar e interpretar las instrucciones contenidas en el programa fuente.[22]

Los intérpretes realizan la traducción y ejecución de forma simultanea, es decir, un intérprete lee el código fuente y lo va ejecutando al mismo tiempo.

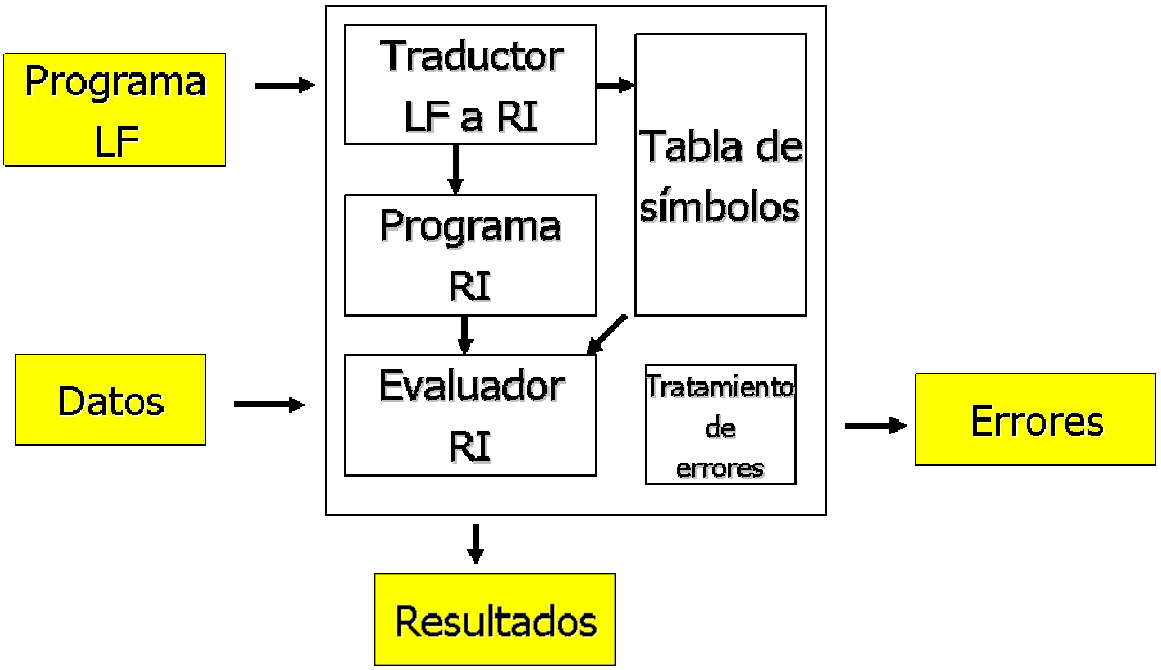


Figura 1. 5. Entradas y salidas del proceso de interpretación[23]

[21] Cf. Lenguajes de programación: "Lenguajes de programación", material en línea, disponible en: http://www.lenguajes-de-programacion.com/lenguajes-de-programacion.shtml, consultado el 28/01/09.
[22] loc. cit.
[23] Fuente: Ismael Perea: *Apuntes de programación para tercer semestre de la licenciatura en informática*, 2008.

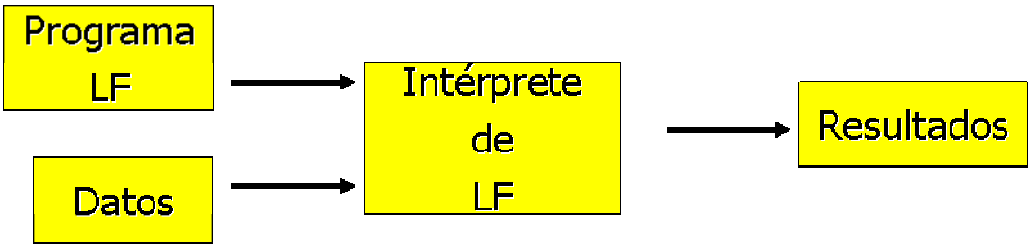


Figura 1.6. El proceso de interpretación en detalle[24]

Las diferencias entre un compilador y un intérprete básicamente son:

- Un programa compilado puede funcionar por sí solo mientras que un código traducido por un intérprete no puede funcionar sin éste.
- Un programa traducido por un intérprete puede ser ejecutado en cualquier máquina ya que, cada vez que se ejecuta el intérprete, tiene que compilarlo.
- Un archivo compilado es mucho más rápido que uno interpretado.[25]

Bibliografía del tema 1

1. Farret, *Introducción a la programación. Lógica y diseño*, 4ª ed., México, Thomson Learning, 2002.
2. Louden, Kenneth C., *Programming Languages: Principles and Practice,* 2ª ed., Brooks Cole/Course Technology, 2003.
3. Pratt, T.W., *Lenguajes de programación*, México, Prentice-Hall, 1987.
4. Sebesta, Robert, *Concepts of Programming Languages*, 3ª ed., EE.UU. Addison Wesley, 1986.

[24] Fuente: Perea, Ismael: Apuntes de programación para tercer semestre de la licenciatura en informática, 2008
[25] Sara Álvarez: "Proceso de traducción de los lenguajes de programación", 24/02/06, material en línea, disponible en: http://www.desarrolloweb.com/articulos/2387.php, recuperado el 28/01/09.

5. Sethi, Ravi, *Lenguajes de programación*, México, Addison Wesley, 1992.
6. Smith, Jo Ann, *C++ Programación Orientada a objetos*, México, Thomson Learning, 2002.
7. Wang, Paul, *Java con programación orientada a objetos y aplicaciones en la WWW*, México, Thomson Learning, 2002.

Actividades de aprendizaje

A.1.1. Investiga la biografía de los creadores de la notación BNF: John Backus y Peter Naur.

A.1.2. Imprime la *timeline* de los lenguajes de programación que viene anexa a este apunte. Escoge algún lenguaje que te llame la atención y sigue su evolución. Márcala con algún resaltador de textos y busca con qué otros lenguajes se ha relacionado y/o fusionado.

A.1.3. Realiza un cómic sobre la historia y evolución de los lenguajes de programación. Busca antecedentes y creadores de los siguientes lenguajes: autocódigos, FORTRAN, COBOL, Algol 60, LISP, APL, BASIC, APL, Simula, Algol 68, Pascal, PROLOG, Small Talk, C, Modula-2, ADA, C++, Delphi, Python, Visual Basic, Perl, Java, JavaScript, PHP, C#, VB.NET y Ruby. Puedes utilizar para su elaboración animación digital, video, etc.

Ejemplo de cómic elaborado por alumnos de la carrera de informática.

A.1.4. En los sistemas operativos Unix o Linux existe un compilador de C que se llama gcc. Utiliza una cuenta en un servidor Linux o Unix y contesta las siguientes preguntas:

a) La instrucción **$gcc holaMundo.c** genera:

a) holaMundo
b) holaMundo.s
c) a.out
d) holaMundo.o

b) La instrucción **$gcc holaMundo.c –o holaMundo** genera:

a) holaMundo.s
b) holaMundo
c) a.out
d) holaMundo.exe

c) La instrucción **$gcc holaMundo.c –c** genera:

a) a.out
b) holaMundo.o
c) holaMundo.s
d) holaMundo

d) La instrucción **$gcc holaMundo.c –S** genera:

a) código binario
b) código *assembly*
c) código *assembler*
d) bytecode

Cuestionario de autoevaluación

1. ¿Qué es un lenguaje de programación?
2. ¿Qué es un lenguaje traductor?
3. ¿Qué es un compilador?
4. ¿Cuál es la diferencia entre un intérprete y un compilador?
5. ¿Cuáles son las etapas de la compilación?
6. ¿Qué es un analizador sintáctico?

7. ¿Qué es un analizador semántico?

8. ¿Para qué sirve una tabla de símbolos?

9. ¿Qué es un metalenguaje?

10 ¿Qué es el léxico de un lenguaje de programación?

Examen de autoevaluación

Selecciona el inciso de la respuesta correcta de las siguientes preguntas:

1. Un paradigma de programación es:

a) Un conjunto de teorías generales, suposiciones, leyes y técnicas que forman una visión del mundo
b) Una forma de representar y manipular el conocimiento
c) Un enfoque particular o filosofía para la construcción de programas
d) Una metodología de desarrollo de software

2. Los cuatro principales paradigmas de programación son:

a) Imperativo, OO, funcional y procedimental
b) OO, lógico, orientado a aspectos e imperativo
c) Lógico, imperativo, funcional y OO
d) Orientado a aspectos, OO, estructural y funcional

3. La expresión regular que describe los siguientes lexemas: _REQUEST, _4NUM y S_NOMBRE del token “identificador” es:

a) <identificador>::= letra (letra | dígito | ‘_’)*, <letra> ::= (‘a’..’z’) U (‘A’..’Z’) y <dígito> ::= ‘0’..’9’
b) <identificador>::= letra (letra | dígito | ‘_’)*, <letra> ::= ‘A’..’Z’ y <dígito> ::= ‘0’..’9’
c) <identificador>::= (letra | ‘_’) (letra | dígito | ‘_’)*, <letra> ::= (‘a’..’z’) U (‘A’..’Z’) y <dígito> ::= ‘0’..’9’
d) <identificador>::= (‘_’ | letra) (letra | dígito | ‘_’)*, <letra> ::= ‘A’..’Z’ y <dígito> ::= ‘0’..’9’

4. En esta estructura de datos se almacena información sobre variables, constantes, funciones y procedimientos, tipos de datos, memoria asignada, ámbito, alcance, etc. Es como un diccionario en donde cada fase de la compilación introduce esta información y después la utilizan de varias formas.

a) Árbol sintáctico
b) Tabla de errores
c) Código intermedio
d) Tabla de símbolos

5. A partir del modelo de Von Neumann los programadores se dieron cuenta de que sería de gran ayuda asignar símbolos nemotécnicos a los códigos de instrucción, así como a las localizaciones de memoria, y nació el:

a) Lenguaje ensamblador
b) Lenguaje máquina
c) Lenguaje de marcado
d) Lenguaje objeto

6. Un ejemplo de metalenguaje es:

a) COBOL
b) UML
c) XML
d) Java

7. Un [...] es un traductor que toma un programa fuente, lo traduce y a continuación lo ejecuta de forma secuencial.

a) Compilador
b) Intérprete
c) Analizador sintáctico
d) Optimizador de código

8. Las abstracciones de control básicas incluyen enunciados o sentencias que combinan unas cuantas instrucciones de máquina en una sentencia abstracta más comprensible. x = x + 3 es un ejemplo de este tipo de abstracción y se conoce como:

a) Procedimiento
b) Comparación
c) Asignación
d) Declaración

9. Es como la ortografía de un lenguaje natural.

a) Sintaxis
b) Semántica
c) Léxico
d) Gramática

10. El código de programación que no se compila y que es interpretado por el navegador es:

a)	b)	c)	d)

public class HolaMundo extends Applet {
public void paint(Graphics g) {
g.drawString("Hola mundo",0,25) ;
}
}
	<HTML>
	<HEAD>
	<TITLE>Hola mundo</TITLE>
	</HEAD>
	<BODY>
	Hola mundo
	
	</BODY>
	</HTML>
		class HelloWorldSwing {
		static public void main(String args[]) {
		javax.swing.JOptionPane.showMessageDialog(null,"Hola mundo");
		}
		}
			<SCRIPT language="JavaScript" type="text/javascript">
			document.write("Hola Mundo");
			</SCRIPT>

Tema 2. Programación imperativa

Objetivo particular

Al término de esta unidad, el alumno identificará los conceptos más importantes que dan soporte y fundamento al paradigma de programación imperativa y podrá aplicarlos en la resolución de problemas algorítmicos propios de la automatización de la información y manipulación de datos.

Temario detallado

2.1 Definición

2.2 Modularidad

2.3 Concepto de celda de memoria variable

2.4 Operaciones de asignación

2.5 Operaciones de repetición

2.6 Secuencia de transformación de datos

2.7 Campos de aplicación

Introducción

Uno de los paradigmas más representativos de la programación y de hecho el primer paradigma formalmente aceptado es el imperativo. Imperar significa mandar, ordenar, y eso es exactamente lo que hacemos al programar, ordenarle a la computadora algo que queremos que haga. Los primeros lenguajes de programación nacieron de la necesidad del programador de indicarle a la computadora una serie de instrucciones a seguir para resolver un problema. En este tema veremos los conceptos fundamentales del paradigma imperativo que son: celda de memoria variable, operaciones de asignación y operaciones de repetición.

2.1 Definición

El paradigma imperativo apareció en los 50 con los primeros lenguajes de programación. También es llamado procedimental o algorítmico. Se llama así porque está basado en comandos u órdenes (enunciados imperativos) que actualizan variables que están almacenadas en memoria. Está basado en el modelo de Von Neumann, el cual define una máquina capaz de ejecutar una serie de instrucciones de forma secuencial, una tras otra. Estas instrucciones deben estar almacenadas en memoria principal para poder ser leídas y ejecutadas por la CPU (Unidad Central de Proceso).

2.2 Modularidad

La programación imperativa se rige por dos conceptos básicos para la construcción de programas: la estructura y el módulo. De ahí que se hable de programación estructurada y de programación modular. Los dos tipos de programación imperativa señalan que un programa se debe dividir en subprogramas, segmentos o módulos para hacerlo más legible y manejable. Posteriormente se enlazan para lograr la funcionalidad deseada. Esta técnica también es conocida como refinamiento sucesivo, divide y vencerás ó análisis descendente (Top-Down).

Cada uno de los módulos en que se dividió el programa tiene una tarea bien definida y algunos necesitan de otros para poder operar. En caso de que un módulo necesite de otro, puede comunicarse con éste mediante una interfaz de comunicación.

Cada módulo es un procedimiento o función o conjunto de éstas, incluso son librerías con determinada funcionalidad que se pueden llamar desde un programa.

2.3 Concepto de celda de memoria variable

La programación imperativa se basa en tres conceptos importantes: celda de memoria variable, operaciones de asignación y operaciones de repetición. La memoria de una computadora juega un papel primordial para que un programa pueda ser ejecutado. En ella se almacenan las instrucciones y los datos con los que el programa trabajará. La memoria es un conjunto de celdas identificadas por una dirección que es única: dos celdas no pueden tener la misma dirección. Pero en lugar de que en nuestros programas hagamos referencia a las direcciones de memoria (que están definidas en números hexadecimales), las "bautizamos" con un nombre (variable), así es más fácil referenciar el contenido de una celda de memoria.

2.4 Operaciones de asignación

Cada valor calculado en un programa debe ser "almacenado", es decir asignado a una celda. Esta es la razón de la importancia de la **sentencia de asignación** en el paradigma imperativo. Las asignaciones se definen así:[26]

Expresión 1 = Expresión 2

Donde:

Expresión 1	es la localidad de memoria ya "bautizada"
Expresión 2	es el valor que le estamos asignando
=	lexema de asignación

[26] Rafael Menéndez-Barzanallana; "Metodologías usadas en ingeniería del software: Paradigma imperativo, material en línea, disponible en: http://www.wikilearning.com/curso_gratis/metodologias_usadas_en_ingenieria_del_software-paradigma_imperativo/3618-4, recuperado el 28/01/09.

Existen tres tipos de asignación:

Aritmética: se refiere a cuando asignamos un valor numérico derivado de alguna operación aritmética. La variable que recibe el valor tuvo que haber sido declarada como tipo de dato numérico como *integer*, *long* o *double*, salvo algunas excepciones como en PHP o Ruby, en donde no hay declaración de variables. Ejemplo:

valor = 3 + 5 + 9;

Lógica: se refiere a cuando asignamos un valor de cierto o falso a una variable. La variable que recibe el valor tuvo que haber sido declarada como tipo de dato *boolean*, salvo algunas excepciones como en PHP o Ruby, en donde no hay declaración de variables. Ejemplo:

valor = 20 < 5; (el valor que se asigna a esta variable es *false*)

Cadena de caracteres: se refiere a cuando asignamos como valor una cadena de caracteres (o uno solo, incluyendo el espacio en blanco) a una variable. La variable que recibe el valor tuvo que haber sido declarada como tipo de dato *string* o *char*, salvo algunas excepciones como en PHP o Ruby, en donde no hay declaración de variables. Ejemplo:

valor = “este es un ejemplo de asignación de cadena de caracteres”;

2.5 Operaciones de repetición

Un programa imperativo, normalmente realiza su tarea ejecutando repetidamente una secuencia de pasos elementales, ya que en este modelo computacional la única forma de ejecutar algo complejo es repitiendo una secuencia de instrucciones.[27]

El flujo lógico de un programa se controla a través de estructuras, que pueden ser secuenciales, repetitivas (iterativas), selectivas y de salto. Las operaciones de repetición o iterativas son implementadas en un programa de alto nivel con las sentencias *for*, *while* y *repeat*.

2.6 Secuencia de transformación de datos

Una vez que se ha determinado qué datos son los que un programa va a necesitar, se les asocia una dirección de memoria y se efectúa una secuencia de transformaciones en los datos hasta obtener el valor esperado. Se transforman los datos desde un estado *inicial* hasta un estado *final* usando *reglas* de transformación. Se crea un grafo de estados intermedios con estas reglas y se analiza cuál es el estado intermedio más próximo al objetivo final.

[27] "Paradigmas, Imperativo", 15/08/03, material en línea, disponible en: http://wilucha.com.ar/Paradigma/A_ParaImpera.html, recuperado el 28/01/09.

Ejemplo:

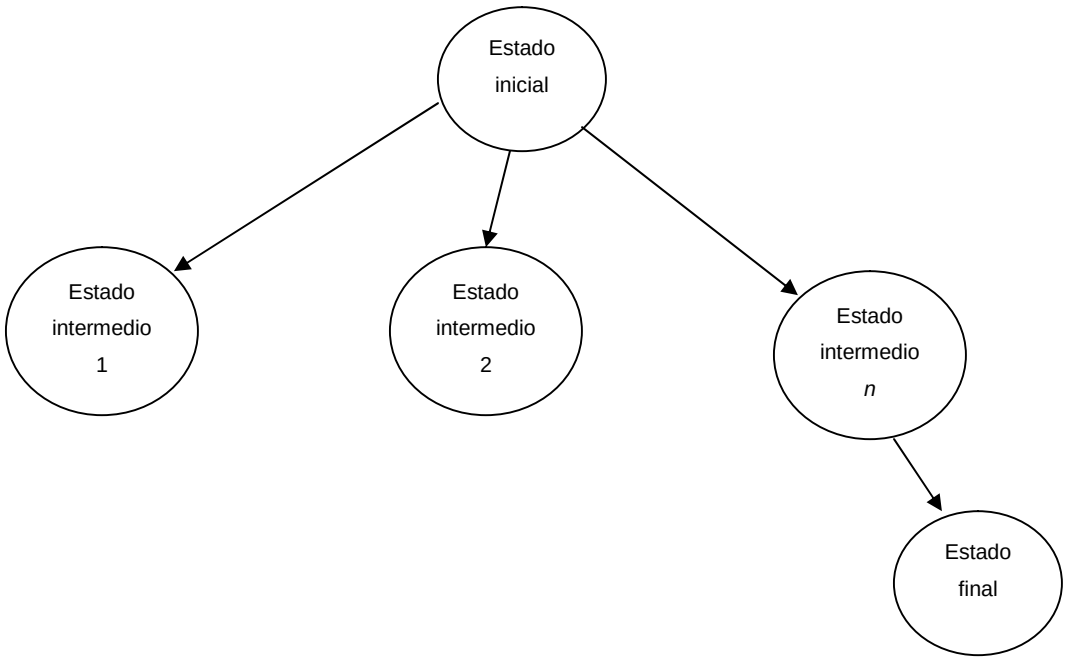


Figura 2.1. Representación de un grafo de estados del proceso de transformación de datos[28]

2.7 Campos de aplicación

Los lenguajes imperativos pueden resolver prácticamente cualquier problema en cualquier área: desde simples hasta complejos cálculos matemáticos. Se pueden hacer cualquier tipo de aplicaciones: de nóminas, de control aéreo, de inteligencia artificial, de control de dosis de medicamentos, para cajeros automáticos, para naves espaciales, para dispositivos móviles, aplicaciones en línea y en tiempo real, pago de impuestos, etc. Hay que recordar que fue el primer paradigma que le vino a poner orden a la manera de hacer programas, y por tanto su filosofía marcó la línea a seguir para resolver problemas de la vida cotidiana.

[28] Fuente: Ismael Perea: A*puntes de Programación del tercer semestre de la carrera de informática.* 2008.

Bibliografía del tema 2

1. Berlage, Thomas, *Concepts and paradigms*, Wokingham, Addison Wesley, 1991.
2. Farret, *Introducción a la programación. Lógica y diseño*, 4ª ed., México, Thomson Learning, 2002.
3. Knuth, Donald, *The art of Computer Programming*, 2ª ed., Reading Mass., Addison Wesley, 1977, vol. 1, Fundamental Algorithms.
4. López Román, Leobardo, Programación *estructurada. Un enfoque algorítmico*. México, Alfaomega, 2002.
5. _____, *Programación estructurada en TurboPascal 7*, México, Alfaomega, 2002.
6. Louden, Kenneth C., *Programming Languages: Principles and Practice*, Second Edition, Brooks Cole, 2003.
7. Pratt, T.W., *Lenguajes de programación*, México, Prentice-Hall, 1987.
8. Sebesta, Robert, *Concepts of Programming Languages*, 3ª ed., EE.UU., Addison Wesley, 1986.
9. Sethi, Ravi, *Lenguajes de programación*, México, Addison Wesley, 1992.
10. Wilson, L. B., y R. Clark, *Comparative Programming Languages*, EE.UU., Addison Wesley, 1988.

Actividades de aprendizaje

A.2.1. Visita la página http://www.tiobe.com y consulta el índice de los 20 lenguajes de programación más populares según este organismo. Revisa cuál es el paradigma que tiene más presencia en el top 20.

A.2.2. Programa el "¡Hola Mundo!" en C. Compáralo con el "¡Hola Mundo!" en Java (también prográmalo o busca el código en la Web). Explica las diferencias en ambos lenguajes para programar lo mismo.

A.2.3. Programa el siguiente código en C y explica el concepto de memoria variable a partir de los valores que se obtienen:

```
//Apuntadores
#include<stdio.h>
int main(){
        int alpha=1;
        int *beta;    //el * permite una dirección de memoria como
valor asignado a una variable
        alpha=1;
        beta=&alpha;
        printf("El valor %d esta almacenado en la dirección
%u\n",alpha,&alpha);
        printf("El valor %u esta almacenado en la dirección
%u\n",beta,&beta);
        printf("El valor %d esta almacenado en la dirección
%u\n",*beta,&beta);

return 0;

}
```

A.2.4. Busca en la Web el algoritmo de las torres de Hanoi. Impleméntalo en C agrupando las operaciones que vayas a utilizar en funciones.

Cuestionario de autoevaluación

1. ¿Qué es el paradigma imperativo?
2. ¿Qué es el concepto de celda de memoria variable?
3. ¿Qué son las operaciones de asignación?
4. ¿Qué son las operaciones de repetición?
5. ¿Qué es la transformación de datos?

Examen de autoevaluación

Contesta las siguientes preguntas eligiendo la opción correcta.

1. Paradigma de programación que se caracteriza por un modelo abstracto de la computadora que consiste en un almacenamiento en memoria de datos y cálculos codificados y la ejecución de una secuencia de comandos que modifican el contenido de ese almacenamiento:

a) lógico
b) algorítmico
c) orientado a objetos
d) funcional

2. Son los tres conceptos principales del paradigma imperativo:

a) int x;	|	if (x==0)	|	x=0
b) var x: integer;	|	while (x >0)	|	x=3
c) float x;	|	return x	|	x=3+8
d) boolean x;	|	switch (true)	|	x=true

3. Los tres tipos de asignación son:

a) aritmética, lógica y por parámetros
b) de cadena de caracteres, lógica y funcional
c) funcional, de cadena de caracteres y aritmética
d) lógica, de cadena de caracteres y aritmética

4. Son estructuras de control iterativas:

a) break, continue y return
b) if, switch y case
c) for, while y repeat
d) if, while y goto

5. Para guiar los cálculos o flujo de un programa (transformación de los datos a partir de un algoritmo) se utilizan cálculos o flujo de un programa de tipo:

a) secuenciales, selectivas y de salto
b) de salto, repetitivas y secuenciales
c) repetitivas, selectivas y de salto
d) selectivas, secuenciales e iterativas

Tema 3. Programación orientada a objetos

Objetivo particular

Al término de este tema, el alumno explicará los conceptos más importantes que dan soporte y fundamento al paradigma de programación orientado a objetos (clase, objeto, herencia, polimorfismo y encapsulamiento) y podrá aplicarlos en la resolución de problemas algorítmicos propios de la automatización de la información y manipulación de datos.

Temario detallado

3.1 Definición

3.2 Clases y objetos

3.3 Atributos y métodos

3.4 Relaciones estáticas y dinámicas entre objetos

3.5 Herencia

3.6 Polimorfismo

- 3.6.1 Sobrecarga
- 3.6.2 Sobreescritura

3.7 Encapsulamiento (ocultamiento de la información)

3.8 Diagrama de clases

3.9 Campos de aplicación

Introducción

La programación orientada a objetos (POO) surge de la necesidad de contar con lenguajes que pudieran implementar soluciones computables de una forma más parecida a la que utilizamos en la vida real, ya que nosotros concebimos nuestro mundo como un conjunto de cosas u objetos con propiedades o atributos que los definen como tal.

3.1 Definición

Es el paradigma que define objetos y clases como la base para la programación. Cada objeto está definido por sus atributos y su comportamiento está definido por las operaciones que dichos objetos pueden hacer. La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.[29] Su uso se popularizó en los 90. Actualmente son muchos los lenguajes de programación que soportan la orientación a objetos. Ejemplos: Java, C++, Smalltalk, PHP y Ruby.

3.2 Clases y objetos

Una **clase** es una plantilla que encapsula los datos y las abstracciones de datos necesarios para describir el contenido y comportamiento de alguna entidad del mundo real. Es una descripción generalizada (patrón o plantilla) que describe una colección de objetos similares.

Un **objeto** es una instancia de una clase específica: los objetos heredan los atributos y operaciones de una clase (su clase padre). Todos los objetos de una clase tienen el mismo conjunto de atributos y el mismo número de operaciones. Difieren solamente en los valores de sus atributos respectivos.

Para modelar una clase y un objeto se utiliza un lenguaje de modelado llamado **UML** (Lenguaje Unificado de Modelado):

[29] "Programación orientada a objetos" 01/08, material en línea, disponible en: http://cea-elmrabet.blogspot.com/2008/01/oop.html, recuperado el 28/01/09.

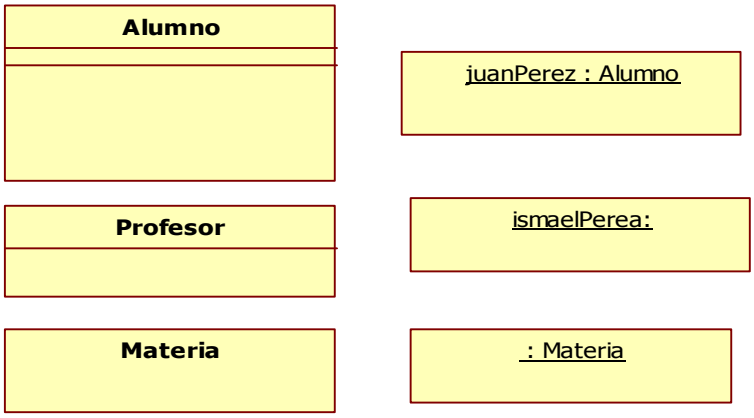
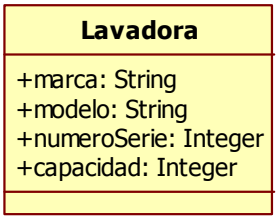


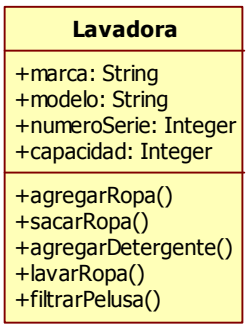
Figura 3.1. Tokens o símbolos de UML que se utilizan para modelar clases y objetos

3.3 Atributos y métodos

Un **atributo** es una propiedad, rasgo o característica de una clase y describe un rango de valores que la propiedad podrá contener en los objetos (instancias).



Las **operaciones** también llamadas **métodos** o **servicios** son algo que la clase puede hacer o que nosotros u otra clase pueden hacer a una clase.



La **responsabilidad** es una descripción de lo que hará la clase, es decir, de lo que sus atributos y operaciones intentan realizar en conjunto.

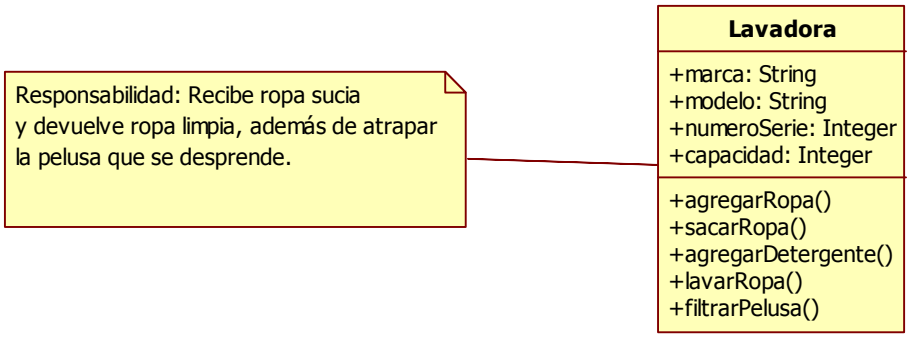


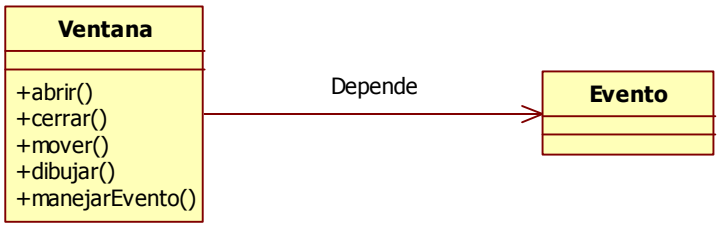
Figura 3.2. Ejemplo de responsabilidad

3.4 Relaciones estáticas y dinámicas entre objetos

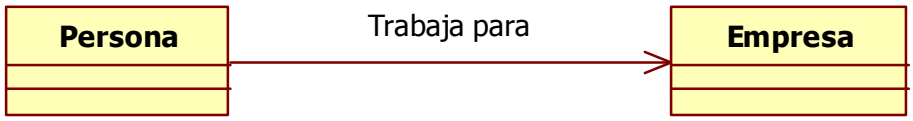
Entre clases se pueden entablar relaciones, ya que muy difícilmente una clase está aislada del resto. Una relación es una conexión entre elementos. Las tres relaciones más importantes son:

- Dependencia
- Generalización (herencia)
- Asociación:
    - Agregación
    - Composición

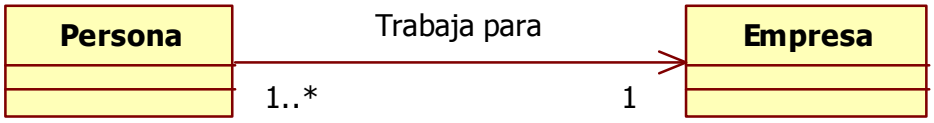
Una **dependencia** es una relación de uso que declara que un cambio en la especificación de un elemento afecta a otro elemento que la utiliza, pero no necesariamente a la inversa.

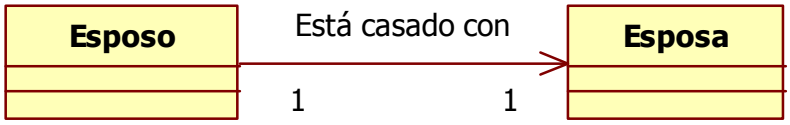


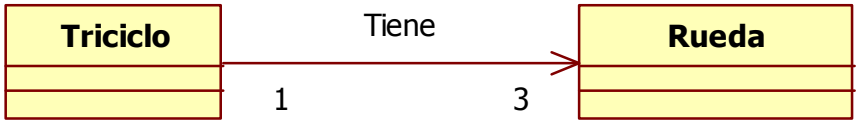
Una **asociación** es una relación estructural que especifica que los objetos de un elemento están conectados con los objetos de otro.



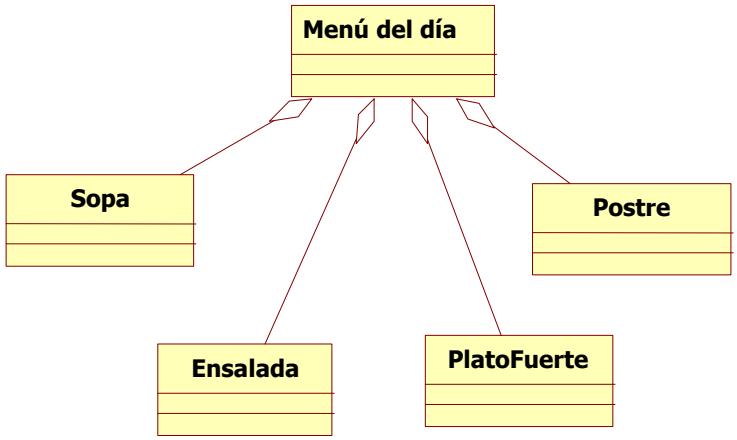
Multiplicidad: Como ya se dijo, una asociación representa una relación estructural entre objetos. En algunas ocasiones es necesario señalar cuántos objetos de una clase pueden relacionarse con un objeto de la clase asociada. El "cuántos" se conoce como multiplicidad.



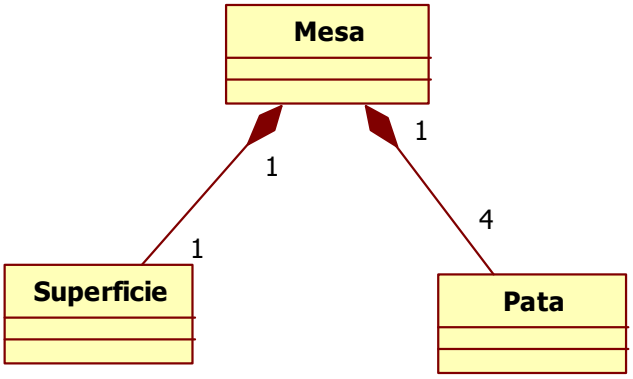




Agregación: A veces se desea modelar una relación “todo-parte”, en la cual una clase representa una cosa grande (el “todo”), que consta de elementos más pequeños (las “partes”). Dicho de otra forma, una clase consta de otras clases. Este tipo de asociación se llama agregación.



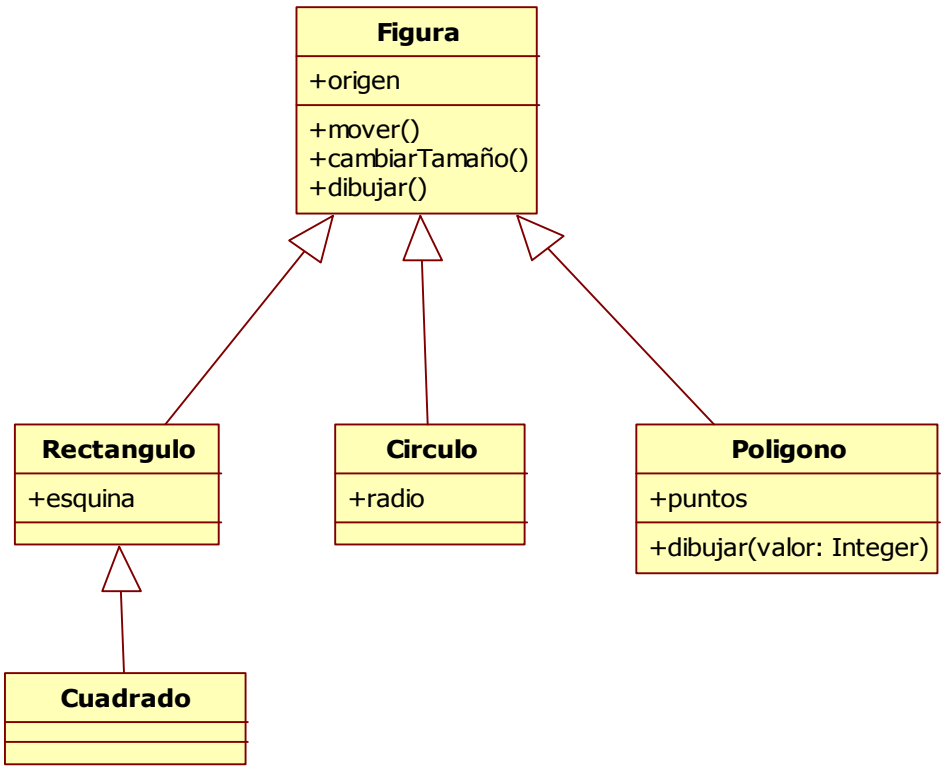
Composición: Una composición es un tipo especial de agregación donde cada componente dentro de una composición puede pertenecer tan sólo a un todo.



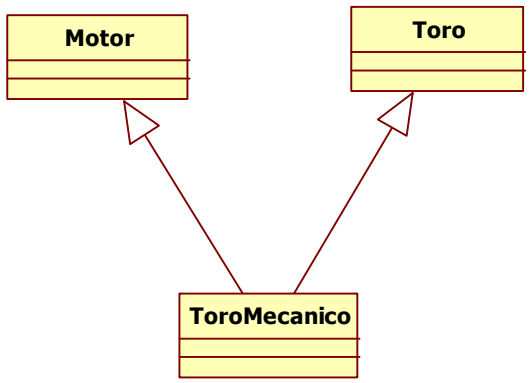
3.5 Herencia (generalización)

Una **generalización** es una relación entre un elemento general (llamado superclase o clase padre) y un caso más específico (especialización) de ese elemento (llamado subclase o clase hija). La generalización es la

implementación del concepto de **herencia**. En la generalización un hijo puede sustituir al padre. La clase hija hereda atributos y operaciones de su o sus padres. Una clase hija puede tener sus propios atributos y operaciones e incluso redefinir o implementar dichas operaciones a través del concepto de polimorfismo.



Herencia múltiple: Una clase puede ser hija de dos o más clases. A esto se le llama herencia múltiple.



3.6 Polimorfismo

Es cuando las operaciones de los objetos de una misma clase tienen comportamientos diferentes. Incluso el comportamiento de la clase padre difiere del comportamiento de las clases hijas. Es obvio que comparten el mismo nombre. Existen dos tipos de polimorfismo:

3.6.1 Sobrecarga

Es cuando un método (operación) de una clase recibe diferentes parámetros para realizar su tarea.

3.6.2 Sobreescritura

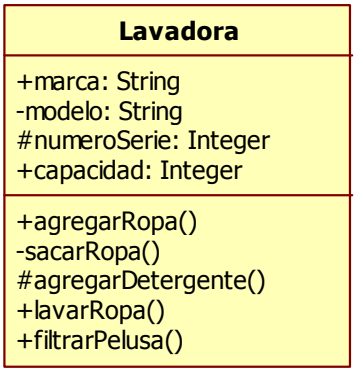
Es cuando el comportamiento de un método (operación) es diferente con respecto a las demás instancias o incluso al mismo padre. Esto significa que se redefine el código del método. Cuando esto pasa, es necesario definir al método padre como abstracto; esto es, no definir ningún comportamiento en el

padre, solo el nombre, para dejarlo a las instancias que lo definan según sus necesidades.

3.7 Encapsulamiento (ocultamiento de la información)

La visibilidad se aplica a los atributos y a las operaciones de una clase, y a la misma clase también. Establece la proporción en que otras clases podrán utilizarlos. Existen tres niveles de visibilidad:

- Nivel **público**: la funcionalidad se extiende a otras clases. Para el nivel público se utiliza el lexema +
- Nivel **privado**: solo la clase original puede utilizar el atributo u operación. Para el nivel privado se utiliza el lexema –
- Nivel **protegido**: la funcionalidad se otorga sólo a las clases que se heredan de la clase original. Para el nivel protegido se utiliza el lexema #

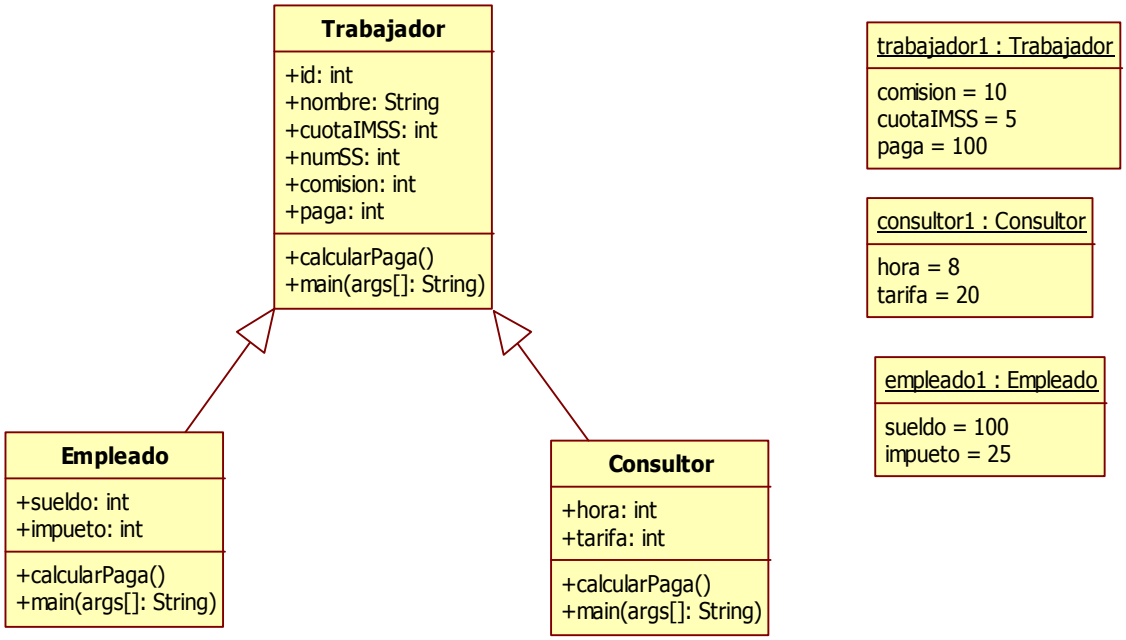


El encapsulamiento permite reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Así aumenta la cohesión de los componentes del sistema. Cada objeto está aislado del exterior y expone una interfaz a otros objetos que describe cómo pueden interactuar con él. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden

acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas.

3.8 Diagrama de clases

Un diagrama de clases es una representación gráfica que modela a las clases y sus relaciones. La notación estándar para un diagrama de clases es la que define UML. Ejemplo:



3.9 Campos de aplicación

Al igual que el paradigma imperativo, prácticamente con los lenguajes orientados a objetos se puede hacer cualquier cosa: controlar el plan de vuelo de una aeronave, aplicaciones para Web, aplicaciones gráficas, dispositivos móviles, videojuegos, realidad virtual, chips de memoria, etc. Algunos ejemplos de lenguajes son:

- Ada
- C++
- C#
- Delphi
- Eiffel
- Java
- Objective-C
- Oz
- PHP (en su versión 5)
- PowerBuilder
- Python
- Ruby
- Smalltalk
- VB.NET

Bibliografía del tema 3

1. Alfonseca, Manuel, et. al., *Compiladores e intérpretes: teoría y práctica*, Madrid, Pearson Educación, 2006.
2. Berlage, T., *Concepts and paradigms*, Wokingham, Addison Wesley, 1991.
3. Bronson, Gary J., *C++ para Ingeniería y ciencias*, México, Thomson Learning, 1999.
4. Ceballos, Francisco Javier, *Microsoft Visual C++6 aplicaciones para Win32*, Madrid, Alfaomega/Rama, 2002.
5. Graham, Ian, *Métodos orientados a objetos*, México, Addison Wesley, 1995.
6. Louden, Kenneth C., *Programming Languages: Principles and Practice*, Second Edition, Brooks Cole, 2003.
7. Orós, Juan Carlos, *Diseño de páginas Web interactivas con JavaScript*, 2ª ed., Madrid, Alfaomega/Rama, 2001.

8. Pratt, T.W., *Lenguajes de programación*, México, Prentice-Hall, 1987.
9. Sebesta, Robert, *Concepts of Programming Languages*, 3ª ed., EE.UU., Addison Wesley, 1986.
10. Sethi, Ravi, *Lenguajes de programación*, México, Addison Wesley, 1992.
11. Smith, Jo Ann, *C++ Programación Orientada a objetos*, México, Thomson Learning, 2002.
12. Wang, Paul, *Java con programación orientada a objetos y aplicaciones en la WWW*, México, Thomson Learning, 2002.
13. Wilson, L. B., y R. Clark, *Comparative Programming Languages*, EE.UU., Addison Wesley, 1988.

Actividades de aprendizaje

A.3.1. Visita, conoce y recuerda la página de la OMG http://www.omg.org. Todo lo relacionado con el paradigma orientado a objetos, UML, lenguajes de programación, estándares, herramientas CASE de modelado, etc., se pueden encontrar en este sitio, ya que la OMG es el organismo que rige, autoriza y certifica todo lo relacionado con el POO.

A.3.2. Investiga qué es UML. Instala en tu computadora StarUML u otra herramienta CASE para el modelado de UML. Modela algunas clases y posteriormente genera el código en Java o C++. Compara el código con el que hubieras hecho por separado.

A.3.3. Implementa seis clases con atributos y operaciones que ejemplifiquen una asociación de tipo agregación.

A.3.4. Visita http://www.youtube.com y busca el video "*Java is everywhere*" http://www.youtube.com/watch?v=guXCmQDy9Es y http://www.youtube.com/watch?v=Jg-vhAdtauE&feature=related. Saca tus conclusiones de qué otras cosas se pueden hacer con Java además de videojuegos o aplicaciones para Web.

A.3.5. Ejecuta el siguiente código y explica la diferencia entre Java y JavaScript. Los dos programas son OO, pero no se ejecutan de la misma forma:

```
<HTML>
<!—-este es un ejemplo de POO-->
       <HEAD>
              <TITLE>EJEMPLO DEL HOLA MUNDO CON UN
              INTERPRETE Y UN COMPILADOR</TITLE>
       </HEAD>
       <BODY>
              <B>
              <CENTER>
Ejemplo de un programa interpretado y
compilado elaborado por:
<FONT COLOR='blue' SIZE="4">Ismael Perea</FONT>
</CENTER>
</B>
<SCRIPT LANGUAGE="JavaScript" type="text/javascript">
document.write("Hola Mundo esto es una prueba ");
</SCRIPT>
<P>
<APPLET width="208" height="50" CODE="HolaMundo.class"></APPLET>
<P>
Fin del ejemplo
</BODY>
</HTML>
```

Este archivo se debe guardar con el nombre de holaMundo.html

```
//
// Applet HolaMundo de ejemplo
//

/* este es un ejemplo de un applet
en java para la clase de POO
elaborado por: Ismael Perea
*/

import java.awt.Graphics;
import java.applet.Applet;

public class HolaMundo extends Applet {

	public void paint( Graphics g ) {
		g.drawString( "¡Hola alumnos!",0,25 ) ;
	}
}
```

Este archivo se debe guardar con el nombre de HolaMundo.java y compilar con javac, y así obtendremos el archivo HolaMundo.class

Cuestionario de autoevaluación

1. ¿Qué es el paradigma orientado a objetos?
2. ¿Qué es una clase?
3. ¿Qué es un objeto?
4. ¿Qué es un atributo?
5. ¿Qué es una operación?
6. ¿Qué es una relación?
7. ¿Qué es la responsabilidad de una clase?
8. ¿Qué es una dependencia?
9. ¿Qué es la generalización?
10. ¿Qué es el polimorfismo?

Examen de autoevaluación

Contesta las siguientes preguntas eligiendo la opción correcta.

1. Esta relación es del tipo “tiene un”, o sea, un objeto del todo tiene objetos de la parte:

a) Especialización
b) Generalización
c) Agregación
d) Dependencia

2. Es una descripción de lo que hará la clase, es decir, de lo que sus atributos y operaciones intentan realizar en conjunto:

a) Responsabilidad
b) Herencia
c) Polimorfismo
d) Instancia

3. Una clase hija es una [...] de una clase padre:

a) Generalización
b) Agregación
c) Composición
d) Especialización

4. Es algo que la clase puede realizar, o que nosotros u otra clase pueden hacer a una clase. Su nombre se escribe en minúsculas si consta de una sola palabra; si son más, se unen iniciando con letra mayúscula:

a) Atributo
b) Operación
c) Responsabilidad
d) Nombre de la clase

5. Token que indica que un atributo u operación es privado:

a) +
b) –
c) #
d) *

6. En la siguiente declaración: `import java.lang.System;` la clase es:

a) Lang
b) Java
c) System
d) Object

7. Token que indica que un atributo u operación es público:

a) –
b) #
c) +
d) *

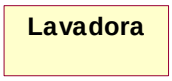
8. Es un tipo de relación cuando una clase utiliza a otra. Declara que un cambio en la especificación de un elemento puede afectar a otro elemento que la utiliza, pero no necesariamente a la inversa:

a) Agregación
b) Dependencia
c) Composición
d) Generalización

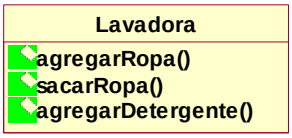
9. Representa una instancia de una clase:

a)

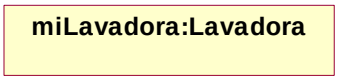
b)



c)



d)



10. Propiedad o característica de una clase que describe un rango de valores que la propiedad podrá contener en las instancias:

a) Operación
b) Polimorfismo
c) Herencia
d) Atributo

TEMA 4 PROGRAMACIÓN FUNCIONAL

Objetivo particular

Al término de este tema, el alumno explicará los conceptos más importantes que dan soporte y fundamento al paradigma de programación funcional (funciones, recursividad y listas) y los podrá aplicar en la resolución de problemas algorítmicos propios de la automatización de la información y manipulación de datos.

Temario detallado

4.1 Definición

4.2 Programas con funciones

4.3 Recursividad

4.4 Listas

4.5 Implementación de algoritmos

4.6 Lenguajes funcionales puros e híbridos

4.7 Introducción a los lenguajes funcionales

4.8 Campos de aplicación

Introducción

Si hasta ahora pensabas que programar en ensamblador, C, Java, o incluso Ruby era todo lo que había en el mundo de los lenguajes de programación, pues no es así. ¿Te imaginas programar el movimiento de los discos de la torre de Hanoi sin una estructura *if* o *case*, o sin el *while* o el *for*, o que no pudieras declarar una variable ni mucho menor le pudieras asignar un valor? Parece algo loco, pero hay un paradigma que en su definición pura no necesita de estos conceptos para realizar programas. Su trabajo lo logra utilizando funciones puras, trasladando el concepto de función matemática a la programación.

4.1 Definición

Es el paradigma de programación que difiere de otros paradigmas ya que define a los programas como función, y trata a las funciones como datos, logrando así minimizar los efectos colaterales de ejecución y la administración automática de la memoria. Con este paradigma se logra gran flexibilidad de los lenguajes, es conciso en la notación y la semántica es fácil de entender.

4.2 Programas con funciones

Un programa es una descripción de un cálculo. Un programa entonces es equivalente a una función matemática.

Una función es una regla que asocia a cada elemento *x* de algún conjunto *X* de valores un elemento *y* único de otro conjunto *Y* de valores. Matemáticamente se define una función así:

y = f(x) ó
f: X → Y

El conjunto X se llama dominio de *f*, y el conjunto Y se llama rango de *f*. La *x* en f(x) se llama **variable independiente**. Y la *y* del conjunto Y de la ecuación y=f(x) se llama **variable dependiente**. Cuando f no está definida para todas las *x* de X, se llama **función parcial** y cuando sí está definida para todas las *x* de X se llama **función total**.

Los programas, procedimientos y funciones se pueden representar por medio de una función. En el caso de un programa *x* representa la entrada y *y* la salida. En un procedimiento o función *x* representa los parámetros y *y* los valores devueltos. En el paradigma funcional no se hace distinción entre programa, función o procedimiento, solo importa los valores de entrada y salida.

4.3 Recursividad

La recursividad en términos de programas es cuando una función o método se llama a sí misma cuantas veces requiera para resolver un problema dado. Pero no quiere decir que sea la mejor forma de hacerlo ni la más eficiente, al contrario, la recursividad es cara en el uso de memoria y exige una mayor capacidad de procesamiento.

El ejemplo clásico de una función recursiva es el cálculo de la factorial de un número. Por ejemplo, 6! es igual a 6 * 5 * 4 * 3 * 2 * 1. Algebraicamente, podemos considerar el cálculo factorial como (n!):

n * (n-1) * (n-2) * (n-3)...(n - (n+1))

La definición recursiva del cálculo factorial en LISP es:

```
defun factorial (n)
  (cond ((zerop n) 1)
          (T
    (* n (factorial (1- n))))
  ) ;_ fin de cond
) ;_ fin de defun
```

4.4 Listas

La lista es el elemento principal cuando se programa en un lenguaje funcional, ya que tradicionalmente una función estará implementada por listas de elementos. Tanto los datos como los programas son listas. De ahí viene el nombre del lenguaje LISP, que es un acrónimo de "LISt Processing". Por cierto, hay un chiste de esto: las listas en LISP están delimitadas por paréntesis, y entonces se dice que el significado de LISP es: "Lost In Stupid Parentheses".

En LISP hay dos tipos de elementos con los que se programa:

- **Átomos**: son los datos elementales y pueden pertenecer a varios tipos: números, caracteres, cadenas de caracteres o símbolos.
- **Listas**: son secuencias de átomos o de listas encerradas entre paréntesis. Además, existe una lista especial, "nil", que es la lista nula, que no tiene ningún elemento.

Hay funciones en LISP cuyos nombres son símbolos (**+** para la suma, ***** para el producto, etc.) por ejemplo: (+ 5 9).

4.5 Implementación de algoritmos

Este paradigma basa su programación en un conjunto de funciones (casi siempre recursivas) y alguna expresión cuya salida representa el resultado de algún algoritmo. Los lenguajes funcionales son declarativos, esto quiere decir que se describen relaciones entre las variables en términos de funciones y reglas de inferencia (procedimiento que infiere hechos a partir de otros hechos conocidos), dejando al traductor la responsabilidad de encontrar el mejor algoritmo para encontrar el resultado buscado. También se basa en el cálculo lambda λ con constantes; este cálculo ayuda a crear valores de funciones sin tener que darles un nombre. Las funciones de este tipo no modifican su salida con entradas iguales, y al no tener efectos colaterales cumplen con la regla de transparencia referencial.

Para implementar un algoritmo hay que considerar que un lenguaje de programación es completo en *Turing,* si tiene valores enteros, funciones aritméticas sobre dichos valores, así como un mecanismo para definir nuevas funciones utilizando las ya existentes, además de selección y recursión.

Al dejar que el mismo traductor implemente el mejor algoritmo para hacer una determinada tarea hace que los programas funcionales sean más independientes de la arquitectura de la computadora. Veamos un ejemplo de implementación funcional:

Vamos a implementar el algoritmo del MCD (máximo común divisor). Para resolverlo se utiliza el algoritmo de Euclides que consiste en varias divisiones euclidianas sucesivas. En C nuestro algoritmo queda:

```
void mcd(int u, int v, int *x){
      int y, t, z;
      z = u;
      y = v;
      while (y!=0){
            t = y;
            y = z%y;
            z = t;
      }
      *x = z;
} // Fin de mcd
```

La versión funcional (sin asignación y con recursión) de este código es:

```
int mcd(int u, int v){
      if (v==0) return u;
      else return mcd(v, u%v);
}
```

Como se pudo observar las dos versiones son muy diferentes, y aunque hacen lo mismo, implementar el MCD de manera recursiva resulta en un código más compacto y elegante.

4.6 Lenguajes funcionales puros e híbridos

Matemáticamente las variables siempre representan valores reales, pero en los lenguajes de programación imperativos por ejemplo las variables se refieren a localizaciones de memoria, así como a valores. Ya que en matemáticas no existe este concepto de localización en la memoria o los valores de 1 de las variables, el enunciado $x = x + 1$ no tiene sentido. Por eso el paradigma funcional elimina el concepto de variable a excepción del uso como nombre para un valor. Por consecuencia, no hay operaciones de asignación. Solo hay constantes, parámetros y valores.

Si un lenguaje de programación funcional (también llamado recursivo) trabaja de esta forma, sin variables ni operaciones de asignación, se dice que es un lenguaje funcional **puro**. La mayoría de los lenguajes funcionales conservan alguna idea de asignación, lo que los hace **impuros** o **híbridos**, pero permiten trabajar de forma pura si así lo requerimos. Además de la falta de asignaciones, tampoco hay ciclos. La forma de hacer que una operación se repita (a falta de ciclos *while*) es la **recursión**. Algunos de los principales lenguajes de programación funcional son: Hope, LML, Clean, Haskell, FP, Miranda, SML y LISP.

4.7 Introducción a los lenguajes funcionales

LISP: Abreviación de LISt Processor, fue desarrollado en el MIT en los 60. Está basado en el cálculo lambda λ desarrollado por Church. No hay un estándar para LISP, así que nos vamos a encontrar con muchas variantes de este lenguaje. Una de estas variantes es Scheme. Es así como surgen los dialectos. Scheme es un dialecto de LISP.

En Scheme los programas y datos son expresiones y son de dos tipos: átomos y listas. Los átomos son como las constantes e identificadores de un lenguaje imperativo como C: incluyen números, cadenas, nombres, funciones, etc. Una

lista es simplemente una secuencia de expresiones separadas por espacios y rodeadas por paréntesis, por ejemplo: (+ 2 3).

Los programas se ejecutan evaluándolos como expresiones, y éstas a su vez se evalúan aplicando el primer elemento de una lista (que debe ser una función) al resto de los elementos que vienen siendo los argumentos. Ejemplo:

```
(mcd 8 18)
```

La función mcd se aplica a los argumentos 8 y 18.

Así, 2 + 3 se escribe (+ 2 3), en donde la función + se aplica a los argumentos 2 y 3.

El cálculo del MCD en Scheme-LISP quedaría así:

```
(define (mcd u v)
      (if (= v 0) u
            (mcd v(modulo u v))
      )
)
```

En la función `mcd` se utiliza la función `if-then-else`, pero a diferencia de los demás paradigmas, el *if* es una función, no una estructura de control selectiva. `(if a b c)` significa:

```
if a
      then b
      else c
```

Esta función representa tanto el control como el cómputo de un valor. Primero se evalúa *a*, y dependiendo del resultado, se evalúa ya sea *b* o *c*, convirtiéndose el valor resultante en el valor devuelto por la función. El *if* en otros lenguajes carece de valor.

La principal desventaja de los lenguajes funcionales es la ineficiencia en su ejecución. Por ser dinámicos, deben ser interpretados más que compilados, con la consecuente pérdida de velocidad de ejecución.

4.8 Campos de aplicación

Los lenguajes funcionales son más cercanos a la manera en que funciona la mente humana, pues permiten a los programadores describir sus algoritmos como expresiones que serán evaluadas. Este paradigma encuentra diversas aplicaciones en áreas como las bases de datos, ingeniería del software, procesadores de lenguajes, inteligencia artificial, redes neuronales y sistemas expertos. Los lenguajes funcionales permiten la creación de procedimientos en tiempo de ejecución, lo que lleva a un gran nivel de modularidad que difícilmente puede alcanzarse en otros paradigmas de programación.

Bibliografía del tema 4

1. Berlage, Th., *Concepts and paradigms*, Wokingham, Addison Wesley, 1991.
2. Decker Rick y Stuart Hirsfield, *Máquina analítica*, México, Thomson Learning, 2001.
3. Farret, *Introducción a la programación. Lógica y diseño*, 4ª ed., México, Thomson Learning, 2002.
4. Knuth, Donald, *The art of Computer Programming*, 2ª ed., Massachusetts, Addison Wesley. 1997. Vol. 1, *Fundamental Algorithms.*
5. Louden, Kenneth C., *Programming Languages: Principles and Practice,* Second Edition, Brooks Cole, 2003.

6. Pratt, T.W., *Lenguajes de programación*, México, Prentice Hall, 1987.
7. Wilson, L. B., y R. Clark, *Comparative Programming Languages*, EE.UU., Addison Wesley, 1988.

Actividades de aprendizaje

A.4.1. Busca en la Web la historia y evolución de LISP. Haz un árbol genealógico de los dialectos que han surgido a partir de este lenguaje; sus creadores, años de aparición y equipos en los que pueden correr.

A.4.2. Busca en la Web un programa en LISP y otro en Java o C que implemente el algoritmo del movimiento de la Reina en el juego de ajedrez. Compara el rendimiento de cada uno en la ejecución y analiza por qué uno es más lento que los otros, aunque sean menos líneas de código.

A.4.3. Investiga y realiza un mapa conceptual sobre las características de Haskell. Debe incluir las diferencias y similitudes con Scheme.

Cuestionario de autoevaluación

1. ¿Cómo maneja los programas el paradigma funcional?
2. ¿Qué es una función?
3. ¿Qué es una variable independiente?
4. ¿Qué es una variable dependiente?
5. ¿Para qué sirve el cálculo lambdaλ?

Examen de autoevaluación

1. Elije el inciso que conteste correctamente cada pregunta.

1. La *y* del conjunto Y de la ecuación y=f(x) se llama:

a) Cálculo lambda λ
b) Variable dependiente
c) Variable independiente
d) Procedimiento

2. A los lenguajes funcionales también se les llama:

a) Lenguajes lógicos
b) Lenguajes procedurales
c) Lenguajes imperativos
d) Lenguajes recursivos

3. En los lenguajes funcionales, las operaciones repetitivas no se expresan mediante ciclos o lazos, sino mediante:

a) Funciones de control
b) Funciones secuenciales
c) Funciones recursivas
d) Funciones imperativas

4. En el paradigma funcional se eliminan los conceptos de:

a) variable como una localización de memoria y operaciones de asignación
b) variable como nombre para un valor y operaciones de asignación
c) operaciones de repetición y de recursividad
d) operaciones de repetición y asignación

5. El valor devuelto por *(* (+ 50 30) (- 80 50))* es:

a) 1400
b) 2400
c) 3400
d) 4000

6. En LISP, los programas son:

a) Procedimientos anidados
b) Funciones recursivas
c) Expresiones de listas
d) Listas recursivas

II. A partir del contenido de cada columna, elije el inciso de la versión que le corresponda, escribiendo la letra en el espacio en blanco.

1.

procedure mcd(u, v: in integer; x out integer)is
y, t, z: integer;
begin
z:=u;
y:=v;
loop
exit when y=0
t:=y;
y:=z mod y;
z:=t;
end loop;
x:=z;
end mcd;
	(define (mcd u v)
	(if (= v 0) u
	(mcd v (modulo u v))))
		function mcd(u, v: in integer) return integer is
		begin
		if v=0 the
		return u;
		else
		return mcd(v, u mod v);
		end if;
		end mcd;
			Int mcd(int u, int v){
			if (v==0) return u;
			else
			return mcd(v, u % v);
			}

a) Versión funcional recursiva en C del MCD
b) Versión no funcional en ADA del MCD
c) Versión del tipo y = f(x) del MCD en ADA
d) Versión funcional en LISP del MCD

2.

En *y = f(x)* ...	**En un programa ...**	**En una función ...**

a) ***x*** representa los parámetros y ***y*** los valores devueltos
b) ***x*** representa cualquier valor proveniente de ***X*** y ***y*** la variable dependiente
c) ***x*** representa las entradas y ***y*** las salidas

TEMA 5. Programación lógica

Objetivo particular

Al término de este tema, el alumno:

Identificará los conceptos más importantes que dan soporte y fundamento al paradigma de programación lógica, y los aplicará en la resolución de problemas algorítmicos propios de la automatización de la información y manipulación de datos.

Temario detallado

5.1 Definición
5.2 Hechos
5.3 Reglas
5.4 Cláusulas de Horn
5.5 Predicados
5.6 Introducción a los lenguajes lógicos
5.7 Campos de aplicación

Introducción

Si hasta ahora pensabas que programar en ensamblador, C, Java, o incluso Ruby e inclusive en LISP era todo lo que había en el mundo de los lenguajes de programación, pues no es así. Con los lenguajes funcionales se puede programar el movimiento de los discos de la torre de Hanoi sin una estructura *if* o *case*, o sin el *while* o el *for*, o sin declarar una variable ni mucho menos asignarle un valor. Todo se hace a través de funciones de programación matemáticamente declaradas. Pero también podemos prescindir de las funciones, o por lo menos, las podemos combinar con expresiones de "*cierto o*

falso” para programar cualquier cosa, como por ejemplo la siguiente expresión: “un caballo es un mamífero”, la cual es cierta. Parece algo inusual, pero hay un paradigma que en su definición pura no necesita de otra cosa más que de expresiones lógicas como la del ejemplo anterior para realizar programas. Se llama paradigma lógico.

5.1 Definición

La programación lógica aplica el *corpus*[30] de conocimiento sobre lógica para el diseño de lenguajes de programación y es una combinación de programación declarativa y programación funcional. Esto quiere decir que describen relaciones entre las variables en términos de funciones y reglas de inferencia (procedimiento que infiere hechos a partir de otros hechos conocidos), dejando al traductor la responsabilidad de encontrar el mejor algoritmo para encontrar el resultado buscado.

Los programas lógicos están construidos únicamente por expresiones (enunciados) lógicas las cuales o son verdaderas o son falsas, y algoritmos específicos para implementar las reglas de inferencia. No hay propiamente expresiones interrogativas o imperativas como en los otros paradigmas.

5.2 Hechos

La programación lógica trabaja con expresiones y sus relaciones que se especifican a través de reglas:

‘Para cumplirse el hecho A debe cumplirse B_1 y B_2 y B_3 y…B_n’

Estas reglas se llaman cláusulas de Horn y constituyen un subconjunto de los predicados de primer orden de la lógica matemática. Un hecho *P*

[30] “Conjunto lo más extenso y ordenado posible de datos o textos científicos, literarios, etc., que pueden servir de base a una investigación”. Fuente: Diccionario de la Lengua Española en línea, Vigésima segunda edición. http://www.rae.es

es un caso especial de regla para el cual no hay condiciones B_i que deban cumplirse, así se escribe como *P.*[31]

Hay dos partes diferenciadas en la programación lógica:

1) La base de hechos y las relaciones entre estos, que se conoce como **base de conocimiento**, cuya relación está a cargo del programador, y
2) El **motor de inferencia**, que es mecanismo de exploración de la base de hechos y relaciones para elaborar conclusiones.[32]

Veamos el siguiente ejemplo en Prolog (lenguaje lógico) de una base de conocimiento (los comentarios se ponen entre llaves):

```
Matrimonio (María, Juan).  {Hecho}
Matrimonio (Lupita, José). ).  {Hecho}
Matrimonio (Carmen, Paco) ). {Hecho}.
Matrimonio (Elena, Federico) ).  {Hecho}.
Padres (Carmen, Lupita, José) ). {Hecho}.
Padres (Paco, María, Juan) ).  {Hecho}.
Padres (Elena, María, Juan) ). {Hecho}.
Suegros (L) :- Matrimonio (L, X), Padres (X, H, I). {Regla}
```

Este programa funciona de la siguiente manera: el intérprete pregunta a Prolog y éste responde usando el motor de inferencia. El signo **?-** indica que Prolog está listo para procesar una pregunta tras haber leído la base de conocimiento:

---

[31] Francisco Martínez, Francisco A. Martínez Gil, Gregorio Martín Quetglás: *Introducción a la programación estructurada*, Universidad de Valencia, 2003, p. 52. También disponible en formato digital, en: http://books.google.es/books?id=-cVzTPOWf3kC.

[32] Francisco Martínez, Francisco A. Martínez Gil, Gregorio Martín Quetglás: *Introducción a la programación estructurada*, Universidad de Valencia, 2003, p. 52. También disponible en formato digital, en: http://books.google.es/books?id=-cVzTPOWf3kC.

```
?- Suegros (Carmen).
X = Paco
H = María
I = Juan
no.
```

Prolog intenta encontrar en los hechos las variables que satisfagan la regla, esto es, que hagan la regla verdadera. Se van probando en el orden en que están escritos en la base de conocimiento. Explorará todas las combinaciones en busca de soluciones. La palabra *no* del final indica que no encontró más soluciones y despliega en pantalla todas las variables que verifican la regla.

5.3 Reglas de inferencia

Para explicar qué son las reglas de inferencia, debemos saber que los enunciados lógicos de un programa lógico están compuestos de los siguientes elementos o tokens:

- **Constantes**: son por lo general números o nombres. También se les llama átomos, ya que no se pueden dividir en partes más pequeñas. 1 es un ejemplo de constante.
- **Predicados**: son los nombres de funciones que son verdaderas o falsas. Pueden tomar varios argumentos. La función natural(2) es un ejemplo de predicado.
- **Funciones**: se distinguen las funciones que son verdaderas o falsas (predicados) del resto de las funciones. La función sucesor(x) es un ejemplo de función.
- **Variables**: representan cantidades todavía no especificadas. Ejemplo: x es una variable.

- **Conectores**: Incluyen las operaciones *y*, *o* y *no*, además de las operaciones sobre datos booleanos. Otros conectores son la implicación → y la equivalencia ↔.
- **Cuantificadores**: son operaciones que introducen variables. Existen cuantificadores universales y existenciales.
- **Símbolos de puntuación**: incluyen paréntesis izquierdo y derecho, la coma y el punto. Los paréntesis sirven para encerrar argumentos y para agrupar operaciones.

Las reglas de inferencia son las formas de derivar o de probar nuevos enunciados a partir de un conjunto dado de enunciados. Estas reglas nos permiten construir el conjunto de todos los enunciados que pueden derivarse o comprobarse a partir de un conjunto dado de enunciados. Se trata de enunciados que son verdaderos siempre que los enunciados originales sean verdaderos. Ejemplo:

Los siguientes enunciados son lógicos:

```
Un caballo es un mamífero.
Un ser humano es mamífero.
Los mamíferos tienen cuatro patas y ningún brazo, o dos pies y
dos brazos.
Un caballo no tiene brazos.
Un ser humano tiene brazos.
Un ser humano no tiene patas.
```

Los enunciados derivados de los anteriores pueden ser:

```
patas(caballo,4).
brazos(caballo,0).
pies(caballo,0).
```

```
patas(serHumano,0).
brazos(serHumano,2).
pies(serHumano,2).
```

Los primeros enunciados lógicos se le llama **axiomas**, y los enunciados derivados se les llama **teoremas**.

5.4 Cláusulas de Horn[33]

Una cláusula de Horn es un enunciado

a_1 y a_2 y a_3 … y a_n → b

donde a los a_1 solo se les permite ser enunciados simples sin conectores. No existen conectores "o" ni cuantificadores. En el enunciado se dice que a_1 hasta a_n; esto implica que *b* es verdadero si todos los a_1 son verdaderos. A *b* se le llama cabeza de la cláusula, y a $a_1 \ldots, a_n$ el cuerpo de la cláusula. Se utilizan para expresar la mayoría, pero no la totalidad, de los enunciados lógicos. Ejemplo:

Veamos cómo queda la descripción lógica para el algoritmo del MCD entre dos números enteros positivos *u* y *v*:

```
El mcd de u y de 0 es u.
El mcd de u y de v, si v no es 0, es el mismo que el mcd de v y
del residuo de dividir u entre v.
```

Pasando lo anterior a predicados nos queda:

---

[33] Ver, Kenneth C. Louden, *Programming Languages: Principles and Practice*, Second Edition, Brooks Cole, 2003, p. 498.

Para toda u, mcd(u,0,u).
Para toda u, para todas v, para todas w, no cero(v) y mcd(v,u mod v,w) → mcd(u,v,w).

Para traducir estos enunciados a cláusulas Horn, solo hay que eliminar los cuantificadores:

mcd(u,0,u).
no cero(v) y mcd(v,u, mod v,w) → mcd(u,v,w).

5.5 Predicados[34]

Como ya se mencionó en el tema de reglas de inferencia, los predicados son los nombres de funciones que son verdaderas o falsas. Pueden tomar varios argumentos.

En el cálculo de predicados, los argumentos a los predicados y funciones deben ser combinaciones de variables, contantes y funciones (a éstos se les llama **términos**). Los términos no pueden tener predicados, cuantificadores o conectores. Las funciones de la programación funcional no son otra cosa más que la evolución de los predicados de la programación lógica.

5.6 Introducción a los lenguajes lógicos

La mayoría de los lenguajes de programación lógica están basados en la lógica de primer orden o cálculo de predicados de primer orden. Esta lógica sirve para cuantificar variables individuales, por ejemplo, "*las plantas son seres vivos*", en donde "*plantas*" está cuantificando universalmente a todas las plantas.

[34] loc. cit.

El lenguaje de programación lógica por excelencia es **Prolog**, cuyo nombre viene del francés *Programation et Lógique*, y el cual es un lenguaje de programación interpretado y cuenta con muchas variantes. Desde el punto de vista del programador, la ventaja principal de Prolog es su facilidad para escribir programas claramente legibles de forma rápida y con pocos errores.

5.7 Campos de aplicación

La programación lógica tiene su hábitat natural en el desarrollo de aplicaciones de inteligencia artificial, como los sistemas expertos, la demostración automática de teoremas, reconocimiento del lenguaje natural, y en menor medida en aplicaciones de propósito general.

Un campo muy socorrido para la programación lógica son los diccionarios de palabras de los celulares, los cuales incorporan un diccionario y un algoritmo que va restringiendo las posibles palabras que puedan ser resultado de la presión de una serie de teclas. Para ello se definen dos predicados que brindan la información necesaria del teléfono celular:

1. Predicado que define el mapeo entre las teclas de celular y los caracteres:

```
teclado([ (1, [1]), (2, [2,a,b,c]), (3, [3,d,e,f]),
(4, [4,g,h,i]), (5, [5,j,k,l]), (6, [6,m,o,n]),
(7, [7,p,q,r,s]), (8, [8,t,u,v]), (9, [9,w,x,y,z]),
(0, [0]), (*, [-]), (#, [#])
]
```

2. Predicado que define un diccionario de palabras:

```
diccionario([ [1,a], [l,a], [c,a,s,a], [a], [d,e], [r,e,j,a],
[t,i,e,n,e],
[c,a,s,a,m,i,e,n,t,o], [d,e,l], [a,n,t,e,s]
])
```

Bibliografía del tema 5

1. Berlage, Thomas, *Concepts and paradigms*, Wokingham, Addison Wesley, 1991.
2. Bramer, Max: *Logic Programming with Prolog*, Springer; 2005.
3. Decker Rick y Stuart Hirsfield, *Máquina analítica*, México, Thomson Learning, 2001.
4. Farrell, Joyce: *Programming Logic and Design, Comprehensive*, 5ª ed., Course Technology, 2008.
5. Farret, *Introducción a la programación. Lógica y diseño,* 4ª ed., México, Thomson Learning, 2002.
6. Knuth, Donald, *The art of Computer Programming*, 2ª ed., Vol. 1, Fundamental Algorithms, EEUU, Addison Wesley, 1977.
7. Louden, Kenneth C., *Programming Languages: Principles and Practice*, Second Edition, Brooks Cole, 2003.
8. Pratt, T.W., *Lenguajes de programación,* México, Prentice-Hall, 1987.
9. Sebesta, Robert, *Concepts of Programming Languages,* 3ª ed., EEUU, Addison Wesley, 1986.
10. Sethi, Ravi, *Lenguajes de programación,* México, Addison Wesley, 1992.
11. Wilson, L. B. y R. Clark, *Comparative Programming Lenguajes*, EEUU, Addison Wesley, 1988.
12. Watson, Des, *High Level Languages and their Compilers,* EEUU, Addison Wesley, 1992.

ACTIVIDADES DE APRENDIZAJE

A.5.1 Busca en la Web y descarga algunos de los siguientes ambientes de programación en Prolog[35]. Posteriormente implementa algunos de los algoritmos que se listan más adelante.

- **ADA PD Prolog** es un intérprete antiguo (1986) y lento para MS-DOS de *Automata Design Associates*.
- **Arity/Prolog32** es un completo entorno de programación en Prolog para Windows de 32 bits que incluye un verdadero compilador, editor, depurador, intérprete y ayuda. Para descargarlo hay que rellenar un formulario.
- **B-Prolog** es un completo sistema CLP (*Constraint Logic Programming*) que ejecuta programas Prolog y CLP(FD). Hay versiones para Windows y varios Unix, incluyendo Linux. Se distribuye también el código fuente.
- **Ciao** es un entorno de programación multi-paradigma que ofrece un completo sistema Prolog de acuerdo con el estándar ISO, permitiendo tanto restricciones como extensiones al lenguaje. Se distribuye el código fuente y binarios para Windows. Aunque las páginas están en inglés, los autores pertenecen a un grupo de investigación de la Universidad Politécnica de Madrid.
- **CU-Prolog** es un lenguaje CLP experimental adecuado para el procesamiento de lenguajes naturales. Hay versiones Unix, MS-DOS (djcup) y Macintosh (MacCup).
- **DGKS** es un intérprete de Prolog escrito en Java. Se puede descargar y ejecutar localmente o directamente a través de la página web.
- **ESL Prolog-2** (PD Version) es una versión limitada pero de buen rendimiento de un intérprete Prolog para MS-DOS (de 1991).

[35] Lista extraída de http://www.rodoval.com/paginalen.php?len=Prolog (04/03/09)

- **GNU Prolog** es un compilador conforme al estándar ISO con algunas extensiones. Incluye también un intérprete interactivo con depurador. Ha sido portado a muchas versiones de Unix (incluyendo Linux) y a Windows (con Cygwin o MSVC++). Se distribuyen ejecutables para Linux y Windows y el código fuente.
- **JIProlog** es un intérprete de Prolog compatible con Java que permite añadir la potencia de Prolog a cualquier aplicación o applet Java.
- **jProlog** es un intérprete Prolog escrito en Java.
- **Kernel** Prolog es un intérprete Prolog escrito en Java con un sistema innovador de built-ins basado en la extensión Fluents.
- **K-Prolog** es un compilador de Prolog para Windows, Linux y otros Unix.
- **LPA PROLOG** Professional es un compilador antiguo de Prolog para MS-DOS de 16 bits, ofrecido ahora gratuitamente para uso personal.
- **NU-Prolog** se distribuye sólo para la enseñanza y la investigación. Sólo código fuente.
- **Open Prolog** es una implementación de Prolog para Macintosh (MacOS 7.5.5 y posteriores). Es postcardware, o sea, sus usuarios deben enviarle al autor (Mike Brady) una postal.
- **Qu-Prolog** (de la Universidad de Queensland) es un Prolog extendido diseñado principalmente como lenguaje de prototipos y como lenguaje táctico para la demostración interactiva de teoremas. Se distribuye gratuitamente sólo para uso no comercial. Sólo para Unix y Linux.
- **Reform Prolog** es un proyecto abandonado sin terminar, pero se distribuye una versión beta (código fuente).
- **Strawberry Prolog** es un compilador para Windows y Unix/Linux. Sólo la Light Edition es gratuita.
- **SWI-Prolog** es un compilador dirigido principalmente a la investigación y la educación. Se distribuyen fuentes y binarios para Linux, Windows y MacOS X (Darwin).

- **Visual Prolog** es un entorno de programación en Prolog para Windows sucesor de Turbo Prolog y PDC Prolog. La Personal Edition es gratuita, pero sólo debe usarse para su aprendizaje y no se debe distribuir los ejecutables generados, que mostrarán al principio un letrero.
- **W-Prolog** es un intérprete de un lenguaje tipo Prolog implementado en Java. Es muy portable y puede ejecutarse como aplicación o como applet.
- **XSB** es una extensión de Prolog para incluir una implementación eficiente de memorización y una implementación inicial de HiLog. Se puede descargar el código fuente y ejecutables para Windows.
- **YAP** es un compilador de Prolog de alto rendimiento desarrollado en la Universidad de Oporto. Se distribuyen las fuentes y binarios para Linux, SPARC/Solaris y Windows.

A.5.2. Implementa alguno de los siguientes algoritmos en Prolog:

- Ordenamiento burbuja (bubble sort)
- Ordenamiento por inserción (insert sort)
- Ordenamiento de mezcla (merge sort)
- Ordenamiento rápido (quick sort)
- Ordenamiento ingenuo (naive sort)

CUESTIONARIO DE AUTOEVALUACIÓN

1. ¿De qué están constituidos los programas lógicos?
2. ¿Qué es una base de conocimiento?
3. ¿Qué es un motor de inferencia?
4. ¿Qué son las reglas de inferencia?
5. ¿Qué es una cláusula de Horn?
6. ¿Qué es un predicado?

Examen de autoevaluación

1. Elije el inciso que conteste correctamente cada pregunta

1. Paradigma que aplica el *corpus* de conocimiento sobre lógica para el diseño de lenguajes de programación y es una combinación de programación declarativa y programación funcional:

a) Paradigma funcional
b) Paradigma imperativo
c) Paradigma lógico
d) Paradigma orientado a objetos

2. Son formas de derivar o de probar nuevos enunciados a partir de un conjunto dado de enunciados:

a) Expresiones lógicas
b) Reglas de inferencia
c) Predicados
d) Hechos

3. Es el mecanismo de exploración de la base de hechos y relaciones para elaborar conclusiones:

a) Motor de inferencia
b) Cuantificadores
c) Conectores
d) Predicados

4. La base de hechos y las relaciones entre éstos se conoce como:

a) Reglas de inferencia
b) Motor de inferencia
c) Base de conocimiento
d) Cuantificadores

5. Son por lo general números o nombres. También se les llama átomos, ya que no se pueden dividir en partes más pequeñas:

a) Predicados
b) Reglas de Horn
c) Conectores
d) Constantes

6. Son los nombres de funciones que son verdaderas o falsas. Pueden tomar varios argumentos:

a) Predicados
b) Conectores
c) Cláusulas de Horn
d) Variables

7. Son operaciones que introducen variables. Existen universales y existenciales:

a) Predicados
b) Cuantificadores
c) Constantes
d) Conectores

8. Incluyen paréntesis izquierdo y derecho, la coma y el punto. Los paréntesis sirven para encerrar argumentos y para agrupar operaciones:

a) Reglas de inferencia
b) Constantes
c) Símbolos de puntuación
d) Funciones

9. Incluyen las operaciones *y*, *o* y *no*, además de las operaciones sobre datos booleanos. Otros son la implicación → y la equivalencia ↔:

a) Conectores
b) Símbolos de puntuación
c) Cláusulas de Horn
d) Reglas de inferencia

10. Representan cantidades todavía no especificadas:

a) Constantes
b) Variables
c) Funciones
d) Reglas de Horn

Bibliografía básica

Alfonseca, Manuel, et. al., *Compiladores e intérpretes: teoría y práctica*, Madrid, Pearson Educación, 2006.

Berlage, Thomas, *Concepts and paradigms*, Wokingham, Addison Wesley, 1991.

Bramer, Max: *Logic Programming with Prolog*, Springer; 2005.

Bronson, Gary J., *C++ para Ingeniería y ciencias*, México, Thomson Learning, 1999.

Ceballos, Francisco Javier, *Microsoft Visual C++6 aplicaciones para Win32*, Madrid, Alfaomega/Rama, 2002.

Decker Rick y Stuart Hirsfield, *Máquina analítica*, México, Thomson Learning, 2001.

Farrell, Joyce: *Programming Logic and Design, Comprehensive*, 5ª ed., Course Technology, 2008.

Farret, *Introducción a la programación. Lógica y diseño*, 4ª ed., México, Thomson Learning, 2002.

Graham, Ian, *Métodos orientados a objetos*, México, Addison Wesley, 1995.

Knuth, Donald, *The art of Computer Programming*, 2ª ed., Reading Mass., Addison Wesley, 1977, vol. 1, Fundamental Algorithms

López Román, Leobardo, Programación *estructurada. Un enfoque algorítmico*. México, Alfaomega, 2002.

_____, *Programación estructurada en TurboPascal 7*, México, Alfaomega, 2002.

Louden, Kenneth C., *Programming Languages: Principles and Practice,* 2ª ed., Brooks Cole/Course Technology, 2003.

Orós, Juan Carlos, *Diseño de páginas Web interactivas con JavaScript*, 2ª ed., Madrid, Alfaomega/Rama, 2001.

Pratt, T.W., *Lenguajes de programación*, México, Prentice-Hall, 1987.

Sebesta, Robert, *Concepts of Programming Languages*, 3ª ed., EE.UU. Addison Wesley, 1986.

Sethi, Ravi, *Lenguajes de programación*, México, Addison-Wesley, 1992.

Smith, Jo Ann, *C++ Programación Orientada a objetos*, México, Thomson Learning, 2002.

Wang, Paul, *Java con programación orientada a objetos y aplicaciones en la WWW*, México, Thomson Learning, 2002.

Wilson, L. B., y R. Clark, *Comparative Programming Languages*, EE.UU., Addison Wesley, 1988.

Watson, Des, *High Level Languages and their Compilers,* EEUU, Addison-Wesley, 1992.

RESPUESTAS A LOS EXÁMENES DE AUTOEVALUACIÓN

PROGRAMACIÓN

	Tema 1	Tema 2	Tema 3	Tema 4		Tema 5
				I	II	
1.	c	b	c	b	b, d, c, a	c
2.	c	b	a	d	b, c, a	b
3.	d	d	d	c		a
4.	d	c	b	a		c
5.	a	d	b	b		d
6.	c		c	c		a
7.	b		c			b
8.	c		b			c
9.	c		d			a
10.	d		d			b