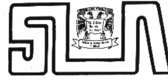


LENGUAJES DE PROGRAMACIÓN

1. Plataforma Teórico Conceptual.

Introducción.

Lenguajes de programación hay en gran cantidad, algunos han evolucionado a lo largo del tiempo y siguen vigentes en el transcurso de muchos años, mientras que otros han sido operativos durante un período más o menos largo y actualmente no se usan.
Dada esta gran variedad de lenguajes, no se pretende dar una visión de todos, sino una clasificación en diversos tipos y concretar algunos de ellos. En general un lenguaje es un método conveniente y sencillo de describir las estructuras de información y las secuencias de acciones necesarias para ejecutar una tarea concreta.

Los lenguajes de programación utilizan juegos de caracteres "alfabeto" para comunicarse con las computadoras. Las primeras computadoras sólo utilizaban informaciones numéricas digitales mediante el código o alfabeto digital, y los primeros programas se escribieron en ese tipo de código, denominado código máquina basado en dos dígitos 0 y 1, por ser entendible directamente por la máquina (computadora). La enojosa tarea de programar en código máquina hizo que el alfabeto evolucionase y los lenguajes de programación comenzaran a utilizar códigos o juegos de caracteres similares al utilizado en los lenguajes humanos. Así, hoy día la mayoría de las computadoras trabajan con diferentes tipos de juegos de caracteres de los que se destacan el código ASCII y el EBCDIC.
De este modo, una computadora a través de los diferentes lenguajes de programación utilizan un juego o código de caracteres que serán fácilmente interpretados por la computadora y que pueden ser programados por el usuario.
Dos son los códigos más utilizados actualmente en computadoras, ASCII (American Standard Code for Information Interchan-c) y EBCDIC (Extended Binary Coded Decimal Intercha-@ige Code).
El código ASCII básico utiliza 7 bits (dígitos binarios. 0,1) para cada carácter a representar, lo que supone un total de 27 (128) caracteres distintos. El código ASCII ampliado utiliza 8 bits y en ese caso consta de 256 caracteres. Este código ASCII ha adquirido una gran popularidad. ya que es el estándar en todas las familias de computadoras personales.
El código EBCDIC utiliza 8 bits por carácter v por consiguiente consta de 256 caracteres distintos. Su notoriedad reside en ser el utilizado por la firma. En general un carácter ocupara un byte de almacenamiento de memoria.

Al igual que los lenguajes humanos, tales como el inglés o el español, los lenguajes de programación poseen una estructura (gramática o sintaxis) y un significado (semántica). La gramática española trata de los diferentes, modos (reglas) en que pueden ser combinados los diferentes tipos de palabras para formar sentencias o frases aceptables en español. Los lenguajes de computadoras tienen menos combinaciones aceptables que los lenguajes naturales, sin embargo, estas combinaciones deben ser utilizadas correctamente; ello contrasta con los lenguajes naturales que se pueden utilizar aunque no sigan reglas gramaticales e incluso aunque no sean comprendidos.

Finalmente, un lenguaje de programación lo definiremos, como un conjunto de reglas, símbolos y palabras especiales que permiten construir un programa.

1.1. Lenguajes Naturales y lenguajes de programación.

Lengujes naturales

Los lenguajes naturales son aquellos con los cuales hablamos y escribimos en nuestras actividades cotidianas. Entre sus ventajas podemos citar que es fácil de usar y de entender: no debemos aprendernos ningún lenguaje nuevo y cualquiera puede leer la especificación y comentarla o criticarla. Entre los inconvenientes están la imprecisión y la ambigüedad. Aunque el análisis de requisitos se haya realizado correctamente, una especificación en lenguaje natural puede dar lugar a que la implementación final no cumpla estos requisitos. Además, debido a su propia facilidad de uso e imprecisión, las especificaciones suelen ocultar lagunas que sólo se pondrán de manifiesto a la hora de programar, es decir, al traducir la especificación a un lenguaje de programación. El uso de subconjuntos del lenguaje, como el llamado "inglés estructurado", atenúa estas deficiencias pero sigue sin resolver problemas como la corrección, consistencia o completitud de la propia especificación o de los programas desarrollados a partir de ella.

Los lenguajes de programación ocupan una posición intermedia entre los lenguajes naturales humanos y los precisos lenguajes de máquina.

Gracias a la progresiva complejidad de los lenguajes traductores que permiten convertir las instrucciones de un lenguaje de programación al lenguaje de máquina, la programación puede usar lenguajes de computación que se parecen cada vez más a los lenguajes naturales.

También se habla de lenguajes naturales para referirse al software que permite que los sistemas de computación acepten, interpreten y ejecuten instrucciones en el lenguaje materno o "natural" que habla el usuario final, por lo general el inglés. La mayor parte de los lenguajes naturales comerciales están diseñados para ofrecer a los usuarios un medio de comunicarse con una base de datos corporativa o con un sistema experto.

Podemos resumir que los lenguajes naturales se distinguen por las siguientes propiedades:

- Desarrollados por enriquecimiento progresivo antes de cualquier intento de formación de una teoría.
- La importancia de su carácter expresivo debido grandemente a la riqueza de el componente semántico (polisemántica).
- Dificultad o imposibilidad de una formalización completa.

Lenguajes de programación

Un lenguaje de programación consiste en un conjunto de órdenes o comandos que describen el proceso deseado. Cada lenguaje tiene sus instrucciones y enunciados verbales propios, que se combinan para formar los programas de cómputo.

Los lenguajes de programación no son aplicaciones, sino herramientas que permiten construir y adecuar aplicaciones.

Existen muchos lenguajes de programación con características y aptitudes muy diferenciadas. Todo ello se encuentra en dos grandes grupos:

- Los lenguajes máquina.
- Los lenguajes simbólicos. Lenguaje de programación en el que las instrucciones de los diferentes programas se codifican utilizando los caracteres de las lenguas naturales. La ejecución de un programa.

Entre los primeros se encuentran los denominados lenguajes en código máquina. En estos lenguajes, la codificación de estos lenguajes se hace utilizando un lenguaje binario de ceros y unos que son los únicos símbolos que puede entender cualquier computador. Cada sistema físico tiene su código máquina distinta por lo que un programa escrito en un determinado código máquina sólo vale para un sistema físico.

A los lenguajes máquina les sucedieron, los lenguajes simbólicos los cuales utilizan caracteres naturales para escribir las instrucciones de los programas. Los lenguajes simbólicos se dividen a su vez en:

- Lenguajes simbólicos de bajo nivel o ensambladores.
- Lenguajes simbólicos de alto nivel.

Dentro de los segundos se puede distinguir a su vez los lenguajes procedurales y los relacionales.

1. Un *lenguaje procedural* es aquel lenguaje de programación en el que hay que señalar tanto lo que se quiere hacer como el modo de hacerlo. Los lenguajes de tercera generación son de tipo procedural.
2. Un *lenguaje relacional* es un tipo de lenguaje de programación en el que sólo hay que especificar lo que se quiere obtener, sin necesidad de especificar a su vez el camino a seguir para obtener los resultados deseados. Este tipo de lenguaje son de muy alta productividad en desarrollo pero muy ineficientes en ejecución.

La diferencia entre uno y otro es que los primeros exige que se diga tanto lo que se quiere hacer como la forma en que hay que hacerlo mientras que los relacionales sólo exigen que se diga lo que se quiere hacer, pero no es necesario que se exprese el camino para realizarlo.

A medida que se va subiendo de nivel los lenguajes son más sencillos y más productivos en desarrollo, pero en contra partida son menos eficientes a la hora de su ejecución.

Los programas escritos en lenguajes simbólicos se denominan programas fuente y los programas no son directamente ejecutables su ejecución implica su previa traducción a código máquina para obtener los denominados programas objeto o absolutos. Esta traducción se hace a través de los ensambladores, compiladores o intérpretes, a los lenguajes que de forma genérica se les denomina procesadores de lenguajes.

A los lenguajes máquina se les conoce como lenguajes de primera generación. Los ensambladores son los lenguajes de segunda generación. Los simbólicos de alto nivel de tipo procedural se les denomina de tercera generación y a los relacionales se considera que son de cuarta generación.

A la tercera generación pertenecen lenguajes conocidos como el FORTRAN, COBOL, RPG, BASIC, PL1, SIMULA, ALGOL, PASCAL, ADA, C, LISP, PROGOL, etc. Estos lenguajes de tercera generación son de tipo universal.

A diferencia de los lenguajes de tercera generación, los de cuarta generación no son tan universales y van asociados a determinados sistemas operativos y en muchos casos a determinados sistemas de almacenamiento de información, lo que les resta uno de los mayores intereses de los lenguajes de tercera generación.. La mayor ventaja de los lenguajes relacionados es que son muy productivos en desarrollo a alcanzar niveles de productividad de hasta 6 y 8 veces superior a los que alcanzan los de tercera generación.

Entre los lenguajes de cuarta generación tenemos el CSP de IBM asociado al sistema de gestión de base de datos DB2; el NATURAL de SOFTWARE AG asociado al sistema de datos ADABAS; el SQL que es un lenguaje convertido en estándar mundial como lenguaje de cuarta generación para la consulta de bases relacionales; etc.

Además de los lenguajes señalados hay que hacer referencia a una serie de lenguajes orientados a objetos y cuya utilización tiene preferentemente lugar en los puestos cliente. Entre estos destaca el Visual Basic de Microsoft, el Power Builder y el Delphi.

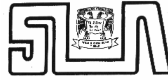
Además hay que prestar una seria atención al lenguaje de programación Java, desarrollado por SUN y cuya misión fundamental es dar la posibilidad de desarrollar aplicaciones altamente interactivas bajo la modalidad de Web, en el contexto de Internet.

En pocos años y como consecuencia del desarrollo que está teniendo la inteligencia artificial se podrá programar, con ciertas limitaciones sintácticas, en los lenguajes naturales

1.2. Paradigmas en lenguajes de programación.

Existen diversos lenguajes y paradigmas de programación que se han diseñado para facilitar la tarea de la programación en diferentes ámbitos. Por ejemplo, la programación orientada a objetos es un paradigma dirigido al mejoramiento en la calidad del software por medio de la observación de aspectos tales como la corrección, robustez, extensibilidad, compatibilidad y sobre todo la reusabilidad del software. La programación lógica, por su parte, es un paradigma orientado a la expresión de los problemas en términos lógicos para su posterior solución por métodos de inferencia y otras técnicas lógicas.

En la práctica, cada paradigma de programación es implementado a través de diversos lenguajes. Sólo como un ejemplo, la programación orientada a objetos encuentra su recipiente en lenguajes tales como Java, C++, Eiffel, Objective C, el paquete CLOS de Common Lisp, etc.



Existen cuatro modelos básicos de computación que describen casi todos los lenguajes de programación actuales: el imperativo, el aplicativo, el lenguaje con base en reglas y el orientado a objetos. Se describe en forma breve cada uno de estos modelos.

Lenguajes imperativos. Los *lenguajes imperativos o de procedimiento* son lenguajes controlados por mandatos u orientados a enunciados (instrucciones). Un programa se compone de una serie de enunciados, y la ejecución de cada enunciado hace que el intérprete cambie el valor de una localidad o más en su memoria, es decir, que pase a un nuevo estado.

El desarrollo de programas consiste en construir los estados de máquina sucesivos que se necesitan para llegar a la solución. Ésta suele ser la primera imagen, que se tiene de la programación, y muchos lenguajes de uso amplio (por ejemplo, C, C++, FORTRAN, ALGOL, PL/I, Pascal, Ada, Smalltalk, COBOL) manejan este modelo.

Lenguajes aplicativos. Un punto de vista alternativo de la computación representado por un lenguaje de programación consiste en examinar la función que el programa representa y no sólo los cambios de estado conforme el programa se ejecuta, enunciado por enunciado. Esto se puede conseguir observando el resultado deseado en vez de los datos disponibles. En otras palabras, en vez de examinar la serie de estados a través de los cuales debe pasar la máquina para obtener una respuesta, la pregunta que se debe formular es: ¿Cuál es la función que se debe aplicar al estado de máquina inicial accediendo al conjunto inicial de variables y combinándolas en formas específicas para obtener una respuesta? Los lenguajes que hacen énfasis en este punto de vista se conocen como *lenguajes aplicativos o funcionales.*

Lenguajes base en reglas. Los *lenguajes con base en reglas* se ejecutan verificando la presencia de una cierta condición habilitadora y, cuando se satisface, ejecutan una acción apropiada. El lenguaje más común con base en reglas es Prolog, que también se conoce como *de programación lógico,* puesto que las condiciones habilitadoras básicas son ciertas clases de expresiones lógicas de predicados. La ejecución de un lenguaje reglas es similar a la de un lenguaje imperativo, excepto que los enunciados no secuenciales.

Programación orientada a objetos. En este tipo de lenguaje, se construyen objetos complejos de datos y luego designa un conjunto limitado de funciones para que operen con esos datos. Los objetos complejos se designan como extensiones de objetos más simples y heredan propiedades del objeto más sencillo. Al construir objetos a concretos de datos, un programa orientado a objetos gana la eficiencia de los lenguajes imperativos, y al construir clases de funciones que utilizan un conjunto restringido de objetos de datos, se construye la flexibilidad y confiabilidad del modelo aplicativo.

1.3. Razones de estudio de lenguajes de programación.

Cualquier notación para la descripción de algoritmos y estructuras de datos puede llamarse lenguaje de programación; sin embargo nosotros requerimos además que un lenguaje de programación sea implementado (implantado) en una computadora. Cientos de lenguajes de programación se han diseñado e implementado. Ya en 1969, Samment enumeró 120 que han sido usados ampliamente y muchos otros se han desarrollados desde entonces . Sin embargo la mayoría de los programadores

nunca se aventuran a usar más de unos cuantos lenguajes y muchos limitan su programación a uno o dos.

A continuación se describen seis razones primordiales para el estudio de los lenguajes de programación:

1. *Mejorar la habilidad para desarrollar algoritmos eficaces.* Muchos lenguajes tienen ciertas caracteństicas que , usadas adecuadamente , benefician al programador pero cuando se usan en forma inadecuada pueden desperdiciar grandes cantidades de tiempo de computadora o de conducir al programador a errores lógicos que hacen perder mucho tiempo , además, el costo de la reclusión varía según la implementación del lenguaje.
2. *Mejorar el uso del lenguaje de programación disponible.* A través de entendimiento de cómo se implementan las características del lenguaje que uno usa, se mejora grandemente la habilidad para escribir programas más eficientes.
3. *Enriquece su vocabulario de construcciones útiles de programación.* Con frecuencia se nota que los lenguajes sirven tanto para una ayuda como para pensar como para construir, los lenguajes sirven también para estructurar lo que uno piensa, hasta el punto que es difícil pensar en alguna forma que no permita la expresión directa con palabras. El entendimiento de las técnicas de implementación es particularmente, por que para emplear un constructor mientras se programa en un lenguaje que no proporciona directamente el programador debe dar su propia implementación del nuevo constructor en términos de los elementos primitivos ofrecidos realmente por el lenguaje.
4. *Permite una mejor selección de lenguaje de programación.* Cuando la situación lo amerita, el conocimiento de una variedad de lenguajes permite la selección de lenguaje correcto para un proyecto particular por tanto, reduce enormemente el esfuerzo de codificación requerido.
5. *Hace más fácil el aprendizaje de un nuevo lenguaje.* Un lingüista, a través de un conocimiento de las estructuras en que se basan los lenguajes naturaleza, puede aprender un lenguaje extranjero más rápido y fácil que el esforzado principiante que entiende poco de su estructura lengua natal
6. *Facilita el diseño de un nuevo lenguaje.* Pocos programadores piensan en sí mismos como diseñadores; es más ningún programa tiene una interfaces del usuario que es, en realidad, una forma de lenguaje. La interfase del usuario consiste en unos formatos y comandos que son proporcionados por el para comunicarse con el programa. El diseñador de la interfase del usuario de un programa tal como un editor de textos, un sistema, operativo o un paquete de gráficas debe estar familiarizado con mucho de los resultados que están presentes en el diseño de un lenguaje de programación de propósitos generales.

1.4. Evolución de los lenguajes de programación.

Los diseños de lenguaje y los métodos de implementación han evolucionado de manera continua desde que aparecieron los primeros lenguajes de alto nivel en la década de 1950.

Los lenguajes principales FORTRAN; LISP y COBOL fueron diseñados originalmente en los años cincuenta, PL/I, SNOBOLA Y APL se empezaron a usar en los años 60, Pascal, Prolog, Ada, C y Smalltalk son diseños que datan de los años 70, y C++, ML datan de los años ochenta. En las décadas de 1960 y 1970, se solían desarrollar nuevos lenguajes como parte de proyectos importantes de desarrollo de software.

Los lenguajes más antiguos han experimentado revisiones periódicas para reflejar la influencia de otras áreas de la computación; los más nuevos reflejan una composición de experiencias adquiridas en el diseño y de usos de estos y cientos de otros lenguajes más antiguos.

Algunas de las principales influencias en la evolución de diseños de lenguajes se listan a continuación:

1. *Capacidades de las computadoras.* Las computadoras han evolucionado de las máquinas pequeñas lentas y costosas máquinas de tubos de vacío de los años 50 a las supercomputadoras y microcomputadoras de hoy .

2. *Aplicaciones.* El uso de la computadora se ha difundido rápidamente, de la concentración original de aplicaciones militares críticas, científicas, negocios industriales de los años 50 donde el costo podría estar justificado en los juegos en computadora, en computadoras personales y aplicaciones en casi todas las áreas de la actividad humana de hoy.

3. *Métodos de programación.* Los diseños de lenguajes han evolucionado para reflejar, nuestra cambiante comprensión de los buenos métodos para escribir programas largos y complejos y para reflejar los cambios en el entorno en el cuál se efectúa la programación.

4. *Métodos de implementación.* El desarrollo de mejores métodos de implementación ha influido en la selección de las características que se habrán de incluir en los nuevos diseños.

5. *Estudios teóricos.* La investigación de las bases conceptuales del diseño e implementación de lenguajes , a través del uso de métodos de matemáticos formales, ha profundizado nuestro entendimiento de las fortalezas y debilidades de las características de los lenguajes y , por tanto, ha influido en la inclusión de estas características en los nuevos diseños de lenguaje.

6. *Estandarización.* La necesidad de lenguajes estándar que se puedan implementar con facilidad en una variedad de computadoras y que permita que los programas sean transportados de una computadora a otra ejerce una fuerte influencia conservadora sobre la evolución de los diseños de lenguajes.

A continuación se muestran algunas influencias importantes sobre los lenguajes de programación:

Año	Influencias y Nueva Tecnología
1951-55	*Hardware*: Computadoras de tubos de vacío; memorias de línea aplazada de mercurio. *Métodos*: Lenguajes ensamblador; conceptos base: subprogramas, estructuras de datos. *Lenguajes*: Uso experimental de compiladores de expresión.
1956-60	*Hardware*: Almacenamiento en cinta magnética; memorias de núcleo; circuitos de transistores. *Métodos*: Tecnología de compiladores inicial; gramáticas BNF; optimización de código; intérpretes; métodos de almacenamiento dinámicos y procesamiento de listas. *Lenguajes*: FORTRAN, ALGOL 58, ALGOL 60, COBOL, LISP.
1961-65	*Hardware*: Familias de arquitecturas compatibles, almacenamiento en discos magnéticos *Métodos*: Sistemas operativos de multiprogramación, compiladores de sintaxis-dirigida. *Lenguajes*: COBOL-61, ALGOL 60 (revisada), SNOBOL, JOVIAL, notación APL

1966-70	*Hardware*: Aumento de tamaño y velocidad y reducción de los costes; mini computadoras, microprogramación; circuitos integrados. *Métodos*: Sistemas interactivos y tiempos-compartidos; compiladores optimizados; sistemas de escritura traductores. *Lenguajes*: APL, FORTRAN 66, COBOL 65, ALGOL 68, SNOBOL 4, BASIC, PL/I, SIMULA 67, ALGOL-W
1971-75	*Hardware*: Microcomputadores; Edad de mini computadoras; sistemas de almacenamiento pequeños; declive de las memorias de núcleo y crecimiento de memorias de semiconductores *Métodos*: Verificación de programas; programación estructurada; inicio del crecimiento de ingeniería de software como disciplina de estudio *Lenguajes*: Pascal, COBOL 74, PL/I (standar), C, Scheme, Prolog
1976-80	*Hardware*: Microcomputadores de calidad comercial, sistemas de gran almacenamiento; computación distribuida. *Métodos*: Abstracción de datos; semánticas formales; técnicas de programación en tiempo real, concurrencia y fijos. *Lenguajes*: Smalltalk, Ada, FORTRAN 77, ML.

Año	Influencias y Nueva Tecnología
1981-85	*Hardware*: Computadores personales; primeras estaciones de trabajo; juegos de vídeo; redes de área local; Arpanet. *Métodos*: Programación orientada a objetos; entornos interactivos; editores de sintaxis dirigida. *Lenguajes*: Turbo Pascal, Smalltalk-80, crecimiento de Prolog, Ada 83, Postscript.
1986-90	*Hardware*: Edad de microcomputadores; crecimiento de estaciones de trabajo de ingenierías; arquitectura RISC; redes globales; Internet. *Métodos*: computación cliente/servidor. *Lenguajes*: FORTRAN 90, C++, SML (ML Standar).
1991-95	*Hardware*: Estaciones de trabajo y microcomputadores mucho más económicos; arquitectura paralelas masivas; voz, vídeo, fax, multimedia. *Métodos*: Sistemas abiertos; entorno de ventanas; Infraestructura de Información Nacional ("autopistas de la información"). *Lenguajes*: Ada 95, lenguajes de procesos (TCL, PERL).

La evolución de los lenguajes de programación ha estado guiada por la evolución de:

- Las computadoras y sus sistemas operativos.
- Las aplicaciones.
- Los métodos de programación.
- Los fundamento teóricos.
- La importancia dada a la estandarización.

Los lenguajes de programación han evolucionado a través de generaciones. En cada nueva generación, van necesitándose menos instrucciones para indicarle a la computadora que tarea efectuar. Es decir, un programa escrito en un lenguaje de primera generación (maquina y/o ensamblador) puede requerir mas de 100 instrucciones; ese mismo programa requerirá menos de 25 instrucciones en un lenguaje de tercera generación (Alto nivel).

1.4.1 Origen. (Autocódigos).

Los primeros lenguajes de programación surgieron de la idea de Charles Babagge a mediados del siglo XIX. Consistía en lo que el denominaba la maquina analítica, pero que por motivos técnicos no pudo construirse hasta mediados del siglo XX.
Con la colaboración de Ada Lovelace, la cual es considerada como la primera programadora de la historia, pues realizo programas para aquella supuesta maquina de Babagge, en tarjetas perforadas.
Como la maquina no llego nunca a construirse, los programas de Ada, lógicamente, tampoco llegaron a ejecutarse, pero si suponen un punto de partida de la programación.
En 1936, Turing y Post introdujeron un formalismo de manipulación de símbolos (la denominada máquina de Turing) con el que se puede realizar cualquier cómputo que hasta ahora podemos imaginar.
Esta fue una vía de comunicación entre los problemas formales de la computación y de la matemática. La unión permitió demostrar que no existe ninguna máquina de Turing que pueda reconocer si una sentencia es o no un teorema de un sistema lógico formal; pero también permitió demostrar que si un cálculo puede explicitarse sin ambigüedad en lenguaje natural, con ayuda de símbolos matemáticos, es siempre posible programar un computadora digital capaz de realizar el cálculo, siempre que la capacidad de almacenamiento de información sea la adecuada.
Desde el punto de vista de la ingeniería, los progresos en lenguajes de programación han sido paralelos a los diseños de las nuevas computadoras. Babbage ya escribió programas para sus máquinas, pero los desarrollos importantes tuvieron lugar, igual que en las computadoras, alrededor de la segunda guerra mundial.
Cuando surgió la primera computadora, el famoso Eniac, su programación se basaba en componentes físicos, o sea, que se programaba, cambiando directamente el Hardware de la maquina, lo que se hacia era cambiar cables de sitio para conseguir así la programación binaria.
Los "Lenguajes Maquina" y los "Lenguajes Ensambladores" (primera y segunda generación) son dependientes de la maquina. Cada tipo de maquina, tal como VAX de digital, tiene su propio lenguaje maquina distinto y su lenguaje ensamblador asociado. El lenguaje ensamblador es simplemente una representación simbólica del lenguaje maquina asociado, lo cual permite una programación menos tediosa que con el anterior. Sin embargo, es necesario un conocimiento de la arquitectura mecánica subyacente para realizar una programación efectiva en cualquiera de estos niveles de lenguajes.

1.4.2 Inicio. (FORTRAN, COBOL, ALGOL 60, LISP, APL, BASIC).

FORTRAN

En los años 50 se realizaron varios compiladores primitivos y fue en 1957 cuando apareció el primer compilador de FORTRAN. El compilador de FORTRAN (FORmula TRANslator) estaba diseñado para traducir a lenguaje máquina expresiones y operaciones matemáticas, e incluso permitía la manipulación de matrices.
La aparición del FORTRAN fue un gran cambio para los programadores que no todos aceptaron de buen grado. No les gustaba que sus programas fueran tratados por la computadora como meros datos, y argumentaban que el código máquina generado por el compilador nunca podría ser tan eficiente como el escrito por ellos directamente.

Esto no era generalmente así, puesto que el FORTRAN no fue diseñado pensando en crear un lenguaje bien estructurado sino pensando en crear un traductor de expresiones aritméticas a código máquina muy eficiente. Por ello, el diseño lógico de la computadora IBM 704 para el que fue creado casi puede deducirse del lenguaje FORTRAN.

En diferentes versiones, cada vez más estructuradas, el lenguaje FORTRAN se ha utilizado extensivamente desde que apareció hasta hoy en día, y puede considerarse el lenguaje estándar del cálculo científico.

COBOL

(COmmon Business Oriented Langauge = lenguaje orientado a negocios comunes). Se ha usado mucho desde los años 60 en aplicaciones de computadoras aplicadas a la administración.

Cobol ha a evolucionado a través de revisiones del diseño, que empezó con la primera versión en 1960 y que condujo a la ultima revisión en 1974 (COBOL 1974).

La implementación y el uso difundido de COBOL ha conducido a esfuerzos por estandarizar la definición del lenguaje. La primera definición estándar de COBOL se publico en 1968; la última definición revisada y actualizada apareció en 1974.

El aspecto más impresionante en un programa de COBOL es al organización en cuatro divisiones. Esta organización es en gran parte el resultado de dos objetivos del diseño: el de separar los elementos del programa dependientes de la maquina y el de separar las descripciones de los algoritmos , para que cada uno pueda modificarse sin afectar al otro. El resultado es una organización tripartita de programa.

Las representaciones de datos de COBOL , tienen un sabor definitivo de aplicación a los negocios pero son bastantes flexibles. La estructura de datos básica es el registro.

Los números y las cadenas de caracteres son los tipos de datos elementales básicos. Las operaciones primarias integradas incluyen operaciones aritméticas simples, lógicas y relaciónales.

El lenguaje es notable por su sintaxis parecida al ingles, lo que ase que los programas sean relativamente fáciles de leer. La mayoría de los compiladores de COBOL anteriores eran muy lentos, pero las mejoras resientes en las técnicas de compilación han conducido a compiladores de COBOL relativamente rápidos, lo que produce un código ejecutable bastante eficiente.

ALGOL60

Algoritmie Languge (lenguaje algortmico)
Lenguaje de programación creado en los años 60´s que se usaba principalmente para el diseño de aplicaciones de calculo, la versión más conocida data del año 1968 que se conoció con el nombre de Proporciona recursos para estructurar datos, similares a los que se pueden encontrar en lenguajes derivados de ALGOL 60 como Pascal, C o Modula-2. Permite sobrecarga de operadores y colocar una declaración en cualquier lugar donde pueda aparecer un enunciado (como en C++).

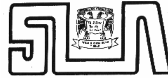
LISP

El lenguaje LISP lo diseño e implementó primero John Mc Carthy y un grupo del instituto de tecnología de Massachusetts al rededor de 1960. El lenguaje se usa mucho en las investigaciones computacionales en forma mas prominente en el área de inteligencia artificial (robótica, proceso del lenguaje natural prueba de teoremas, sistemas de inteligencia , etc.). LISP es diferente a la mayoría de los otros lenguajes en muchos aspectos. Lo más impresionante es la equivalencia de formas entre los programas y los datos en el lenguaje en el cual se permite que las estructuras de datos se ejecuten como programas y que los programas se ejecuten como datos. LISP ofrece una amplia variedad de primarias para la creación, destrucción y modificación de listas (incluyendo las listas de propiedad). Las primarias básicas de las operaciones aritméticas se proporcionan. Las estructuras de control en LISP son relativamente simples. Las expresiones usadas para construir programas se escriben en estricta forma polaca de cambrdge pueden incluir ramificación condicional.

Las referencias de LISP ase basan principalmente en la regla de asociación más reciente para referencias foráneas, que con frecuencia se implementan usando una simple lista enlazada de asociaciones actuales. LISP se implementa más fácilmente con un interprete de software y simulación de software para todas las primarias la mayoría de las implementaciones también proporcionan un compilador que puede usarse para compilar las definiciones de la función seleccionada en código de máquina. Estas funciones compiladas son ejecutables entonces por el interprete de hardware (pero sigue requiriéndose la simulación de software para muchas operaciones).

APL

Sus siglas significan (A Programming Language). Un lenguaje de programación.
Este programa fue desarrollado por Kenneth Inverson a mediados de la década de 1960 para resolver problemas matemáticos.
Este lenguaje se caracteriza por su brevedad y por su capacidad de generación de matrices y se utiliza en el desarrollo de modelos matemáticos.



APL tiene dos características que lo distinguen y lo apartan de los otros lenguajes:

- Lenguaje interactivo. APL es el único lenguaje que está diseñado expresamente para ser interactivo.
- Proceso directo de estructuras completas de datos. En APL las operaciones primarias aceptan arreglos completos como argumentos y producen arreglos completos como resultados. Por lo tanto, la unidad básica de datos tiende a ser un arreglo completo.

APL se basa en estructuras en datos de arreglos homogéneos, con componentes de tipo numérico o carácter. El tipo es estrictamente dinámico y no tiene relación directa con el programador. Está integrada una gran clase de operadores primarios, incluyendo muchos que crean, destruyen y modifican arreglos. Todas las primarias de APL se definen como funciones que regresan un valor.

BASIC

Begginer´s All Purpose Symbolic Instruction Code (código de Instrucciones Simbólicas de Todo Propósito para Principiantes).

Lenguaje simbólico de programación de tercera generación desarrollado en la década de los 60´s y destinado en sus inicios a la enseñanza de la programación

Por su sencillez fue el lenguaje básico utilizado inicialmente en los computadores personales. En general, y al contrario de lo que suele suceder con los más importantes lenguajes simbólicos de programación, en BASIC se suele trabajar en modo intérprete.

El tiempo de ejecución que emplea el programa compilado puede ser hasta diez veces inferior que el empleado en el caso de actuar bajo intérprete. Mientras se está en fase de desarrollo es muy conveniente trabajar en intérprete porque la inmediata ejecución del programa en cada momento permite detectar los errores con gran rapidez, lo que es muy positivo en un lenguaje destinado a principiantes que lógicamente cometen un gran número de errores al desarrollar un programa.

El BASIC es un lenguaje poco estandarizado y hay diferencias bastantes apreciables del que se utiliza en unos y otros sistemas.

Aunque es un lenguaje poco utilizado en la informática profesional, se ha empleado mucho en los microcomputadores y mientras en los primeros años de los 80´s fue el lenguaje en el que se solían desarrollar las aplicaciones en micros.

Hay una versión avanzada del BASIC, denominada QUICK BASIC que además de dar mayores prestaciones presenta las características de un lenguaje de programación algo estructurado, evitando así el desarrollo de los típicos programas espagueti.

A fines de los 90´s el BASIC ha dejado de utilizarse. Su heredero, aunque ya casi irreconocible, es el Visual Basic cuya orientación a objetos lo ha hecho casi imprescindible en la programación de la parte cliente de cualquier sistema cliente servidor.

1.4.3 Consolidación. (*PL/1, SIMULA 67, ALGOL 68, PASCAL)*.

PL/1.

PL1 son una sigla de Portfolio Lenguage One (Lenguaje de portafolio uno). es un lenguaje de programación creado por la compañía Atari. y su computadora de portafolio.
PL1 usa sólo 12 Kb de espacio y es más de 10 veces mas rápido que PBASIC. Soporta gráficos animados, marcos, líneas, conjuntos de caracteres, archivo y funciones del cordón, precisión doble la aritmética del punto flotante y programación estructurada. Para los usuarios más experimentados.
Es un idioma fácil para usar; la sintaxis es sencilla, y las reglas son uniformes. Tiene muchas funciones de ayuda dentro de la programación. Es un excelente primer idioma, y mantiene un camino de crecimiento excelente el profesional. PL/I puede usarse por resolver una gama amplia de problemas numéricos en cualquier esfera en donde se desarrolle la informática, puede ser en matemáticas, en la ingeniería, en cualquier ciencia, como en la medicina, etc.
PL/I está en casa en cualquier aplicación comercial. Los archivos son fáciles de usar. Usted puede hacer un informe en la pantalla, e introducir datos por medio del teclado.

El lenguaje también incluye un conjunto complejo de características para soportar el uso de archivos externos para la entrada y salida. La compilación de los programas en PL/l es difícil por la complejidad del lenguaje y la necesidad de compilar en forma eficiente el código ejecutable.

SIMULA67.

Esta versión surgió en 1967 es una extensión de Algol 60 bastante diferente a su predecesor (Simula I), que era básicamente un lenguaje de simulación de procesos. Simula 67 es un lenguaje de programación de propósito general que surge de los modelos de simulación. Parte del éxito de este lenguaje se debe a que se realizaron implementaciones para ordenadores IBM, DEC, Control Data y UNIVAC.

SIMULA 67 fue el primer lenguaje de programación que incorporó el concepto de clase. La clase de Simula se puede considerar como una generalización muy flexible del concepto de bloque de ALGOL. Su mecanismo de definición de subclases permite establecer jerarquías de clases.

Influyó directamente en las clases de C++ y SMALLTALK e indirectamente, a través de MESA, en los módulos de Modula-2. Su concepto de clase también de adaptó en lenguajes como CLU o ALPHARD.

ALGOL 68

Tercera revisión de ALGOL. Van Wijngaarden (E.U) proponía ampliar ALGOL, crear un "ALGOL Generalizado". Se basaba en la idea de que la complejidad de los lenguajes era la causa de sus limitaciones.

La premisa de que un lenguaje no debería estar enterrado entre reglas sintácticas, sino apoyado por ellas, condujo a un lenguaje en el que un fallo en la lógica de un programa era prácticamente

indetectable. Por otra parte, Niklaus Wirth prefería simplificar el lenguaje para que resultase más operativo. Wirth consideraba un error considerar una característica esencial de un lenguaje de alto nivel la capacidad de expresar un programa de la forma más breve posible.

ALGOL 68 se ha utilizado muy poco, en parte debido a que su informe original utilizaba una terminología nueva y era difícil de leer. Por ejemplo, llama modos (modes) a los tipos de datos y unidades (units) a las expresiones. No obstante, ALGOL 68 fue decisivo para la aparición de Pascal.

El lenguaje incorpora un conjunto relativamente pequeño de conceptos ortogonales. Se minimizan las restricciones acerca del uso combinado de sus distintas características.

ALGOL 68 utilizó la notación VWF (Van Wijngaarden Form) para definir su sintaxis e inglés semi-formal para especificar su semántica. La notación utilizada es más completa y adecuada para la definición de un lenguaje de programación que la BNF.

PASCAL.

El lenguaje de programación Pascal fue desarrollado originalmente por Niklaus Wirth, un miembro de la International Federation of Information Processing (IFIP) El Profesor Niklaus Wirth desarrolló Pascal para proporcionar rasgos que estaban faltando en otros idiomas en ese entonces.
Sus principales objetivos para Pascal eran para el lenguaje eran ser eficiente para llevarse a cabo y correrse los programas, permita bien el desarrollo de estructuras y también organizar programas, y para servir como un vehículo para la enseñanza de los conceptos importantes de programación de la computadora.
Pascal que se nombró gracias al matemático Blaise Pascal,. el original idioma de Pascal apareció en 1971 con última revisión publicada en 1973.
Fue diseñado para enseñar las técnicas de programación y otros temas a los estudiantes de la universidad y era el idioma de opción de los años 60 a los 80s.

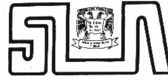
Pascal es un lenguaje estructurado en bloques. Un programa en Pascal está formado siempre por un solo bloque de programas principal, que contiene dentro de él definiciones del subprograma usado.

La parte más innovadora del diseño descansa en el trato de los tipos de datos. Una gran sección de diferentes tipos de datos se proporciona: enteros, reales, carácter, enumeraciones, boolianos, arreglos, registros, archivos cuenciales y una forma limitada de conjuntos. Pascal permite que se dé por el programador la definición separada de un tipo de datos.

Pascal puede implementarse en forma eficiente sobre computadoras convencionales. La traducción se hace en el código de máquina ejecutable, a pesar de que algunas implementaciones en Pascal traduce al código de máquina virtual, que interpreta y ejecuta un intérprete del software.

1.4.4 Lenguajes de los 80´s. (Prolog, Small Talk, C, Modula-2, Ada).

PROLOG
(programación lógica)



En Octubre de 1981, el gobierno japonés y más concretamente el Ministerio Japonés de Comercio Internacional e Industria (MITI), anuncia la puesta en marcha de un proyecto revolucionario equiparable a la carrera del espacio norteamericana.

Están dispuestos a ofrecer al mundo la siguiente generación, la Quinta Generación de computadoras. Unas máquinas de Inteligencia Artificial que pueden pensar, sacar conclusiones, emitir juicios e incluso comprender las palabras escritas y habladas.

La Quinta Generación prevé máquinas diseñadas para el tratamiento lógico, de capacidades análogas a las capacidades de anteriores generaciones de computadoras para tratar operaciones aritméticas. Se trata de computadoras que tienen el PROLOG como lenguaje nativo (lenguaje máquina), con capacidad para procesar millones de inferencias lógicas por segundo (LIPS).

La programación lógica, representada por lenguajes como Prolog, permite utilizar un subconjunto de la lógica de primer orden, concretamente las cláusulas de Horn, para especificar sistemas. Su base formal permite razonar sobre estas especificaciones, que pueden ser interpretadas directamente mediante resolución, con lo que podemos realizar pruebas de la especificación. Además su propio carácter de declarativos permite representar el qué, en lugar del cómo.
Sin embargo, el inconveniente de Prolog es que no está orientado al proceso. Al tratarse de un lenguaje secuencial, no permite especificar sistemas que interactúan con el entorno o sistemas compuestos por muchas partes que funcionan concurrentemente. Existen lenguajes lógicos concurrentes, como Parlog, que subsanan estas deficiencias.
Otro inconveniente que presentan estos lenguajes es que no admiten tipos de datos, ni siquiera definiciones de datos, por lo que son poco adecuados para especificaciones complejas. Podemos decir, por tanto, que, aunque a veces hayan sido usados para la especificación, los lenguajes lógicos no son lenguajes de especificación propiamente dichos.

C

El lenguje c, es un lenguaje de programación de tercera generación diseñada para poder crear aplicaciones de todo tipo inicialmente se usaba en entornos de tipo UNIX aunque después sé extendió en todas las plataformas. Su creador fue Dennis Ritchie.
El lenguaje C reúne características de programación intermedia entre los lenguajes ensambladores y los lenguajes de alto nivel; con gran poderío basado en sus operaciones a nivel de bits (propias de ensambladores) y la mayoría de los elementos de la programación estructurada de los lenguajes de alto nivel, por lo que resulta ser el lenguaje preferido para el desarrollo de software de sistemas y aplicaciones profesionales de la programación de computadoras.

Su diseño incluyo una sintaxis simplificada, la aritmética de direcciones de memoria (permite al programador manipular bits, bytes y direcciones de memoria) y el concepto de apuntador; además, al ser diseñado para mejorar el software de sistemas, se busco que generase códigos eficientes y uno portabilidad total, es decir el que pudiese correr en cualquier máquina. Logrados los objetivos anteriores, C se convirtió en el lenguaje preferido de los programadores profesionales.
En 1980 Bjarne Stroustrup de los laboratorios Bell de Murray Hill, New Jersey, inspirado en el lenguaje Simula67 adiciono las características de la programación orientada a objetos (incluyendo la ventaja de una biblioteca de funciones orientada a objetos) y lo denomino C con clases.
Para 1983 dicha denominación cambio a la de C++. Con este nuevo enfoque surge la nueva metodología que aumenta las posibilidades de la programación bajo nuevos conceptos.

MODULA-2

Modula-2 es descendiente directo de Pascal. Modula-2 tiene poderosas extensiones, especialmente su concepto de "modelo" (el cuál permite la compilación independiente y la creación de librerías de programas) y el permitir el procesamiento asíncrono. A pesar de su potencia, existen eficientes implementaciones de Modula-2 y junto con C y Ada, debe considerarse un importante candidato para las futuras aplicaciones de programación de sistemas. Modula-2, estuvo influenciado por las necesidades especilaes de la programación de sistemas, especialmente la necesidad de la compilación independiente de procedimiento.

SMALLTACK.

Smalltalk es un verdadero lenguaje de programación orientado a objetos integrado con un entorno de desarrollo multiventana. Smalltalk no es solo un hermoso lenguaje de computación orientado a objetos.
El entorno de desarrollo merece similar valoración y ha sido copiado muchas veces, desde el Sistema Operativo de Apple hasta MS Windows y Borland Pascal (en una menor extensión). Muchos conceptos de Smalltalk como los browsers y las técnicas de browsing han encontrado hoy su rumbo en muchas herramientas de desarrollo de la generación X.
Los entornos integrados de desarrollo Smalltalk poseen un factor "divertido-de-usar" que no estaba disponible previamente en lenguajes que requerían de las etapas de edición-compilación-depuración. Los cambios se graban instantáneamente y los mismos pueden probarse rápidamente.
El modelo de desarrollo incremental está realmente adoptado en Smalltalk. Smalltalk fue desarrollado dentro del Grupo de Investigación del Aprendizaje en el Centro de Investigación de Xerox en Palo Alto a comienzos de los 70'. Las principales ideas de Smalltalk se le atribuyen generalmente a Alan Kay con raíces en Simula, LISP y SketchPad. Dan Ingalls escribió el código de las primeras ventanas solapables, los pop-up menús y la clase BitBlt.
Adele Goldberg y Dave Robson escribieron los manuales de referencia para Smalltalk y fueron miembros clave del equipo de desarrollo. Un programa de licenciamiento de Xerox y Xerox Special Information Systems distribuyó el entorno de desarrollo Smalltalk a un limitado número de desarrolladores y grandes compañías. Sin embargo la distribución generalizada a la comunidad de desarrollo no sucedió hasta la fundación de una nueva compañía llamada ParcPlace Systems Inc., dirigida por Adele Goldberg. Un segundo Smalltalk (Smalltalk/V) fue desarrollado por Digitalk en los Angeles California, con financiamiento de Ollivetti y otros clientes.
Este Smalltalk estaba dirigido a cubrir la necesidad de un producto pequeño, de alta velocidad, basado en PC. Antes de la adquisición por parte de ParcPlace Systems Inc., Digitalk era el líder en volumen de ventas. Object Technology International Inc. (OTI) desarrolló un conjunto de herramientas de manejo para todos los Smalltalks llamado ENVY/Developer para proveer el control de versiones y el manejo de configuraciones en grandes proyectos.
OTI desarrolló una máquina virtual de 32-bits para el producto de Digitalk para Apple y participó en una amplia gama de proyectos de investigación, desde herramientas cliente servidor orientadas a objetos, Smalltalk integrado, y procesamiento de imágenes de radar para operaciones militares hasta sistemas con restricciones y generadores de máquinas virtuales portables. IBM desarrolló la familia de productos VisualAge para Smalltalk en colaboración con Object Technology International Inc. Hoy, ObjectShare (antiguamente ParcPlace-Digitalk) e IBM permanecen como los distribuidores dominantes de entornos de desarrollo en Smalltalk. Algunos nuevos Smalltalks se hallan en etapa de desarrollo.

ADA.

Nombrado en honor de la primera persona programador de computadoras del mundo, Augusta Ada Byron King, Condesa de Lovelace, e hija del poeta ingles Lord Byron.

Ada es un idioma de la programación de alto nivel pensado para las aplicaciones en vías de desarrollo donde la exactitud, seguridad, fiabilidad, y manutencion son primeras metas.

Ada es un fuertemente del tipo orientado a Objeto. Se piensa que trabaja bien en un ambiente del multi-lenguaje y ha standarizado los rasgos para apoyar la unión a otros idiomas.

La Razón de Ada proporciona una descripción de los rasgos principales del idioma y sus bibliotecas y explicaciones de las opciones hecha por los diseñadores del idioma.

El desarrollo del lenguaje Ada siguió un patrón único en la historia de los lenguajes más importantes de la programación. En los años 70's el Departamento de Defensa de Estados Unidos se comprometió a desarrollar un lenguaje de programación estándar para aplicaciones en sistemas incorporados, en los cuales en una o más computadoras son parte de un sistema más grande tal como un sistema de aeroplanos, barcos o de comunicaciones, se produjo un conjunto detallado de especificaciones que describían varias de las capacidades requeridas. Estas se volvieron a definir en forma progresiva a través de una serie de documentos preliminares conocidos como " Strawman ", " Woodenman ", "Tinman", "Ironman" y por último "Steelman". Cada uno de las especificaciones preliminares eran criticadas por un numeroso grupo de expertos en lenguajes de programación. Después de algunas revisiones de estos diseños, se sostuvo otra vez una critica extensa, que en 1979 condujo a la selección final de un diseño que después se llamo "Ada" (por Ada Lovelace, una pionera en computación). Whitaker (1978) da una historia más extensa del desarrollo del Ada.

El diseño del Pascal fue el punto de inicio para el diseño de Ada, pero el lenguaje resultante es diferente de él en muchos aspectos importantes. Como el lenguaje no ha tenido un uso muy extenso todavía, son probables algunos cambios posteriores como se gane más experiencia.

Se intenta que Ada soporte la construcción de programas grandes por medio de equipos de programadores, un programa en Ada se diseña por lo regular como una colección de "componentes de software" más grandes llamados paquetes.

Un paquete contiene un conjunto integrado de definiciones de tipo, datos objeto y subprogramas para manipular estos datos objeto. Esta naturaleza especial de Ada y de la programación en Ada viene de esta énfasis en la construcción de programas que usan paquetes.

El programa en Ada puede incluir un conjunto de tareas separadas que se van a ejecutar en forma concurrente.

Las estructuras del control de secuencias dentro de un subprograma Ada utiliza expresiones de estructuras de control de nivel de las proposiciones similares a las de Pascal. El aspecto más notable de las características de control de secuencia de Ada es la provisión de tareas que puede ejecutarse en forma concurrente y pueden controlarse usando un reloj y otros mecanismos de planeación.

Las estructuras del control de datos en Ada utiliza la organización de estructura en bloque estática para hacer referencias foráneas dentro de grupos pequeños de subprogramas, como Pascal.

Dominios de aplicación de los lenguajes de programación.

El lenguaje apropiado que se use a menudo, depende del dominio de la aplicación que resuelve el problema. El lenguaje adecuado que conviene usar para diversos dominios de alicación ha evolucionado a lo largo de los últimos30 años. A continuación se muestra una tabla con algunos de los lenguajes más importantes para diversos dominios de aplicación.

Epoca	Aplicación	Lenguajes principales	Otros lenguajes
Años Sesenta	Negocios	Cobol	Ensamblador
	Científica	FORTRAN	ALGOL, BASIC, APL
	Sistemas	Ensamblador	JOVIAL, Forth
	IA	Lisp	SNOBOL
Hoy	Negocios	COBOL, hoja de cálculo	C, PL/1, 4GL
	Científica	FORTRAN; C; C++	BASIC, Pascal
	Sistemas	C,C++	Pascal, Ada, BASIC, Modula
	IA	LISP, Prolog	
	Edición	TeX, Postscript, procesamiento de texto	
	Proceso	Shel de UNIX, TCL, PERL	Marvel
	Nuevos Paradigmas	ML, Smalltalk	Eiffel

1.4.5 Definición de un lenguaje de programación. (sintaxis, semántica, metalenguaje, especificación, gramática).

Un lenguaje de programación como lo definimos anteriormente es como un conjunto de reglas, símbolos y palabras especiales que permiten construir un programa.

Un lenguaje de programación es un lenguaje especial, no natural, diseñado con un vocabulario, morfología y sintaxis muy simples y rígidas y orientado a la programación de instrucciones elementales cuya ejecución por un determinado sistema físico da lugar a la realización de una tarea

1.4.5.1. Sintaxis

La sintaxis es el conjunto de reglas que gobiernan la construcción o formación de sentencias (instrucciones) válidas en un lenguaje. La sintaxis de un lenguaje de programación es el aspecto que ofrece el programa. Proporcionar las reglas de sintaxis para un lenguaje de programación significa decir cómo se escriben los enunciados, declaraciones y otras construcciones de lenguaje.

La sintaxis, cuya definición sería "la disposición de palabras como elementos en una oración para mostrar su relación,"' describe la serie de símbolos que constituyen programas válidos.

Solamente las sentencias correctamente sintácticas pueden ser traducidas por un lenguaje de programación, y los programas que contienen errores de sintaxis son rechazados por la computadora. Cada lenguaje de programación posee sus propias reglas sintácticas.
El vocabulario de un lenguaje es un conjunto de símbolos (en ocasiones se denominan símbolos terminales). Los símbolos usuales son: letras, dígitos, símbolos especiales (, ; : / & + - *, etc.), palabras reservadas o claves if (si), then(entonces), repeat (repetir), for (o), begin (inicio), end (fin).
Las reglas sintácticas son los métodos de producción de sentencias o instrucciones válidas que permitirán formar un programa. Las reglas sintácticas permiten reconocer si una cadena o serie de símbolos es correcta gramaticalmente y a su vez información sobre su significado o semántica.
Las reglas sintácticas deben definir los conceptos de sentencia (instrucción), expresión, identificador, variables, constantes, etc., y deben permitir de modo fácil verificar si una secuencia de símbolos es una sentencia, expresión, etc., correcta del lenguaje.

Para definir las reglas sintácticas se suelen utilizar dos tipos de notaciones: la formalización de Backus-Naur-Form (BNF) y los diagramas o grafos sintácticos. La notación o formalización BNF, es uno de los métodos empleados para la definición de reglas sintácticas y se concibió para que se permitiera decidir en forma algorítmica cuándo una sentencia es válida o no en un lenguaje. Esta notación fue ideada por P. Naur que junto con otro grupo de científicos desarrollaron en 1960 el lenguaje de programación ALGOL 60.

El estilo sintáctico general de un lenguaje, esta dado por la selección de diversos elementos sintácticos básicos. Los más destacados son los siguientes:

1. Conjunto de caracteres.
2. Identificadores.
3. Símbolos de operadores.
4. Palabras clave y palabras reservadas.
5. Comentarios
6. Espacios en blanco.
7. Delimitadores y corchetes.
8. Formato de campos libres y fijos.
9. Expresiones.
10. Enunciados.

1.4.5.2. Semántica.

La semántica es el conjunto de reglas que proporcionan el significado de una sentencia o instrucción del lenguaje. La semántica de un lenguaje de programación es el significado que se da a las diversas construcciones sintácticas. Por ejemplo, para proporcionar la sintaxis que se usa en Pascal para declarar un vector de 10 elementos, V, de enteros se daría una declaración en Pascal, como:
V: array[0..9] of entero;

Por otra parte, en C, se especificaría como:
int V[10];

Si bien ambos crean objetos de datos similares en el tiempo de ejecución, su sintaxis es muy diferente. Para entender el significado de la declaración, se necesita conocer la semántica tanto de Pascal como de C para esta clase de declaraciones de arreglo.

1.4.5.3. Gramática.

La definición formal de la sintaxis de un lenguaje de programación se conoce ordinariamente como una gramática, en analogía con la terminología común para los lenguajes naturales. Una gramática se compone de un conjunto de reglas (llamadas producciones) que especifican las series de caracteres (o elementos léxicos) que forman programas permisibles en el lenguaje que se está definiendo. Una gramática formal es simplemente una gramática que se especifica usando una notación definida de manera estricta. Las dos clases de gramáticas útiles en tecnología de compiladores incluyen la gramática BNF (o gramática libre del contexto) y la gramática normal.

1.4.6 Translación de lenguajes de programación. (compilación, programa fuente, análisis léxico, análisis sintáctico, análisis semántico, tabla de símbolos, generación de código, programa objeto).

El proceso de traducción de un programa, de su sintaxis original a una forma ejecutable medular en toda implementación de lenguajes de programación. La traducción puede ser bastante sencilla, como en el caso de los programas en Prolog o LISP, pero con frecuencia, el proceso puede ser bastante complejo. Casi todos los lenguajes se podrían implementar con sólo una traducción trivial si uno estuviera dispuesto a escribir un intérprete de software y a aceptar velocidades lentas de ejecución. En la mayoría de los casos, sin embargo la ejecución eficiente es un objetivo tan deseable que se hacen esfuerzos importantes para traducir los programas, a estructuras ejecutables con eficiencia, en especial código de máquina interpretable por hardware. El proceso de traducción se vuelve gradualmente más complejo a medida que la forma ejecutable del programa se aleja más en cuanto a estructura respecto al original

1.4.6.1. Compilación.

Los traductores se agrupan de acuerdo con el número de pasos que efectúan sobre el programa fuente. El compilador estándar emplea típicamente dos pasos sobre el programa fuente. El primer paso de análisis descompone el programa en los componentes que lo constituyen y obtiene información, como el uso de nombres de variables, del programa. El segundo paso genera típicamente un programa objeto a partir de esta información recogida.

Traducción (compilación). En términos generales un traductor denota cualquier procesador de lenguajes que acepta programas en cierto lenguaje fuente (que puede ser de alto o de bajo nivel) como entrada y produce programas funcionalmente en otro lenguaje objeto (que también puede ser de alto o de bajo nivel) como salida:

Hay varios tipos especializados de traductores:
Un *ensamblador* es un traductor cuyo lenguaje objeto es también alguna variedad de lenguaje máquina para una computadora real pero cuyo lenguaje fuente, un lenguaje ensamblador, constituye en gran medida una representación simbólica del código de máquina objeto. Casi todas las instrucciones en el lenguaje fuente se traducen una por una a instrucciones en el lenguaje objeto.

Un *compilador* es un traductor cuyo lenguaje fuente es un lenguaje de alto nivel y cuyo lenguaje objeto se aproxima al lenguaje de máquina de una computadora real, ya sea que se trate de un lenguaje ensamblador o alguna variedad de lenguaje máquina. El C por ejemplo, se compila comúnmente a un lenguaje ensamblador, el cuál es convertido luego en lenguaje de máquina por un ensamblador.

Un *cargador o editor de vínculos* es un traductor cuyo lenguaje objeto es un código de máquina real y cuyo lenguaje fuente es casi idéntico; y está compuesto por lo general de programas en lenguaje de máquina en forma reubicable junto con tablas de datos que especifican puntos donde el código reubicable se debe modificar para volverlo auténticamente ejecutable.

Un *preprocesador o un macroprocesador* es un traductor cuyo lenguaje objeto es la forma ampliada de un lenguaje de alto nivel como C++ o Pascal y cuyo lenguaje objeto es la forma estándar del mismo lenguaje. El programa objeto que produce un prepocesador queda listo entonces para ser traducido y ejecutado por los procesador lenguaje estándar. La traducción de un lenguaje fuente de alto nivel a programas ejecutables en lenguaje de máquina suele implicar más de un paso de traducción

1.4.6.2. Análisis del programa fuente.

Para un traductor, el programa fuente se presenta inicialmente como una serie larga y no diferenciada de símbolos, compuesta de miles o decenas de miles de caracteres. Desde luego, un programador que ve un programa así lo estructura casi de inmediato en subprogramas, enunciados, declaraciones, etc. Para el traductor nada de esto es manifiesto. Durante la traducción se debe construir laboriosamente, carácter por carácter, un análisis de la estructura del programa.

1.4.6.3. Análisis léxico.

La fase fundamental de cualquier traducción es agrupar esta serie de caracteres en sus constituyentes elementales: identificadores, delimitadores, símbolos de operadores, números, palabras clave, palabras pregonadas, espacios en blanco, comentarios, etc. Esta fase se conoce como análisis léxico, y las unidades básicas de programa que resultan del análisis léxico se llaman elementos (o componentes) léxicos. Típicamente, el analizador (o revisor) es la rutina de entrada para el traductor; lee renglones sucesivos del programa de entrada, los descompone en elementos léxicos individuales y alimenta estos elementos léxicos a las etapas posteriores del traductor para su uso en los niveles superiores de análisis. El analizador léxico debe identificar el tipo de cada elemento léxico (número, identificador, delimitador, operador, etc.) y adjuntar una marca de tipo. Además, se suele hacer la conversión a una representación interna de elementos como números (que se convierten a forma binaria interna de punto fijo o flotante) e identificadores (que se guardan en una tabla de símbolos y se usa la dirección de la entrada de la tabla de símbolos en lugar de la cadena de caracteres). El modelo básico que se usa para proyectar analizadores léxicos es el autóma de estados finitos.
.
Si bien el concepto de análisis léxico es sencillo, esta fase de la traducción suele requerir una mayor proporción del tiempo de traducción que cualquier otra. Este hecho se debe en parte simplemente a la necesidad de explorar y analizar el programa fuente carácter por carácter.

1.4.6.4. Análisis sintáctico.

La segunda etapa de la traducción es el análisis sintáctico (parsing). En ella se identifican las estructuras de programa más grandes (enunciados declaraciones, expresiones, etc.) usando los elementos léxicos producidos por el analizador léxico. El análisis sintáctico se alterna ordinariamente con el análisis semántico. Primero, el analizador sintáctico identifica una serie de elementos léxicos que forman una unidad sintáctica como una expresión, enunciado, llamada de subprograma o declaración. Se llama entonces a un analizador semántico para que procese esta unidad. Por lo común, el analizador sintáctico y el semántico se comunican usando una pila. El analizador sintáctico introduce en la pila los diversos elementos de unidad sintáctica hallada y el analizador semántico los recupera y los procesa. Gran cantidad investigación se ha enfocado al descubrimiento de técnicas eficientes de análisis sintáctico, en particular técnicas basadas en el uso de gramáticas formales.

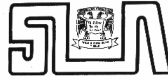
1.4.6.5. Análisis semántico.

El análisis semántico es tal vez la fase medular de la traducción. Aquí, se procesan las estructuras sintácticas reconocidas por el analizador sintáctico y la estructura del código objeto ejecutable comienza a tomar forma. El análisis semántico es, por tanto, el puente entre las partes de análisis y de síntesis de la traducción. En esta etapa también ocurre un cierto número de otras funciones subsidiarias importantes, entre ellas el mantenimiento de las tablas de símbolos, la mayor parte de la detección de errores, la expansión de macros y la ejecución de enunciados de tiempo de compilación. El analizador semántico puede producir en efecto el código objeto ejecutable en traducciones sencillas, pero es más común que la salida de esta etapa sea alguna forma interna del programa ejecutable final, la cual es manipulada luego por la etapa de optimización del traductor antes que se genere efectivamente código ejecutable.

1.4.6.6. Tabla de símbolos.

El analizador semántico divide ordinariamente en un conjunto de analizadores semánticos más pequeños, cada uno de los cuales maneja un tipo particular de construcción de programa. Los analizadores semánticos interactúan entre ellos mismos a través de información que se guarda en diversas estructuras de datos, en particular en la *tabla central de símbolos*. Por ejemplo un analizador semántico que procesa declaraciones de tipo para variables sencillas suele poder hacer poco más que introducir los tipos declarados en la *tabla de símbolos.* Un analizador semántico posterior que procesa expresiones aritméticas puede usar luego los tipos declarados para generar las operaciones aritméticas apropiadas específicas de tipo para el código objeto. Las funciones exactas de los analizadores semánticos varían considerablemente, según el lenguaje y la organización lógica del traductor. Algunas de las funciones más comunes se pueden describir como sigue:

Mantenimiento de tablas de símbolos. Una tabla de símbolos es una de las estructuras de datos medulares de todo traductor. La tabla de símbolos contiene típicamente una entrada por cada identificador diferente encontrado en el programa fuente. El analizador léxico efectúa las introducciones iniciales conforme explora el programa de entrada, pero los analizadores semánticos tienen la responsabilidad principal a partir de ese momento. La entrada de tabla de símbolos



contiene más que sólo el identificador mismo; contiene datos adicionales respecto a los atributos de ese identificador: su tipo (variable simple, nombre de arreglo, nombre de subprograma, parámetro formal, etc.), tipo de valores (enteros, reales, etc.), entorno de referimiento, y cualquier otra información disponible a partir del programa de entrada a través de declaraciones y uso. Los analizadores semánticos introducen esta información en la tabla de símbolos conforme procesan declaraciones, encabezados de programa y enunciados de programa. Otras partes del traductor usan esta información para construir código ejecutable eficiente.
La tabla de símbolos de los traductores para lenguajes compilados se suele desechar al final de la traducción. Sin embargo, puede retenerse durante la ejecución, por ejemplo, en lenguajes que permiten crear nuevos identificadores durante la ejecución o como ayuda para la depuración. Todas las implementaciones de ML, Prolog y LISP utilizan una tabla de símbolos creada inicialmente durante la traducción como una estructura de datos central definida por el sistema en tiempo de ejecución.

Detección de errores. Los analizadores sintácticos y semánticos deben esta para manejar programas tanto incorrectos como correctos. En cualquier punto el analizador léxico puede enviar al analizador sintáctico un elemento sintáctico que no encaja en el contexto circundante (por ejemplo, un delimitador de enunciado en medio de una expresión, una declaración a la mitad de una serie de enunciados, un símbolo de operador donde se espera un identificador).

1.4.6.7. Síntesis del programa objeto

Las etapas finales de la traducción se ocupan de la construcción del programa ejecutable a partir de las salidas que produce el analizador semántico. Esta fase implica necesariamente generación de código y también puede incluir optimización del programa generado. Si los subprogramas se traducen por separado, o si se emplean subprogramas de biblioteca, se necesita una etapa final de vinculación y carga para producir el programa completo listo para ejecutarse.

Optimización. El analizador semántico produce ordinariamente como salida el programa ejecutable traducido y representado en algún código intermedio.
A partir de esta representación interna, los generaciones de código pueden crear el código objeto de salida con el formto apropiado. Sin embargo, antes de la generación de código, se lleva a cabo ordinariamente cierta optimización del programa en la representación interna. Tipicamente el analizador semántico genera la forma del programa interno en forma irregular conforme se analiza cada segmento del programa de entrada.

1.4.6.8. Generación de código.

Después que se ha optimizado el programa traducido en la representación interna, se debe transformar en los enunciados en lenguaje ensamblador, código de máquina u otra forma de programa objeto que va a constituir la salida de la traducción. Este proceso implica dar el formato apropiado a la salida con base en la información que contiene la representación interna de] programa. El código de salida puede ser directamente ejecutable o puede haber otros pasos de traducción por seguir, por ejemplo, ensamblado o vinculación y carga.

1.4.6.9. Programa objeto.

Vinculación y carga. En la etapa final optativa de la traducción, los fragmentos de código que son resultado de las traducciones individuales de subprogramas se funden en el programa final ejecutable. La salida de las fases de traducción precedentes consiste típicamente en programas ejecutables en una forma casi final, excepto cuando los programas hacen referencia a datos externos u otros subprogramas. Estas ubicaciones incompletas en el código se especifican en las tablas de cargador anexas que produce el traductor. El cargador vinculador (o editor de vínculos) carga los diversos segmentos de código traducido en la memoria y luego usa las tablas de cargador anexas para vincularlos correctamente entre sí introduciendo datos y direcciones de subprograma en el código según se requiere. El resultado es el programa ejecutable final listo para usarse.

1.4.7 Interpretación.

En cualquier Lenguaje de alto nivel en que se escriba un programa, éste debe ser traducido a lenguaje máquina antes de que pueda ser ejecutado. Esta conversión de instrucciones de alto-nivel a, instrucciones a nivel de máquina se hace por programas de software del sistema, denominados compiladores e intérpretes. Estos programas especiales se denominan en general traductores.

El proceso de traducción y su conversión en programa objeto, difiere según que el programa sea compilador o intérprete.

Compiladores.

Un compilador es un programa que traduce el programa fuente (conjunto de instrucciones de un lenguaje de alto nivel, por ejemplo COBOL o Pascal) a programa objeto (instrucciones en lenguaje máquina que la computadora pueda interpretar y ejecutar). El compilador efectúa sólo la traducción, no ejecuta el programa. Una vez compilado el programa, el resultado en forma de programa objeto será directamente ejecutable.

Intérpretes

Los lenguajes de programación además de ser compilados pueden ser interpretados. Un intérprete es un programa que procesa los programas escritos en un lenguaje de alto nivel, sin embargo, está diseñado de modo que no existe independencia entre la etapa de traducción y la etapa de ejecución. Un. intérprete traduce cada instrucción o sentencia del programa escrito en un lenguaje “L” a código máquina e inmediatamente se ejecuta, y a continuación se ejecuta la siguiente sentencia. Ejemplos de intérpretes son las versiones de BASIC que se utilizan en la mayoría de las microcomputadoras bien en forma residente (en memoria ROM), bien en forma no residente (en disco).

El intérprete está diseñado para trabajar con la computadora en modo conversacional o interactivo, en realidad se le dan órdenes al procesador a través del intérprete con ayuda, por ejemplo de un teclado y un programa denominado editor. El procesador ejecuta la orden una vez que ésta es

traducida, si no existe ningún error de sintaxis y se devuelve el control al programador con indicación de mensajes (errores de sintaxis, de ejecución, etc.).

El intérprete no traduce todo el programa fuente en un solo paso, sino que ejecuta cada instrucción del programa fuente antes de traducir y ejecutar la siguiente.

El intérprete se sitúa en memoria principal (RAM), junto con el programa del usuario. De este modo el programa no se ejecutará directamente traducido a lenguaje máquina, sino a través de la interpretación que se producirá al ejecutarse el programa interpretador en una especie de traducción simultánea.

Comparación entre intérpretes y compiladores

Los intérpretes y compiladores tienen ventajas e inconvenientes derivados de sus peculiares características. La principal ventaja de los interpretes sobre los compiladores reside en que el análisis sintáctico del programa, se puede ir realizando a medida que se introduce por teclado o bien cuando se interpreta. El intérprete proporciona un error e incluso un mensaje de diagnóstico que indica la naturaleza del problema. El programador puede ir directamente al error y corregirlo; de este modo los programas pueden ser comprobados y corregidos durante el desarrollo de los mismos. Las restantes características a destacar que diferencian un intérprete de un compilador se resumen a continuación.

Los lenguajes compiladores presentan la ventaja considerable frente a los intérpretes de la velocidad de ejecución, por lo que su uso será mejor en aquellos programas probados en los que no se esperan cambios y que deban ejecutarse muchas veces. Así mismo, en general, ocuparán menos memoria en el caso de programas cortos ya que en el caso de interpretación, el interpretador que tendrá un tamaño considerable debe residir siempre en memoria.
Los lenguajes intérpretes encuentran su mayor ventaja en la interacción con el usuario, al facilitar el desarrollo y puesta a punto de programas, ya que los errores son fáciles de detectar y sobre todo de corregir. Por el contrario, como el intérprete no produce un programa objeto, debe realizar el proceso de traducción cada vez que se ejecuta un programa y por ello será más lento el intérprete.

En la actualidad los programadores suelen aprovechar con frecuencia las ventajas tanto de intérpretes como de compiladores. En primer lugar desarrollan y depuran los programas utilizando un intérprete interactivo (por ejemplo, MBASIC intérprete); después compilan el programa terminado a fin de obtener un programa objeto (por ejemplo, MBASIC compilador).

El intérprete ocupa un lugar central en la operación de una computadora. Típicamente el intérprete ejecuta el algoritmo cíclico del diagrama siguiente. Durante cada ciclo, el intérprete obtiene la dirección de la instrucción siguiente del registro de direcciones de programa (e incrementa el valor del registro para que sea la dirección de instrucción que sigue), toma la instrucción designada de la memoria, decodifica la instrucción a un código de operación y un conjunto de designadores de operando, toma los operandos designados (si es necesario) e invoca la operación designada con los operandos designados como argumentos. La operación primitiva puede modificar datos de la memoria o los registros, acceder a dispositivos de entrada/salida o cambiar el orden de ejecución

modificando el contenido del registro de direcciones de programa. Después de la ejecución de la primitiva, el intérprete repite simplemente el ciclo anterior.

Tomar próxima instrucción

Decodificar instrucción (determinar los designadores de opración y operandos

Tomar operandos designados

Saltar a operación designada

Ejecutar Operación Primitiva $_1$

Ejecutar Operación Primitiva $_2$

Ejecutar Operación Primitiva $_k$

Ejecutar Operación de detener

Detener

Interpretación y ejecución e programas

Interpretación y ejecución de programas

Bibliografía.

- Metodología de la programación
  Joyanes, Luis
  Ed. Mc-Graw Hill
  México, 1991

- Lenguajes de programación: diseño e implementación
  Terréense W. Pratt, Marvin V, Zelkowitz
  Prentice-Hall Hispanoamericana, S. A.
  Tercera Edición
  México, 1998

2. PROGRAMACIÓN IMPERATIVA

Introducción.

Es llamada así porque esta basada en comandos que actualizan variables que están en almacenamiento. Surge en 1950, cuando los programadores reconocieron que las variables y comandos de asignación constituyen una simple pero útil abstracción de memoria que se actualiza.
Los lenguajes imperativos o de procedimiento *son lenguajes controlados por mandatos u orientados a enunciados (instrucciones).* Un programa se compone de una serie de enunciados, y la ejecución de cada enunciado hace que el intérprete cambie el valor de una localidad o más en su memoria, es decir, que pase a un nuevo estado.
El desarrollo de programas consiste en construir los estados de máquina sucesivos que se necesitan para llegar a la solución. Ésta suele ser la primera imagen que se tiene de la programación, y muchos lenguajes de uso amplio (por ejemplo, C, C++, FORTRAN, ALGOL, PL/I, Pascal, Ada, Smalltalk, COBOL) manejan este modelo.

Los lenguajes de programación de la tercera generación, la cual se inició en los años 60, se conocen como lenguajes de alto nivel o de procedimientos. Una gran variedad de lenguajes se crearon para satisfacer los requisitos de las aplicaciones. Por lo que se encuentran lenguajes científicos (como el FORTRAN), lenguajes para empresas (como el COBOL) y lenguajes de uso general (incluyendo el BASIC). Los lenguajes de procedimientos deben ser traducidos al código de máquina que la computadora pueda entender. Sin embargo, estos lenguajes permiten al programador señalar cómo se debe efectuar una tarea a un nivel mayor que en los lenguajes de ensamble.
La gran mayoría de las aplicaciones de computadoras de uso en la actualidad, se escribieron utilizando lenguajes de procedimientos de la tercera generación. Se puede esperar que estos lenguajes todavía se utilicen en el futuro.

2.1. Definición

La programación imperativa se define como *un modelado de la realidad por medio de representaciones de la información y de un conjunto de acciones a realizar*. Orden de las acciones en el tiempo.

También podemos decir que es una representación simbólica de una posición de memoria que cambia de valor.

Los lenguajes imperativos son lenguajes controlados por mandatos u orientados a enunciados (instrucciones).

Su sintaxis genérica tiene la forma:

```
enunciado1;
enunciado2;
```

Ejemplos de estos lenguajes son C, C++, FORTRAN, ALGOL, PL/I, Pascal, Ada, Smalltalk, y COBOL).

Debido a su relación cerrada con las arquitecturas de las máquinas, los lenguajes de Programación Imperativa pueden ser implementados eficientemente, la razón fundamental de la Programación Imperativa esta relacionado a la naturaleza y propósito de la programación.

A. Trabajando con iteraciones: Las instrucciones que indican la repetición o iteración se llaman instrucciones iterativas. Ordenan a la UCP que itere o vuelva a ejecutar ciertas instrucciones y el número de veces respectivo. Usa variables locales para acumular él producto y para controlar las iteraciones.
B. Trabajando con recursión : No utiliza variables locales.

Paradigma imperativo

• Modelo: máquina de estados Von Neumann
• Solución del problema 'paso a paso'
• Evolución del lenguaje máquina
• Referencialmente opacos: el resultado de una expresión no es independiente del lugar donde aparece (depende de la historia; efectos colaterales).

Aspectos de Programación Imperativa (Procedimental)

1. Cualquier fragmento aislado de programa debe entenderse y mejorarse con facilidad.
2. Las dificultades comienzan cuando el efecto del cambio puede extenderse a través de un programa muy grande, tal vez introduciendo errores en "un rincón olvidado". Estos errores pueden permanecer sin detección durante años.
3. La estructuración es clave para manejar programas muy grandes. La legibilidad de un programa puede mejorarse organizándolo de tal manera que cada parte pueda entenderse en forma relativamente independiente del resto.
4. La estructura ayuda a mantener la situación dentro del límite de la atención humana.
5. Miller observó que la gente es capaz de recordar aproximadamente siete cosas, pudiendo ser bits, palabras, colores, tonos sabores, etc. Por lo tanto tentativamente podemos suponer que nuestras memorias se encuentran limitadas por el número de símbolos o unidades que podemos manejar, y no por la información que representen esos símbolos.
6. De esta manera, es beneficioso organizar de modo inteligente el material antes de tratar de memorizarlo, permitiéndonos empaquetar la misma cantidad de información en mucho menos símbolos y facilitar así la tarea de memorizar.

Características de los Lenguajes Imperativos

- Estado implícito
- Comandos o Instrucciones
  - asignación, saltos condicionales e incondicionales, bucles...
  - afectan o modifican el estado

Existen muchos lenguajes diferentes de programación imperativos u orientados a procedimientos. Algunos son ampliamente utilizados, otros se hicieron para fines específicos o, incluso, para una sola instalación u organización.

Las características generales de las instrucciones de los lenguajes de alto nivel y las características de otros lenguajes que se utilizan ampliamente, pueden agruparse en las categorías de operaciones de entrada / salida, especificaciones para el formato de datos, especificaciones aritméticas, instrucciones para la transferencia de control, especificaciones para almacenamiento, repetición e instrucciones de documentación.

Propiedades generales de los lenguajes imperativos o de procedimientos.

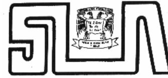
- Orientados a la utilización por programadores profesionales.
- Requiere especificación sobre cómo ejecutar una tarea.
- Se deben especificar todas las alternativas.
- Requiere gran número de instrucciones de procedimiento.
- El código puede ser difícil de leer, entender y mantener.
- Lenguaje creado originalmente para operación por lotes.
- Puede ser difícil de aprender.
- Difícil de depurar.
- Orientados comúnmente a archivos.

2.2. Lenguajes de programación imperativa

Se estudiarán varios lenguajes de programación utilizados comúnmente en Estados Unidos. Este análisis dará una idea acerca de cómo se usan en la elaboración de programas de cómputo para resolver problemas de procesamiento en el entorno de los negocios y la administración.

2.2.1. FORTRAN

El lenguaje de programación FORTRAN (FORmula TRANslation), diseñado en 1957, se creó originalmente para los sistemas de computación IBM. Sin embargo, su empleo se difundió rápidamente, y está disponible en casi cualquier tipo de sistema de cómputo.
Utilizado principalmente para efectuar cálculos aritméticos, algebraicos y numéricos, el FORTRAN está orientado a los procedimientos, es decir, el programa de instrucciones debe definir claramente los procedimientos aritméticos, de entrada / salida y cualquier otro



que se desee, sin preocuparse de cómo se realizan las operaciones en la unidad central de procesamiento.
En el FORTRAN, los enunciados de entrada / salida y los de FORMAT se aplican conjuntamente. Los enunciados de READ (leer) y WRITE (escribir), sirven para las funciones de entrada y salida, cada uno valiéndose de un enunciado FORMAT (formatear) que especifica el tipo de datos, su dimensión y su ubicación en los medios de entrada o de salida.
Los enunciados FORMAT son las declaraciones para especificar los datos que corresponden a las instrucciones particulares de entrada / salida.

2.2.2. BASIC

El lenguaje BASIC (acrónimo del inglés BegInner's AII-purpose Symbolic Instruction Code) se originó en 1964 en el Dartmouth College (con el apoyo de la National Science Foundation). Este lenguaje simbólico de fácil comprensión es similar al FORTRAN. Cuando se introdujo, el BASIC fue destinado para utilizarse en los grandes sistemas de computación de tiempo compartido empleados por varios usuarios. También se le diseñó para instruir a los estudiantes sobre cómo trabajan las computadoras y cómo programarías. Así que los recursos incorporados al BASIC son muy sencillos de entender y recordar.
Desde su introducción, el BASIC ha tenido amplia aceptación en los campos de la educación, investigación e industria. Se aplica en computadoras de todos tamaños; no obstante, es el lenguaje dominante en las microcomputadoras. Muchas computadoras pequeñas incluyen el lenguaje BASIC como parte del paquete original de compra, porque es muy fácil de aprender y, también, por la poderosa capacidad de procesamiento del lenguaje en sí.
En el BASIC, a diferencia del FORTRAN no es necesario valerse de enunciados FORMAT como los de INPUT y PRINT. La computadora lee o imprime los datos directamente conforme se presentan, sin reordenarlos (editarlos) o cambiar su formato (algunas versiones del BASIC permiten el uso de las posibilidades de formato como una opción, pero no es obligatorio como en el FORTRAN).
Además de los datos numéricos, el BASIC, al igual que muchos otros lenguajes, permite el empleo de información con caracteres, llamada cadenas de caracteres. Una cadena puede ser una sola letra o símbolos (como "A", "B", "1", "9", "+" y "$"), o bien diferentes caracteres utilizados en conjunto (como "ABC", "PAGO" y "JUAN"). Adviértase que las cadenas de caracteres se encierran entre comillas y que las comas normales necesarias en la redacción en lenguaje común se colocan fuera de las comillas puesto que tales comas no forman parte de la cadena de caracteres. Así es como se manejan cadenas de datos en el BASIC. En el ejemplo de la nómina, los datos de cada empleado se tratarían como cadenas de datos.
Las operaciones aritméticas elementales en el BASIC son las mismas que en la mayoría de los otros lenguajes. Los elementos de los datos se suman, restan, dividen o multiplican, sirviéndose de los operadores bien conocidos: +, - , / y *. Se pueden incluir en una instrucción variables o constantes.
El control de la ejecución se maneja por medio de enunciados IF-THEN-ELSE. Estas instrucciones permiten al programador probar determinadas condiciones y, después, dependiendo de la condición, transferir el control a una instrucción que se encuentre en cualquier otra parte del programa. En el BASIC a todos los enunciados se les asigna

automáticamente un número de identificación, por lo que es muy fácil referirse a uno en especial.

Las condiciones se expresan en el BASIC con los siguientes operadores:
1. Mayor que: >
2. Menor que: <
3. Igual a: =
4. Mayor que o igual a: > = o =
5. Menor que o igual a: < = o =
6. Diferente de: < >

2.2.3. COBOL

El lenguaje COBOL (acróstico de COmmon Business-Oriented Language) fue creado por un comité de representantes de empresas y universidades patrocinado por el gobierno estadounidense. Las especificaciones para el lenguaje se concluyeron en 1959 y se dieron a conocer durante 1960. Continuamente mejorado y actualizado, es el lenguaje de programación dominante en el área de procesamiento de datos para las empresas en la actualidad.
Puesto que el COBOL fue diseñado explícitamente para el uso en aplicaciones de las empresas, las instrucciones están escritas en un lenguaje semejante al idioma inglés que se parece al empleado en las organizaciones. Además, el COBOL es independiente de la máquina; así, un programa escrito, por ejemplo, para una computadora IBM puede transferirse fácilmente a otros sistemas de cómputo. El COBOL es un lenguaje fácil de aprender que es eficaz si el manejo de datos alfabéticos y alfanuméricos (por ejemplo, nombres de clientes, direcciones y descripciones de mercancías) que se emplean tan a menudo en las organizaciones comerciales. Estas ventajas serán más evidentes a medida que se den detalles del programa modelo.
Los programas de COBOL se forman constructivamente. El elemento del nivel más bajo es un enunciado, que es comparable a una instrucción en el lenguaje FORTRAN. Los enunciados se agrupan en frases y las frases (sentencias) se agrupan en párrafos. Uno o más párrafos pueden utilizarse en conjunto para formar un módulo. Los siguientes niveles superiores son las secciones y las divisiones.

Los programas COBOL consisten en cuatro divisiones:

1. • División de identificación: Contiene el nombre del programa, el del programador, la fecha en que fue escrito y otra información que puede ayudar a identificar el programa y sus objetivos.
2. • División de entorno: Denomina el sistema de computo usado para compilar y ejecutar el programa; identifica los dispositivos que se utilizarán para la entrada y la salida, o bien para conjuntos de datos específicos o resultados necesarios.
3. • División de datos: Describe cuidadosamente los nombres, tipo, extensión y ubicación de los datos, y los resultados que se considerarán como entrada y salida; muestra también las relaciones entre los conjuntos de datos.
4. • División de procedimientos: contiene las frases (es decir, las instrucciones) que especifican las operaciones aritméticas y de proceso así como el orden en el que se llevarán a cabo.

El contenido de las divisiones de identificación y de entorno, se explican por sí mismas. La sección de archivo define un conjunto de datos que se introducen y los resultados que constituyen la salida (es decir, los archivos de los datos). Se utilizan membretes o etiquetas de archivo para identificar cintas y discos; ya que no se utilizan dispositivos de almacenamiento) secundario para la entrada /salida, se omiten las etiquetas de los registros de los archivos.
Los datos son ingresados y egresados en grupos llamados registros. Cada elemento de los datos que es una parte de un registro (record) (tiempo de trabajo, tasa salarial, etc.) debe darse a conocer al sistema, exactamente como se hizo en FORTRAN, por medio del enunciado FORMAT.
En COBOL esto se logra listando el nombre del elemento de datos (adviértase cómo las restricciones en la longitud de los nombres no son tan estrictas como en el FORTRAN, es posible utilizar nombres de variables que se parezcan más a las palabras reales del inglés o del español) y la descripción o especificaciones de tipo y longitud.

Un programa fuente Cobol es un conjunto de instrucciones, párrafos y secciones que se agrupan en cuatro DIVISIONES obligatorias que, escritas en orden, son las siguientes:

IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.

IDENTIFICATION DIVISION (División de identificación)

Tiene que ser incluida en cada programa fuente. Sirve para proporcionar un nombre para identificar el programa. Opcionalmente se puede especificar información acerca del autor, fecha en que fue escrito, etc.
Consta de siete posibles párrafos siendo obligatorio solamente el primero de ellos. Su formato general es el siguiente:

IDENTIFICATION DIVISION.
PROGRAM-ID. Nombre del programa.
[AUTHOR. Nombre programador.]
[INSTALLATION. Instalación.]
[DATE-WRITTEN. Fecha de escritura.]
[DATE-COMPILED. Fecha de compilación.]
[SECURITY. Comentario.]
[REMARKS. Comentario.]

PROGRAM-ID. Nombre del programa. Es el único párrafo obligatorio y sirve para especificar el nombre del programa. Este nombre será usado por el compilador o en la ejecución del programa para indicar algún error.
El nombre del programa debe ajustarse a las reglas, ya vistas, de formación de un identificador Cobol.

ENVIR0NMENT DIVISION (División de entorno)

Permite definir el tipo de ordenador para el que fue escrito el programa, así como los dispositivos periféricos necesarios para soportar los ficheros utilizados en el programa. El formato general para esta división es el siguiente:

```
ENVIRONMENT DIVISION.
[CONFIGURATION SECTION.
[SOURCE-COMPUTER. Nombre ordenador.]
[OBJECT-COMPUTER. Nombre ordenador.]
[SPECIAL-NAMES. Nombre especiales.] ]
[INPUT-OUTPUT SECTION.
FILE-CONTROL.
        Control de archivos.
[I-O CONTROL.
         Control de entrada/salida.] ]
```

CONFIGURATION SECTION (Sección de Configuración)
Esta sección es opcional. Su utilidad es indicar al programa el modelo de ordenador a utilizar y asociar nombres especiales que van a ser usados en el programa.
Para esta última opción se utiliza el párrafo SPECIAL-NAMES siendo interesante la posibilidad de intercambiar la función de la coma y el punto decimal, mediante la cláusula DECIMAL-POINT IS COMMA de la siguiente forma:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
        DECIMAL-POINT IS COMMA.
```

INPUT-OUTPUT SECTION (Sección de entrada/salida)
Esta sección es también opcional. Proporciona información al compilador sobre los archivos utilizados en el programa y su relación con los dispositivos externos. Se explicará con detalle a la hora de trabajar con ficheros.

DATA DIVISION (División de datos)

Se utiliza para realizar una descripción completa de los ficheros que intervienen en el programa, de sus registros lógicos y de las variables de trabajo. Su formato es el siguiente:

```
DATA DIVISION.
[FILE SECTION.
[Declaración del archivo.
[Declaración del registro. ] ] …]
[WORKING-STORAGE SECTION.
[77 Declaración de variables independientes.]
[01 Declaración de registros.] ]
[LINKAGE SECTION.
[77 Declaración de variables independientes.]
[01 Declaración de registros.] ]
[COMMUNICATION SECTION.
[Descripción comunicación.]
```

[Declaración de registros.]]
[SCREEN SECTION.
[01 Descripción de pantallas.]]

FILE SECTION (Sección de Ficheros)
En esta sección se describen detalladamente toda la información referente a los archivos utilizados en el programa, así como los registros junto con sus campos y el tipo de datos que se va a almacenar en ellos. Se verá con más detalle en los apuntes de ficheros.

WORKING-STORAGE SECTION (Sección de Trabajo)
En ella se describen las variables usadas por el programa, ya sea con estructura de registro o como campos independientes.
Para declarar una variable es necesario especificar su número de nivel, su nombre, su tipo y longitud, opcionalmente también se le puede asignar un valor inicial.

2.2.4 PASCAL

Como se ha prestado cada vez más atención a la forma como se diseñan y estructuran los programas de aplicación, han surgido nuevos lenguajes que apoyan mejor el diseño estructurado. Uno de los más conocidos y más ampliamente utilizados es el llamado Pascal. Este lenguaje, que se aplica tanto en pequeñas microcomputadoras como en los sistemas de maxicomputadoras, fue introducido en 1971 por Niklaus Wirth, investigador suizo del Instituto Federal de Tecnología de Zurich. Nombró a este lenguaje en honor de Blas Pascal, el famoso matemático del siglo XVII que desarrolló la primera máquina de calcular.
Una característica importante del Pascal es la de que el programador tiene que planear el desarrollo. Por ejemplo, todas las variables y constantes se definen al principio del programa. (Otros lenguajes permiten que las variables sean introducidas en el citado programa en cualquier momento) El Pascal utiliza puntos y comas para la terminación o la separación de los enunciados (excepto para los de salida que finalizan con un punto). Los comentarios y las observaciones indican por asteriscos o se encierran entre paréntesis.
Una característica inherente de este lenguaje es su tendencia a forzar la estructuración de la lógica. Los programas constan de bloques de instrucciones que empiezan con BEGIN y acaban con END. Todo el programa se trata como si fuera un gran bloque que puede contener otros bloques envueltos o anidados.
Se ejecuta para cada registro (record) de datos en el archivo de entrada, es decir, hasta que se encuentre una marca de fin de archivo (EOF, de end-of-file) en el curso de la entrada. Todos los resultados se escriben completos con la instrucción (WRITELN), la cual contiene los nombres de las variables y los formatos de salida.
El objetivo del Pascal es semejarse en forma más precisa que otros lenguajes a la manera en que los humanos plantean los problemas. El curso lógico de las instrucciones, los diferentes bloques de funciones y de pasos y la capacidad para valerse de variables de cualquier longitud demuestran lo anterior.

Expresiones:

Las expresiones en Pascal son limitadas. No tiene expresiones condicionales de mayor tipo ni bloques de expresiones. No tiene expresiones no triviales de tipo compartido debido a que los agregados y funciones no tienen resultado de tipo compuesto.

Comandos y secuenciadores:

Comandos:

* Ejecución Condicional : { if, case }
* Ejecución Interacción
Indefinido : { repeat, while }
Definido : { for }
Estos comandos pueden ser compuestos libremente
Utilizar el secuenciador "goto". Pascal restringe el uso. Los programadores pueden usar el secuenciador "goto" de forma disciplinada

Declaraciones:

En Pascal se puede declarar :

Constantes:

```
Const
y = 20;
```

Tipos:

```
Type
Array = char [´A´...´Z´]of letras;
```

Variables :

```
Var
a,b : Integer;
c :Real;
```

Procedimientos :

```
Procedure arreglos(var x : integer);
```

Funciones :

```
Function mas(x : real) : real;
```

Pascal, soporta procedimientos y abstracción de funciones. Los procedimientos y funciones usan valores de primera clase.

2.2.5. ADA

El lenguaje más reciente orientado a utilización universal es el Ada, nombrado así en honor de la primera programadora de computación (por su trabajo con el matemático inglés Charles Babbage), Ada Augusta Byron, hija del poeta Lord Byron. La iniciativa para el desarrollo de este lenguaje nació en la Secretaría de la Defensa del gobierno de

Estados Unidos cuando intentó tener un lenguaje de programación que pudiera ser utilizado en la administración o las empresas, en la ciencia, las matemáticas y en las actividades de ingeniería. El hecho de tener un lenguaje que pudiera emplearse en muchas máquinas diversas (y, por lo tanto, haciendo a los programas de cómputo transportables de un sistema a otro) ahorraría millones de dólares cada año.
Este lenguaje, que apenas está adquiriendo una forma útil, reúne las mejores características del Pascal. También se le puede acoplar una variedad de otros componentes programáticos, según las necesidades del usuario. De esta forma, el Ada utiliza el concepto de programática "conectable" tal como los expertos en equipo de cómputo lo han tenido durante tantos años.
Se oirá hablar más acerca del Ada en los próximos años a medida que el lenguaje se utilice en muchos campos y funciones de gobierno y de la administración, desde aplicaciones empresariales hasta programas para la tecnología aeronáutica y astronáutica.

Valores, tipos y subtipos:

La elección de los tipos primitivos y compuestos es similar a Pascal.
No soporta tipos recursivos directamente, ellos tienen que ser programados usando punteros.
Soporta el concepto de subtipo, haciendo una importante distinción entre tipo y subtipo: el subtipo no es el mismo tipo.
Cada declaración de tipo crea un nuevo tipo y distinto.

Enteros -3, -2,...,2, 3, 4, 5
Subtipo {0,.., 5}
Cada valor pertenece solamente a un tipo.

Expresiones:

En ADA se puede escribir expresiones no triviales de cada tipo. En particular los agregados permiten registros y arrays para ser construidos de sus componentes. ADA no tiene expresiones condicionales o bloques de expresiones y el cuerpo de una función es un comando.

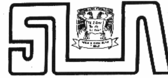
Comandos y secuenciadores:

ADA es igual a Pascal en comandos, los secuenciadores que utiliza Ada: exit y return permiten estructuras de control, además Ada hace uso del "goto".

Declaraciones:

ADA permite declarar:

-Tipos.
-Variables.
-Subtipos.
-Procedimientos.
-Constantes.
-Funciones.
*Paquetes (Packages).



*Tareas (Tasks).
*Excepciones (Exception).

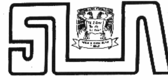
Las declaraciones pueden ser localizadas en bloques de comandos, las cuales permiten que el programador restringa el ámbito de una declaración.

2.3. Selección de los lenguajes de programación

En vista de que los centros de cómputo generalmente tienen dispositivos compiladores y traductores para varios lenguajes diferentes de programación, los programadores y otros miembros del personal técnico de sistemas deben seleccionar uno de ellos para la codificación de una aplicación. (En general, los usuarios no intervienen en este proceso de selección.) Se resumirán brevemente los factores que deben considerarse en la elección de un lenguaje de programación.
La consideración fundamental (suponiendo que los programadores ya conocen los lenguajes) es la capacidad de cada lenguaje. ¿Qué tipos de necesidades de procesamiento se pueden programar con un lenguaje en particular? Después de que se conozcan las características esenciales para satisfacer la necesidad de procesamiento o manejo de datos, se deben evaluar los lenguajes en términos de:

- Especificación de entrada / salida: ¿El lenguaje respalda varios dispositivos (por ejemplo, terminal impresora, rastreador óptico, pantalla catódica)? ¿Qué tan fácil es ejecutar un volumen elevado de procesamientos con elementos de entrada / salida?
- Capacidades de manejo de datos: ¿Qué tipo de datos pueden ser procesados? ¿Cuáles son sus características de producir con nuevo formato y editado (o edición)? ¿Qué clases de conjuntos de caracteres se pueden emplear en el proceso?
- Capacidades de almacenamiento de datos: ¿Qué clases de organización de archivos (que se estudian en el siguiente capítulo) se pueden utilizar? ¿Cuál es su capacidad para almacenar y recabar registros de longitud variable? ¿Es fácil que los conjuntos de datos almacenados puedan ser creados o recabados?
- Ayudas para la programación ¿Qué tan fácil es su uso? ¿Cuáles son sus capacidades de documentación? ¿Cuáles son sus ayudas para la depuración o eliminación de los errores?
- Eficiencia: ¿Es eficiente el proceso de compilación? ¿Es eficaz el código generado durante la compilación? ¿Existe apoyo o soporte para la modificación de las características del lenguaje?
- Arquitectura del compilador
- Número de llamadas a los periféricos

Saber algo sobre las características de los lenguajes puede eliminar con rapidez algunas de ellas de la consideración para algunos tipos de procesamiento. El FORTRAN, por ejemplo, no tiene características intrínsecas para la comunicación de datos. El RPG no es adecuado para ejecutar grandes volúmenes de cálculo puro. El COBOL es conveniente para procesar grandes cantidades de datos cuando el volumen de operaciones de cálculo puro es muy bajo.



La facilidad de programación es una preocupación constante en la elección de lenguaje. Si un lenguaje es fácil de entender y aplicar, puede reducir los errores y acortar el tiempo necesario para habilitar el programa. Se tiene que considerar la facilidad de programación así como la eficiencia de la operación. Los lenguajes de alto nivel son mucho más fáciles de utilizar que el lenguaje ensamblador, pero la codificación no es tan eficiente.

El COBOL, por ejemplo, es fácil de aprender y usar, debido a que los comandos están redactados según la estructura del idioma inglés, y a la gran cantidad de instrucciones. Sin embargo, por tal volumen de instrucciones, resulta mucho menos eficaz en la ejecución que un lenguaje de ensamble. El FORTRAN es de uso mucho más eficaz (para codificar) y funciona con mayor rapidez que el COBOL, pero resulta más compleja la manipulación de datos alfabéticos.

La compatibilidad con los lenguajes que se usan en otros programas de aplicaciones puede ser también un factor decisivo en la elección. Quizá dos o más aplicaciones se combinen en un futuro próximo, por lo que se debe considerar esta posibilidad. La elección del lenguaje programático correcto es un paso muy importante en el ciclo de programación; puede facilitar la operación de la codificación para el usuario y para la organización donde se utiliza.

ACTIVIDADES COMPLEMENTARIAS

1. A partir del estudio de la bibliografía específica sugerida, elabore un mapa conceptual de los temas de la unidad.

Primer nivel

Programación imperativa

Analizar su funcionamiento

Seleccionar un lenguaje adecuado

Segundo nivel

Características

Lenguajes Imperativos

Selección de Lenguajes

- *Especificación de entrada/salida*
- *Capacidades de manejo de datos*
- *Capacidades de almacenamiento de datos*
- *Ayudas para la programación*
- *Eficiencia*
- *Arquitectura del compilador*
- *Número de llamadas a los periféricos*

Se orientan a enunciados (instrucciones)

Tercer nivel

FORTRAN **COBOL** **PASCAL**

Cuarto nivel

JAVA **ADA** **BASIC** **PERL**

Quinto nivel

PHP **XML** **VISUAL** **DBASE...**

2. Investigue cómo se utilizaría la operación de suma, resta o multiplicación, así como división y exponenciación de los números 2*3*5 en los lenguajes BASIC, COBOL, PASCAL, UNIX Y DBASE.

BASIC:

```
DEFINT A-Z
A = 2 + 3
B = 5 – 3
C = 3 / 2
D = 2 * 5
E = 2 ** 3
```

COBOL:

```
77 VAR-A  PIC 9(5) COMP.
77 VAR-B  PIC 9(5) COMP.
77 VAR-C  PIC 9(5) COMP.
77 VAR-D  PIC 9(5) COMP.
77 VAR-E  PIC 9(5) COMP.
ADD 2 TO 3 GIVING VAR-A.
SUBTRACT 3 FROM 5 GIVING VAR-B.
COMPUTE VAR-C = 3 / 2.
MULTIPLY 2 BY 5 GIVING VAR-D.
COMPUTE VAR-E = 2 ** 3.
```

PASCAL:

```
VAR A,B,C,D,E: Integer;
A:= 2 + 3;
B:= 5 - 3;
C:= 3 / 2;
D:= 2 * 5;
E:= 2 ^ 3;
```

Unix (Shell):

```
A = expr `2 + 3`
B = expr `5 – 3`
C = expr `3 \ / 2`
D = expr `2 * 5`
E = expr `2 ** 3`
```

DBASE:

```
DEFINT A-Z
```

```
A = 2 + 3
B = 5 – 3
C = 3 / 2
D = 2 * 5
E = 2 ** 3
```

3. Investigue cómo se puede diferenciar la ejecución de un comando en PASCAL y DBASE.

PASCAL:

```
System("cls");
System("del *.log");
System("otroprog.exe");
```

DBASE:

```
Control = shell("cls",1)
Control = shell("c:\appl\windows\calc.exe",1)
Control = shell("del *.log",0)
DoCmd "select nombre from empleados"
DoCmd "insert into facturas values (2.5, 8, 10.50)"
```

4. Investigue cómo se pueden aceptar datos desde el teclado en BASIC, COBOL, PASCAL y DBASE.

BASIC:

```
DIM ENTRADA$ AS STRING
PRINT "DAME UN VALOR DE ENTRADA"
INPUT$ (ENTRADA$)
```

COBOL:

```
01 ENTRADA  PIC X(50).
DISPLAY "DAME UN VALOR DE ENTRADA"
ACCEPT ENTRADA.
```

PASCAL:

```
VAR ENTRADA: CHAR;
WRITELN("DAME UN VALOR DE ENTRADA");
READLN(ENTRADA);
```

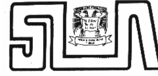
DBASE:

```
DIM ENTRADA AS STRING
```

```
PANTALLA.CAPTION = “DAME UN VALOR DE ENTRADA”)
PANTALLA.SHOW
```

Bibliografía:

- *“Lenguajes de programación”*
  Diseño e implementación
  Pratt – Zelkowitz
  Ed.Prentice may
- *“Sistemas de Información para la Administración”*
  James A. Senn
  Ed. Grupo Editorial Iberoamérica,
  México, 1990
- *“Algorítmica”*
  (Diseño y análisis de algoritmos Funcionales e Imperativos)
  Javier Calve, Juan C. González
  Ed. Addison-Wesley Iberoamericana
  México, 1993



III. Programación Orientada a Objetos.

3.1. Definición.

El Análisis Orientado a Objetos (AOO) se basa en conceptos sencillos, conocidos desde la infancia y que aplicamos continuamente: objetos y atributos, el todo y las partes, clases y miembros. La idea principal de la programación orientada a objetos es construir programas que utilizan objetos de software. Un objeto puede considerarse como una entidad independientemente de cómputo con sus propios datos y programación. Se incluirán datos (llamados campos), que describen sus atributos físicos y programación (llamados métodos), que gobierna la amnera enq ue funciona internamente y en que interactúa con otras partes relacionadas (también objetos). En la programación orientada a objetos, los objetos de software tienen una correspondiencia estrecha con los objetos reales relacionados con el área de la aplkicación. Esta correspondencia facilita la comprensión y el manejo del programa de la computadora. La programación orientada a objetos se aplica en muchos ejemplos realistas y los más grandes se ilustran con diagramas de diseño orientado a objetos.

El paradigma orientado a objetos ha sufrido una evolución similar al paradigma de programación estructurada: primero se empezaron a utilizar los lenguajes de programación estructurados, que permiten la descomposición modular de los programas; esto condujo a la adopción de técnicas de diseño estructuradas y de ahí se paso al análisis estructurado.

Son tres las características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo. El software orientado a objetos permitirá que los objetos independientes se puedan ejecutar en forma simultánea, en procesadores independientes. El diseño orientado a objetos, con herramientas CASE y generadores de código, es la clave para la construcción de poderosas máquinas paralelas. La revolución industrial del software ganará fuerza cuando se difundan las técnicas orientadas a objetos y se disponga de grandes bibliotecas de clases de objetos. Las bibliotecas se enlazarán con depósitos CASE de modo que las nuevas clases se puedan ensamblar rápidamente a partir de las ya existentes.

En el Análisis Orientado a Objetos, los objetos encapsulan tanto atributos como procedimientos (operaciones que se realizan sobre los objetos), e incorpora además conceptos como el polimorfismo o la herencia que facilitan la reutilización de código. El uso de Análisis Orientado a Objetos puede facilitar mucho la creación de prototipos, y las técnicas de desarrollo evolutivo de software. Los objetos son inherentemente reutilizables, y se puede crear un catálogo de objetos que podemos usar en sucesivas aplicaciones. De esta forma, podemos obtener rápidamente un prototipo del sistema, que pueda ser evaluado por el cliente, a partir de objetos analizados, diseñados e implementados en aplicaciones anteriores. Y lo que es más importante, dada la facilidad de reutilización de estos objetos, el prototipo puede ir evolucionando hacia convertirse en el sistema final, según vamos refinando los objetos de acuerdo a un proceso de especificación incremental.

El Análisis Orientado a Objetos ofrece un enfoque nuevo para el análisis de requisitos de sistemas software. En lugar de considerar el software desde una perspectiva clásica de entrada/proceso/salida, como los métodos estructurados clásicos, se basa en modelar el sistema mediante los objetos que forman parte de él y las relaciones estáticas (herencia y composición) o dinámicas (uso) entre estos objetos. Este enfoque pretende conseguir modelos que se ajusten mejor al problema real, a partir del conocimiento del llamado ***dominio del problema***[1] , evitando que influyan en el análisis consideraciones de que estamos analizando un sistema para

[1] *El término dominio del problema o dominio de aplicación es uno de los más usados en el paradigma orientado a objetos. Se refiere al campo de aplicación del sistema, es decir, a qué es el sistema, entendido desde su propio campo de aplicación, más que a su descripción en términos de una implementación en ordenador.*

implementarlo en un ordenador. Desde este punto de vista, el Análisis Orientado a Objetos consigue una abstracción mayor que el análisis estructurado, que modela los sistemas desde un punto de vista más próximo a su implementación en una computadora (entrada/proceso/salida).

Este intento de conocer el *dominio del problema* ha sido siempre importante; no tiene sentido empezar a escribir los requisitos funcionales de un sistema de control de tráfico aéreo, y menos aún diseñarlo o programarlo sin estudiar primero qué es el tráfico aéreo o qué se espera de un sistema de control de este tipo.

El concepto renovador de la tecnología de programación orientada a objetos es la anexión de procedimientos de programas a elementos de datos. A esta nueva unión se le llama encapsulamiento y el resultado es un objeto de software. La programación orientada a objetos ha mejorado de manera esencial la creación del software, pero por sí misma no implica el mejoramiento masivo necesario para la industria de la computación. Las técnicas orientadas a objetos deben combinarse con todos los aspectos disponibles de la automatización del software.

Las técnicas orientadas a objetos modificarán:

- Toda la industria del software;
- La forma en que se venden los programas de aplicación;
- La forma de uso de las computadoras;
- La forma de uso de las redes;
- La forma de analizar los sistemas;
- La forma de diseñar sistemas;
- La forma de utilizar herramientas CASE;
- La forma de planear las empresas;
- El trabajo de todos los profesionales de los sistemas de información.

3.1.1. Ventajas del Análisis Orientado a Objetos.

- ***Dominio del problema.-*** El paradigma Orientado a Objetos es más que una forma de programar. Es una forma de pensar acerca de un problema en términos del mundo real en vez de en términos de un ordenador. El Análisis Orientado a Objetos permite analizar mejor el dominio del problema, sin pensar en términos de implementar el sistema en un ordenador. El Análisis Orientado a Objetos permite pasar directamente el dominio del problema al modelo del sistema.
- ***Comunicación.-*** El concepto Orientado a Objetos es más simple y está menos relacionado con la informática que el concepto de flujo de datos. Esto permite una mejor comunicación entre el analista y el experto en el dominio del problema (es decir, el cliente).
- ***Consistencia.-*** Los objetos encapsulan tanto atributos como operaciones. Debido a esto, el Análisis Orientado a Objetos reduce la distancia entre el punto de vista de los datos y el punto de vista del proceso, dejando menos lugar a inconsistencias o disparidades entre ambos modelos.
- ***Expresión de características comunes.-*** El paradigma Orientado a Objetos utiliza la herencia para expresar explícitamente las características comunes de una serie de objetos. Estas características comunes quedan escondidas en otros enfoques y llevan a duplicar entidades en el análisis y código en los programas. Sin embargo, el paradigma Orientado a Objetos pone especial énfasis en la reutilización, y proporciona mecanismos efectivos que permiten reutilizar aquello que es común, sin impedir por ello describir las diferencias.
- ***Resistencia al cambio.-*** Los cambios en los requisitos afectan notablemente a la funcionalidad de un sistema, por lo que afectan mucho al software desarrollado con métodos estructurados. Sin embargo, los cambios afectan en mucha menor medida a los objetos que componen o maneja el sistema, que son mucho más estables. Las modificaciones necesarias para adaptar una aplicación basada en objetos a un cambio de requisitos suelen estar mucho más localizadas.

- ***Reutilización.***- Aparte de la reutilización interna, basada en la expresión explícita de características comunes, el paradigma Orientado a Objetos desarrolla modelos mucho más próximos al mundo real, con lo que aumentan las posibilidades de reutilización. Es probable que en futuras aplicaciones nos encontremos con objetos iguales o similares a los de la actual.

La ventaja del Análisis Orientado a Objetos es que se basa en la utilización de objetos como abstracciones del mundo real. Esto nos permite centrarnos en los aspectos significativos del dominio del problema (en las características de los objetos y las relaciones que se establecen entre ellos) y este conocimiento se convierte en la parte fundamental del análisis del sistema software, que será luego utilizado en el diseño y la implementación.

3.1.2. Ventajas de la Programación Orientada a Objetos.

- ***Simplicidad:*** como los objetos de software son modelos de objetos reales en el dominio de la aplicación, la complejidad del programa se reduce y su estructura se vuelve clara y simple.
- ***Modularidad:*** cada objeto forma una entidad separada cuyo funcionamiento interno está desacoplado de otras partes del sistema.
- ***Facilidad para hacer modificaciones:*** es sencillo hacer cambios menores en la representación de los datos o los procedimientos utilizados en un programa orientado a objetos. Las modificaciones hechas en el interior de un objeto no afectan ninguna otra parte del programa, siempre y cuando se preserve su comportamiento externo.
- ***Posibilidad de extenderlo:*** la adición de nuevas funciones o la respuesta a ambientes operativos cambiantes puede lograse con sólo introducir algunos objetos nuevos y variar algunos existentes.
- ***Flexibilidad:*** un programa orientado a objetos puede ser muy manejable al adpatrse a diferentes situaciones, pòrque es posible cambiar los patrones de interacción entre los objetos sin alterarlos.
- ***Facilidad para darle mantenimiento:*** los objetos pueden mantenerse por separado, lo que facilita la localización y el arreglo de problemas,a sí como la adición de otros elementos.
- ***Reusabilidad:*** los objetos pueden emplearse en diferentes programas. Es posible construir programas a partir de componenetes prefabricados y preprobados en una fracción del tiempo requerido para elaborar nuevos programas desde el principio.

3.1.3. Conceptos básicos.

Las técnicas orientadas a objetos se basan en organizar el software como una colección de objetos discretos que incorporan tanto estructuras de datos como comportamiento. Esto contrasta con la programación convencional, en la que las estructuras de datos y el comportamiento estaban escasamente relacionadas.

Las características principales del enfoque orientado a objetos son, en primer lugar:

- ***Identidad.-*** Los datos se organizan en entidades discretas y distinguibles llamadas objetos. Estos objetos pueden ser concretos o abstractos, pero cada objeto tiene su propia identidad. Dicho de otra forma: dos objetos son distintos incluso aún en el caso de que los valores de todos sus atributos (p. ej. nombre y tamaño) coincidan. Dos manzanas pueden ser totalmente idénticas pero no por eso pierden su identidad: nos podemos comer una u otra.
- ***Clasificación.-*** Los objetos que tengan los mismos atributos y comportamiento se agrupan en clases. Todas las manzanas tienen una serie de atributos comunes: tamaño, peso, grado de maduración, y un comportamiento común: podemos coger una manzana, moverla o comerla. Los valores de los atributos podrán ser distintos para cada una de ellas, pero todas comparten los mismos atributos y comportamiento (las operaciones que se pueden realizar

sobre ellas). Una clase es una abstracción que describe propiedades (atributos y comportamiento) relevantes para una aplicación determinada, ignorando el resto. La elección de clases es arbitraria, y depende del dominio del problema.

Según esto, una clase es una abstracción de un conjunto posiblemente infinito de objetos individuales. Cada uno de estos objetos se dice que es una instancia o ejemplar de dicha clase. Cada instancia de una clase tiene sus propios valores para sus atributos, pero comparte el nombre de estos atributos y las operaciones con el resto de instancias de su clase.

- ***Polimorfismo.-*** El polimorfismo permite que una misma operación pueda llevarse a cabo de forma diferente en clases diferentes. Por ejemplo, la operación mover, es distinta para una pieza de ajedrez que para una ficha de parchís, pero ambos objetos pueden ser movidos. Una operación es una acción o transformación que realiza o padece un objeto. La implementación específica de una operación determinada en una clase determinada se denomina método.

  Según lo dicho, una operación es una abstracción de un comportamiento similar (pero no idéntico) en diferentes clases de objetos. La semántica de la operación debe ser la misma para todas las clases. Sin embargo, cada método concreto seguirá unos pasos procedimentales específicos. Una de las ventajas del poliformismo es que se puede hacer una solicitud de una operación sin conocer el método que debe ser llamado. Estos detalles de la implantación quedan ocultos para el usuario; la responsabilidad descansa en el mecanismo de selección de la implantación orientada a objetos. La programación con objetos compatibles con conexión se logra al combinar mecanismo y técnicas de programación orientada a objetos y representa su esencia. Los ingredientes clave son:

  - *Objetos intercambiables:* antes que nada debe ser posible representar una colección de objetos similares que resulten intercambiables bajo ciertas operaciones. Tales objetos, por lo general tienen una relación es-un y pueden organizarse en jerarquías de extensión de clase en Java.
  - *Interfaces públicas uniformes:* los objetos intercambiables deben tener ciertas interfaces públicas idénticas para permitir que el mismo procedimiento polimórficofuncione en todos ellos. Esto se logra manteniendo un conjunto de métodos de interfaz pública con descriptores uniformes en toda la jerarquía de clase.
  - *Parámetros polimórficos:* un método polimórfico que opera en el objeto intercambiable, debe declarar parámetros formales capaces de recibir argumentos de cualquier tipo compatible con conexión. Los parámetros deben ser tipos de referencia a superclase.
  - *Acceso dinámico de operaciones intercambiables:* al conservar descriptores uniformes para métodos de interfaz, el mecanismo de sobreescritura de métodos de Java asegura su invocación correcta al momento de la ejecución.

- ***Herencia.-*** El concepto de herencia se refiere a la compartición de atributos y operaciones basada en una relación jerárquica entre varias clases. Una clase puede definirse de forma general y luego refinarse en sucesivas subclases. Cada clase hereda todas las propiedades (atributos y operaciones) de su superclase y añade sus propiedades particulares.

  La posibilidad de agrupar las propiedades comunes de una serie de clases en una superclase y heredar estas propiedades en cada una de las subclases es lo que permite reducir la repetición de código en el paradigma Orientado a Objetos y es una de sus principales ventajas.

3.2. Modelado.

La Técnica de Modelado de Objetos (OMT, Rumbaugh, 1991) es un procedimiento que se basa en aplicar el enfoque orientado a objetos a todo el proceso de desarrollo de un sistema

software, desde el análisis hasta la implementación. Los métodos de análisis y diseño que propone son independientes del lenguaje de programación que se emplee para la implementación. Incluso esta implementación no tiene que basarse necesariamente en un lenguaje Orientado a Objetos.

La Técnica de Modelado de Objetos es una metodología Orientado a Objetos de desarrollo de software basada en una notación gráfica para representar conceptos Orientado a Objetos. La metodología consiste en construir un modelo del dominio de aplicación y ir añadiendo detalles a este modelo durante la fase de diseño. La Técnica de Modelado de Objetos consta de las siguientes fases o etapas.

3.2.1. Fases.

- ***Conceptualización.*** Consiste en la primera aproximación al problema que se debe resolver. Se realiza una lista inicial de requisitos y se describen los casos de uso.
- ***Análisis.*** El analista construye un modelo del dominio del problema, mostrando sus propiedades más importantes. Los elementos del modelo deben ser conceptos del dominio de aplicación y no conceptos informáticos tales como estructuras de datos. Un buen modelo debe poder ser entendido y criticado por expertos en el dominio del problema que no tengan conocimientos informáticos.
- ***Diseño del sistema.*** El diseñador del sistema toma decisiones de alto nivel sobre la arquitectura del mismo. Durante esta fase el sistema se organiza en subsistemas basándose tanto en la estructura del análisis como en la arquitectura propuesta.
- ***Diseño de objetos.*** El diseñador de objetos construye un modelo de diseño basándose en el modelo de análisis, pero incorporando detalles de implementación. El diseño de objetos se centra en las estructuras de datos y algoritmos que son necesarios para implementar cada clase. OMT describe la forma en que el diseño puede ser implementado en distintos lenguajes (orientados y no orientados a objetos, bases de datos, etc.).
- ***Implementación.*** Las clases de objetos y relaciones desarrolladas durante el análisis de objetos se traducen finalmente a una implementación concreta. Durante la fase de implementación es importante tener en cuenta los principios de la ingeniería del software de forma que la correspondencia con el diseño sea directa y el sistema implementado sea flexible y extensible. No tiene sentido que utilicemos Análisis Orientado a Objetos y diseño Orientado a Objetos de forma que potenciemos la reutilización de código y la correspondencia entre el dominio del problema y el sistema informático, si luego perdemos todas estas ventajas con una implementación de mala calidad.

Algunas clases que aparecen en el sistema final no son parte del análisis sino que se introducen durante el diseño o la implementación. Este es el caso de estructuras como árboles, listas enlazadas o tablas hash, que no suelen estar presentes en el dominio de aplicación. Estas clases se añaden para permitir utilizar determinados algoritmos.

Los conceptos del paradigma Orientado a Objetos pueden aplicarse durante todo el ciclo de desarrollo del software, desde el análisis a la implementación sin cambios de notación, sólo añadiendo progresivamente detalles al modelo inicial.

3.2.2. Modelos.

Al igual que los métodos estructurados, la Técnica de Modelado de Objetos utiliza tres tipos de modelos para describir un sistema:

- ***Modelo de objetos.*** Describe la estructura estática de los objetos de un sistema y sus relaciones. El modelo de objetos contiene diagramas de objetos. Este modelo muestra la estructura estática de los datos del mundo real y las relaciones entre estos datos. El modelo de objetos precede normalmente al dinámico y al funcional porque normalmente está mejor definido en la especificación preliminar, es menos dependiente de detalles de la

aplicación, es más estable respecto a la evolución de la solución y es más fácil de entender que el resto. Un diagrama de objetos es un grafo cuyos nodos son clases y cuyos arcos son relaciones entre clases.

- ***Modelo dinámico.*** El modelo dinámico describe las características de un sistema que cambia a lo largo del tiempo. Se utiliza para especificar los aspectos de control de un sistema. Para representarlo utilizaremos DEs.
- ***Modelo funcional.*** Describe las transformaciones de datos del sistema. El modelo funcional contiene DFDs y especificaciones de proceso.

Los tres modelos son vistas ortogonales (independientes) del mismo sistema, aunque existen relaciones entre ellos. Cada modelo contiene referencias a elementos de los otros dos. Por ejemplo, las operaciones que se asocian a los objetos del modelo de objetos figuran también, de forma más detallada en el modelo funcional. El más importante de los tres es el modelo de objetos, porque es necesario describir qué cambia antes que decir cuándo o cómo cambia.

Modelo de objetos.

Los cambios y las transformaciones no tienen sentido a menos que haya algo que cambiar o transformar. La Técnica de Modelado de Objetos considera este modelo el más importante de los tres. El modelo de objetos se representa gráficamente con diagramas de objetos y diagramas de instancias, que contienen clases de objetos e instancias, respectivamente. Las clases se disponen en jerarquías que comparten una estructura de datos y un comportamiento comunes, y se relacionan con otras clases. Cada clase define los atributos que contiene cada uno de los objetos o instancias y las operaciones que realizan o sufren estos objetos.

El modelo de objetos puede contener los siguientes elementos:

Instancias. Cada uno de los objetos individuales.

Clases. Abstracción de objetos con propiedades comunes.
- **Atributos.** Datos que caracterizan las instancias de una clase.
- **Operaciones.** Funciones que pueden realizar las instancias

Relaciones. Se establecen entre clases.
- Asociación. Relación de uso en general.
  - **Multiplicidad.** Número de instancias que intervienen en la relación.
  - **Atributos.** Algunos atributos pueden depender de la asociación.
  - **Calificación.** Limita la multiplicidad de las asociaciones
  - **Roles.** Indican los papeles de las clases en las relaciones
  - **Restricciones y ordenación**
- Composición. Relaciones todo/parte.
- Generalización. Relaciones padre/hijo
  - **Redefinición.** Modificación de las propiedades heredadas.

Enlaces. Instancias de una relación. Relaciona instancias.

Los objetos del modelo pueden ser tanto entidades físicas (como personas, casas y máquinas) como conceptos (como órdenes de compra, reservas de asientos o trayectorias). En cualquier caso, todas las clases deben ser significativas en el dominio de aplicación. Hay que evitar definir clases que se refieren a necesidades de implementación como listas, subrutinas o

el reloj del sistema. No todas las clases figuran explícitamente en la especificación preliminar, algunas están implícitas en el dominio de aplicación o son de conocimiento general.

Podemos empezar haciendo una lista de clases candidatas a partir de la especificación preliminar. Normalmente las clases se corresponden con nombres en este documento. No hay que preocuparse aún de establecer relaciones de generalización/especialización entre las clases. Esto se hace para reutilizar código y estas relaciones aparecen más claramente cuando se definan atributos y operaciones.

Una vez que tenemos una lista de clases candidatas hay que proceder a revisarla, eliminando las incorrectas, siguiendo los siguientes criterios:

- ***Clases redundantes.*** Si dos clases representan la misma información nos quedaremos con la que tenga un nombre más significativa, eliminando la otra. (p. ej. Cliente y Usuario).
- ***Clases irrelevantes.*** Podemos haber incluido clases que sean importantes en el dominio de aplicación, pero que sean irrelevantes en el sistema que pretendemos implementar. Esto depende mucho del problema concreto.
- ***Clases demasiado generales.*** Las clases deben ser específicas. Deben tener unos límites claros y unos atributos y un comportamiento comunes para todas las instancias. Debemos eliminar las clases demasiado generales, normalmente creando otras más específicas. Las clases genéricas se incluirán más adelante si es necesario, al observar atributos u operaciones comunes a varias clases.
- ***Atributos.*** Los nombres que describen propiedades de los objetos son atributos, no clases. Una de las características de un objeto es su identidad (aún cuando los valores de los atributos sean comunes), y otra muy común es su existencia independiente. Los atributos de un objeto no presentan estas dos propiedades. Si la existencia independiente de un atributo es importante, hay que modelarlo como una clase. (P. ej. podemos considerar el Despacho, como un atributo de la clase Empleado, pero esto nos impide hacer una reasignación de despachos, por lo que, si queremos implementar esta operación, deberemos considerar los despachos como objetos en lugar de como atributos).
- ***Operaciones.*** Una clase no puede ser una operación que se aplique sobre los objetos sin tener independencia por sí misma (p. ej. en nuestro modelo de la línea telefónica la Llamada es una operación, no una clase). Sin embargo, esto depende del dominio de aplicación: en una aplicación de facturación telefónica, la Llamada será una clase que contiene atributos tales como Origen, Destino, Fecha, Hora y Número de Pasos.
- ***Roles.*** El nombre de una clase debe depender de su naturaleza intrínseca y no del papel que juegue en el sistema. Podemos agrupar varias clases en una, si intrínsecamente son lo mismo, y luego distinguir estos roles en las asociaciones. (P. ej. La clase Persona puede jugar varios papeles (Propietario, Conductor, etc.) en relación con la clase Vehículo. En otros casos, una misma entidad física puede modelarse mediante varias clases distintas si los atributos y el comportamiento (y especialmente las instancias) son distintos dependiendo del papel que juegue en el sistema.
- ***Objetos internos.*** No debemos incluir en el la fase de análisis clases que no se corresponden con el dominio de aplicación sino que tienen relación con la implementación del sistema.

Una vez identificadas las clases debemos empezar a preparar el diccionario de datos del sistema. Cada clase figurará como una entrada en el diccionario donde se define brevemente su significado y las funciones que realiza. El diccionario de datos también contiene entradas para las asociaciones, las operaciones y los atributos, que deben ser definidos según las vamos incorporando al modelo.

3.3. Relaciones entre objetos (Estáticas y Dinámicas)

El Análisis Orientado a Objetos ofrece un enfoque nuevo para el análisis de requisitos de sistemas software. En lugar de considerar el software desde una perspectiva clásica de entrada/proceso/salida, como los métodos estructurados clásicos, se basa en modelar el sistema mediante los objetos que forman parte de él y las relaciones estáticas (herencia y composición) o dinámicas (uso) entre estos objetos. Este enfoque pretende conseguir modelos que se ajusten mejor al problema real, a partir del conocimiento del llamado dominio del problema, evitando que influyan en el análisis consideraciones de que estamos analizando un sistema para implementarlo en un ordenador. Desde este punto de vista, Análisis Orientado a Objetos consigue una abstracción mayor que el análisis estructurado, que modela los sistemas desde un punto de vista más próximo a su implementación en una computadora (entrada/proceso/salida).

Este intento de conocer el dominio del problema ha sido siempre importante; no tiene sentido empezar a escribir los requisitos funcionales de un sistema de control de tráfico aéreo, y menos aún diseñarlo o programarlo sin estudiar primero qué es el tráfico aéreo o qué se espera de un sistema de control de este tipo.

En programamción, la posibilidad de determinar la clase de un objeto en tiempo de ejecución, y de asignarle almacenamiento, se conoce como ***ligadura dinámica***. La ligadura dinámica también se conoce como *ligadura posterior* y es la técnica de programación que se emplea usualmente para realizar el poliformismo en los lenguajes de programación orientado a objetos.. En los lenguajes limitados estáticamente el compilador asigna el almacenamiento a los objetos, y su tipo únicamente determina su clase. Una clase tiene miembros y operaciones, mientras que un conjunto sólo tiene miembros. La ***ligadura previa*** o ***estática,*** es en la que el compilador lleva a cabo la asignación de tipos.

3.4. Clases, Instancias y Objetos.

Clases.- Una clase o clase de objetos es una abstracción que describe un grupo de instancias con propiedades (atributos) comunes, comportamiento (operaciones) común, relaciones comunes con otros objetos y (lo que es más importante) una semántica común. Así un caballo y un establo tienen los dos un coste y una edad, pero posiblemente pertenezcan a clases diferentes (aunque esto depende del dominio de aplicación: en una aplicación financiera ambos pertenecerían posiblemente a la misma clase: Inversiones). Todas las clases deben ser significativas en el dominio de aplicación. Hay que evitar definir clases que se refieren a necesidades de implementación como listas, subrutinas o el reloj del sistema. No todas las clases figuran explícitamente en la especificación preliminar, algunas están implícitas en el dominio de aplicación o son de conocimiento general. El símbolo gráfico para representar clases es un rectángulo, en el que figura el nombre de la clase. Las clases se representan en los diagramas de clases, que son plantillas que describen un conjunto de posibles diagramas de instancias. Describen, por tanto el caso general.

Podemos empezar haciendo una lista de clases candidatas a partir de la especificación preliminar. Normalmente las clases se corresponden con nombres en este documento. No hay que preocuparse aún de establecer relaciones de generalización/especialización entre las clases. Esto se hace para reutilizar código y estas relaciones aparecen más claramente cuando se definan atributos y operaciones.

La mayoría de las ***instancias***[2] de una clase derivan su individualidad de tener valores diferentes en alguno/s de sus atributos o de tener relaciones con instancias diferentes. No

[2] *instancia*, a pesar de ser el término utilizado habitualmente en la bibliografía en castellano, es una mala traducción del término inglés *instance*, siendo *ejemplar* la traducción correcta. Algunos autores solventan el problema hablando de clases y objetos, pero esto último resulta a veces ambiguo.

obstante pueden existir instancias con los mismos valores de los atributos e idénticas relaciones. El símbolo gráfico para representar instancias es un rectángulo de esquinas redondeadas. Dentro del rectángulo figura la clase a la que pertenece la instancia (entre paréntesis) y los valores de sus atributos.

Las instancias figuran en diagramas de instancias, que se utilizan normalmente para describir ejemplos que aclaren un diagrama de objetos complejos o para describir escenarios determinados (p. ej. situaciones típicas o anómalas, escenarios de prueba, etc.). La diferencia entre instancia y clase está en el grado de abstracción. Un objeto es una abstracción de un objeto del mundo real, pero una clase es una abstracción de un grupo de objetos del mundo real. La abstracción permite la generalización y evita la redefinición de las características (atributos, comportamiento o relaciones) comunes, de forma que se produce una reutilización de estas definiciones comunes por parte de cada uno de los objetos. Por ejemplo todas las elipses (instancias) comparten las mismas operaciones para dibujarlas o calcular su área.

Objetos.- Los objetos del modelo pueden ser tanto entidades físicas (como personas, casas y máquinas) como conceptos (como órdenes de compra, reservas de asientos o trayectorias). En cualquier caso, todas las clases deben ser significativas en el dominio de aplicación.Durante el proceso de desarrollo aparecen tres categorías de objetos:

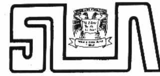
- ***Los objetos del dominio*** son significativos desde el punto de vista del dominio del problema. Existen de forma independiente a la aplicación y tienen sentido para los expertos del dominio.
- ***Los objetos de aplicación*** representan aspectos computacionales de la aplicación que son visibles para los usuarios. No existen en el espacio del problema, solo tienen sentido en el contexto de la aplicación que vamos a desarrollar para solucionarlo. Sin embargo, no dependen exclusivamente de decisiones de diseño, puesto que son visibles al usuario y no pueden ser cambiados sin alterar la especificación de la aplicación. No pueden obtenerse analizando el dominio de la aplicación, pero sí pueden ser reutilizados de aplicaciones anteriores, incluso aunque sean de diferente dominio. Los objetos de aplicación incluyen controladores, dispositivos e interfaces.
- ***Los objetos internos*** son componentes de la aplicación que resultan invisibles para el usuario. Su existencia se deriva de decisiones de diseño para implementar el sistema. No deben aparecer durante el análisis. Una parte importante del diseño consiste en añadir objetos internos para hacer factible la implementación del sistema.

Por tanto, en el modelo de objetos figurarán tanto objetos del dominio como objetos de aplicación. Su construcción puede ser realizada en dos fases:

- En una primera fase podemos construir un modelo que represente los objetos del dominio del problema y las relaciones que existen entre ellos. Este modelo equivale a lo que se llama modelo esencial en la terminología del análisis estructurado.
- En una segunda fase podemos construir un modelo de aplicación, completando el modelo anterior con objetos de aplicación. Para esto jugarán un papel muy importante los casos de uso desarrollados durante la conceptualización del problema.

3.5. Interfaz.

Los métodos abstractos son útiles cuando se quiere que cada implementación de la clase parezca y funcione igual, pero necesita que se cree una nueva clase para utilizar los métodos abstractos. Los interfaces proporcionan un mecanismo para abstraer los métodos a un nivel superior. Un interface contiene una colección de métodos que se implementan en otro lugar.



La razón por la que la aplicación más popular de la programación orientada a objetos hasta los últimos tiempos ha sido el diseño de interfaces de usuario (IU) es, en parte, la amplia influencia de Smalltalk, que hacía hincapié en la interfaz de usario y en las aplicaciones de automatización de oficinas desde los primeros días de Dynabook, y en parte, también, como consecuencia de la complejidad de las interfaces gráficas de usuario típicas (GUI).

Los beneficios de la programación orientada a objetos que son aplicables específicamente en la zona de los Interfaces de Usuario son como siguen:

- La facilidad con la cual es posible utilizar y reutilizar una interfaz congruente entre objetos de "escritorio".
- El hecho consistente en que las interfaces poseen relativamente pocas clases.
- El hecho de que la metáfora del escritorio se corresponde, de forma relativamente sencilla, con la metáfora de los objetos.
- Como consecuencia del establecimiento de la tecnología en este aspecto, existe un suministro abundante de bibliotecas útiles de objetos, y Smalltalk y otros lenguajes incluyen, en la actualidad, algunas de estas características como estándares.
- La mera complejidad de lo GUI modernos hace que la programación orientada a objetos sea una necesidad más que un lujo: si no existiese, entonces, los GUI se verían obligados a inventarla.

3.6. Clase, Estructuras de datos abstractos y Tipos de datos abstractos.

En el análisis orientado a objetos, la estructura es un conjunto enlazado de objetos. Las estructuras son de tres tipos principales:

- De clasificación.- Dícese de una estructura en forma de árbol o de red que está basada eb las primitivas semánticas de inclusión (una clase de) y de pertenencia (IsA) y que indica la herencia puede rrealizar la especificación o la realización. Los objetos pueden participar en más de una de estas estructuras, dando lugar a herencia múltiple.
- De composición.-
- De uso.- Dícese de la estructura de relaciones ecxistente entre clientes y servidores que están conectados mediante el paso de mensajes.

Las asociaciones generales también dan lugar a estructuras.

Un tipo de datos abstractos (abstract data type, ADT) es una abstracción, similar a una clase, que describe un conjunto de objetos en términos de una estructura de datos encapsulada u oculta y las operaciones sobre esa estructura. Los tipos de datos abstractos pueden ser definidos por el usuario al construir una aplicación, en lugar de ser construidos por el diseñador del lenguaje subyacente. Los tipos de datos abstractos incluyen métodos. Por ejemplo un tipo de datos abstractos que represente longitudes expresadas en unidades inglesas incluría métodos para sumar pies y pulgadas.

Los tipos de datos abstractos (abstract data type, ADT) extienden el concepto de los tipos definidos por el usuario (TDU), añadiendo el encapsulado. Los tipos de datos abstractos contienen la representación y operaciones de un tipo de datos, sino que proporciona un muro de protección que impide el uso inadecuado de sus objetos. Toda la interfaz ocurre a través de operaciones definidas dentro del tipo de datos abstractos. Así las operaciones proporcionan un medio bien definido para tener acceso a los objetos de un tipo de datos. Los tipos de datos abstractos san a los objetos una interfaz pública a través de sus operaciones permitidas. Sin embargo, las representaciones y el código ejecutable (el método) de cada operación son privadas.

La capacidad de uso de los tipos de datos abstractos apareció por primera vez con Simula 67. Su implantación se llama *class* (clase). Modula se refiere a su implantación de los tipos de datos abstractos como un *module* (módulo), en tanto que ADA utiliza la palabra *package* (paquete). En todos los casos, los tipos de datos abstractos proporcionan una forma para que los diseñadores de sistemas identifiquen los tipos de datos del mundo real y los empaquen en forma más conveniente y compacta.

Los tipos de datos abstractos se pueden definir para cosas tales como fecha, paneles de la pantalla, pedidos de clientes y solicitudes de partes. Una vez definidos, el diseñador se puede dirigir a los os tipos de datos abstractos en operaciones futuras. El tipo de datos abstractos se define también mediante un conjunto de operaciones permisibles. Estas operaciones proporcionan una especie de armadura que protege a la estructura esencial del uso arbitrario y accidental. Además, cada método o algoritmo de procesamiento utilizado para llevar a cabo una operación queda oculto a los usuarios. Lo que el usuario debe proporcionar es un objeto apropiado para llamar, o solicitar, la operación junto con los parámetros permitidos y aplicables.

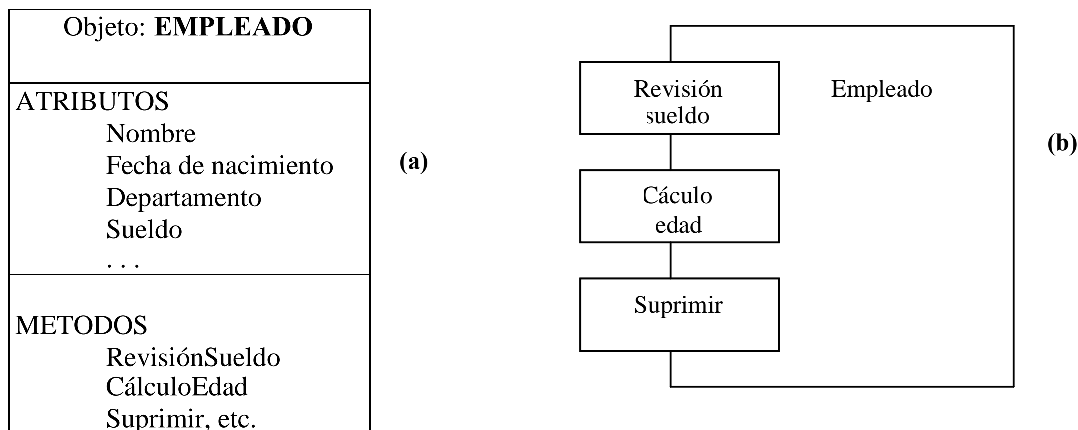


Figura 3.6.1. **(a)** Objeto o tipo de datos abstracto que encierra el concepto de empleado. **(b)** La clase Empleado, en la notación de orientación a objetos más común.

3.7. Clases Concretas, Abstractas, Metaclases, Superclases, Subclases.

Una clase abstracta es una clase que no tiene instancias directas pero cuyas clases descendientes (***clases concretas***) sí pueden tener instancias directas. Una clase concreta puede tener subclases abstractas, pero éstas han de tener subclases concretas (las hojas de una jerarquía de clases han de ser clases concretas).

Las clases abstractas organizan características comunes a varias clases. A veces es útil crear una superclase abstracta que encapsule aquellas características (atributos, operaciones y asociaciones) comunes a un conjunto de clases. Puede ser que esta clase abstracta aparezca en el dominio del problema o bien que se introduzca artificialmente para facilitar la reutilización de código.

Normalmente las clases abstractas se usan para definir métodos que son heredados por sus subclases. Sin embargo, una clase abstracta puede definir el protocolo de una determinada operación sin proporcionar el correspondiente método. Esto de denomina operación abstracta, que define la forma de una operación para la que cada subclase concreta debe suministrar su propia implementación. Una clase concreta no puede tener operaciones abstractas porque los objetos de esta clase tendrían operaciones indefinidas. Para indicar que una operación es abstracta se coloca una restricción {abstracta} junto a ella.

Una de las características más útiles de cualquier lenguaje orientado a objetos es la posibilidad de declarar clases que definen como se utiliza solamente, sin tener que implementar métodos. Esto es muy útil cuando la implementación es específica para cada usuario, pero todos los usuarios tienen que utilizar los mismos métodos. Cuando una clase contiene un método abstracto tiene que declararse abstracta. No obstante, no todos los métodos de una clase abstracta tienen que ser abstractos. Las clases abstractas no pueden tener métodos privados (no se podrían implementar) ni tampoco estáticos. Una clase abstracta tiene que derivarse obligatoriamente, no se puede hacer un new de una clase abstracta.

En lenguajes orientados a objetos (como el SmallTalk), las ***metaclases*** (*metaclass*) son aquellas clases cuyas instancias son clases, por oposición a una clase abastracta, que no posee instancias, sino cólo subclases. Se utiliza de modo impreciso como sinónimo de clase abstracta.

Una ***superclase*** es una clase con uno o más miembros que son clases más especializadas a su vez.. Las ***subclases*** (subclass) se consideran de una clase que posee un enlace AKO[3] con una clase más general, tal como en "un PERRO es una clase de MAMÍFERO".

3.8. Generalización, Especialización Redifinición de clases.

La ***generalización*** es el resultado (o el acto) de distinguir un tipo de objeto como más general, o incluso, que es más que otro. Todo lo que se aplique a un tipo de objeto también se aplica a sus subtipos. Cada instancia de un tipo de objeto es también una instancia de sus supertipos.

Las jerarquías de generalización son importantes para el desarrollador orientado a objetos por dos razones. La primera es que el uso de supertipos y subtipos proporciona una herramienta útil para describir el mundo del sistema de aplicación. La seguridad es que indica las direcciones de herencia entre las clases en los lenguajes de programación orientada a objetos.

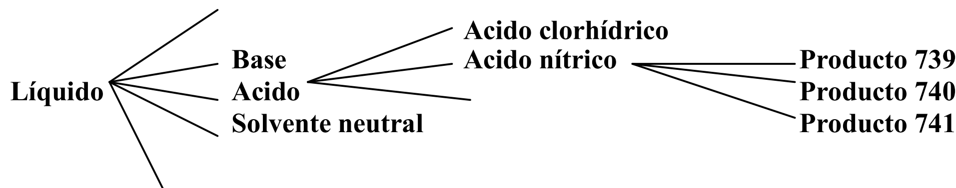


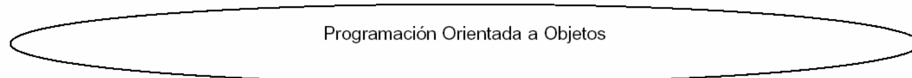
Fig. 3.8.1. Una ***Jerarquía de generalización*** que indica que todas las propiedades de ácido nítrico se aplican a **producto 739, producto 740 y producto 741**, pero cada producto tiene sus propiedades particulares. Del mismo modo, todas las propiedades de ácido se aplican a ***ácido nítrico, ácido clorhídrico***, etcétera.

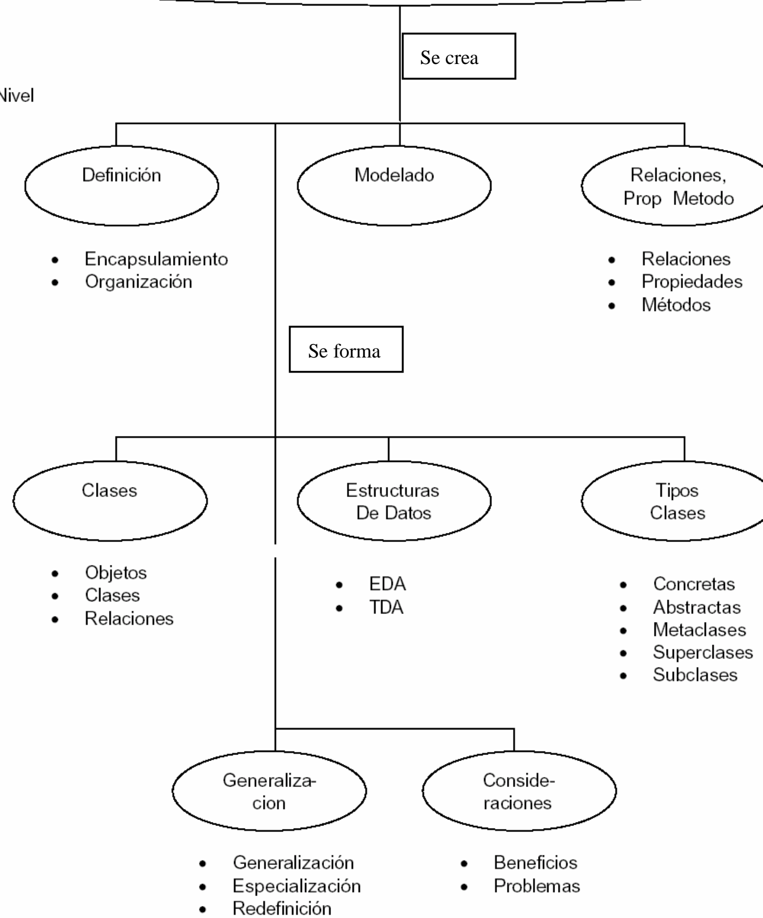
[3] AKO (a kind of) Un-tipo-de se refiere a la relación de herencia que media entre las clases y sus superclases.

Informática IV
Actividades Complementarias

1. A partir del estudio de la bibliografía específica sugerida, elabore un mapa conceptual de los temas de la unidad.

1er Nivel





2. Realice el algoritmo de la raíz cuadrada de un número natural, aplicando la programación orientada a objetos y la orientada a eventos.

HTML:

```
<HTML>
<HEAD>
<TITLE>Ejemplo de cuadro de seleccion (checkbox)</TITLE>
<SCRIPT>
<!--
function Calcula(Formu, Campo) {
   if (Formu.BotonAccion[1].checked) {
      Formu.resultado.value = Math.sqrt(Formu.entrada.value)
   } else {
      Formu.resultado.value = Formu.entrada.value * Formu.entrada.value
   }
}
//-->
</SCRIPT>
</HEAD>

<BODY>
<FORM METHOD=POST>
<P>Valor:<BR>
<INPUT TYPE="text" NAME="entrada" VALUE=0 onChange="Calcula(this.form, this.name);">
<P>Acción:<BR>
<INPUT TYPE="radio" NAME="BotonAccion" VALUE="cuadrado"> Al cuadrado<BR>
<INPUT TYPE="radio" NAME="BotonAccion" VALUE="raiz2"> Raiz cuadrada<BR>
<P>Resultado:<BR>
<INPUT TYPE=text NAME="resultado" VALUE=0>
</FORM>
</BODY>
</HTML>
```

JAVA:

```
<HEAD>

...
<script language="JavaScript">
<!--
```

```
	function Menor(form) {
		var primerOp = parseFloat(form.op1.value);
		var segundoOp = parseFloat(form.op2.value);
		var menor = Math.min(primerOp, segundoOp);
		alert("El menor es el " + menor);
	}

	function Mayor(form) {
		var primerOp = parseFloat(form.op1.value);
		var segundoOp = parseFloat(form.op2.value);
		var mayor = Math.max(primerOp, segundoOp);
		alert("El mayor es el " + mayor);
	}

	function Potencia(form) {
		var base = parseFloat(form.op1.value);
		var exponente = parseFloat(form.op2.value);
		var resultado = Math.pow(base, exponente);
		alert("El resultado es  " + resultado);
	}

	function Cal_sqrt(form) {
		var alfa = parseFloat(form.num.value);
		form.num.value = Math.sqrt(alfa);
	}


//-->
</script>

...
</HEAD>
<BODY>
<form>
<center>
Introduzca un número:
	<input type="text" name="num" size=10><br>
</center>
<table>
<tr><td><input type="button" name="pi" value=" PI "
				onClick="Cal_Pi(this.form)">
<tr><td><input type="button" name="sqrt" value=" SQRT "
				onClick="Cal_sqrt(this.form)">
	<td><input type="button" name="aleator" value=" RANDOM "
				onClick="Cal_aleatorio(this.form)">
```

```
</table>
</center>
<p>
<center>
        Operando 1: <input type="text" name="op1"><br>
        Operando 2: <input type="text" name="op2"><br>
</center>
<p>
<center>
<table>
<tr><td><input type="button" value="Mínimo" onClick="Menor(this.form)">
        <td><input type="button" value="Máximo" onClick="Mayor(this.form)">
        <td><input type="button" value="Potencia" onClick="Potencia(this.form)">
</table>
</center>
</form>
```

SQL:

```
'Crea una consulta compuesta de dos campos, una cantidas y la raiz cuadrada
' de esa cantidad. A estos campos les asigna un nombre alternativo

SELECT Cantidad AS Cant1, SQR(Cantidad) AS Cant2
   FROM Ventas
```

BIBLIOGRAFÍA

- **JAVA, CON PROGRAMACION ORIENTADA A OBJETOS Y APLICACIONES EN LA WWW**
  WANG, PAUL S.
  EDITORIAL INTERNATIONAL THOMSON EDITORES
  PRIMERA EDICION
  MEXICO, 1999

- **ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS**
  MARTIN, JAMES
  ODELL, JAMES J.
  EDITORIAL PRENTICE HALL
  PRIMERA EDICION
  MEXICO, 1994

- **METODOS ORIENTADO A OBJETOS**
  GRAHAM, IAN
  EDITORIAL ADDISON-WESLEY/DIAZ DE SANTOS
  PRIMERA EDICION
  MEXICO, 1996

- **WWW. MONOGRAFIAS.COM**
  APUNTES DE INFORMATICA

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CONTADURIA Y ADMINISTRACION

DIVISION DEL SISTEMA DE UNIVERSIDAD ABIERTA

INFORMATICA IV

UNIDAD 4: PROGRAMACIÓN FUNCIONAL

PROFESOR:

HERNÁNDEZ CASTILLO VICENTE

Informática IV

4. Programación Funcional

Definición

La programación funcional se basa en el uso de las funciones matemáticas para producir mejores efectos y que sus resultados sean más eficientes. La función se define basándose en procedimientos y ésta puede ser recursiva.

Puede ser predefinida por el lenguaje, como en Pascal: la raíz cuadrada de x (square X) o programada en forma funcional, donde los valores son producidos por una función y estos se pasan por unos parámetros a otra función.

Una función es una regla para mapear (o asociar) los miembros de un grupo (grupo dominante) a aquellos de otro (grupo de rango). Una *definición de función,* especifica el dominio, el rango y la regla de mapeo para la función. Una vez que la función ha sido definida, puede ser aplicada a un elemento particular del grupo de dominio: produce la aplicación (o resultados, o regreso) al elemento asociado en el grupo de rango.

Un lenguaje funcional tienen cuatro componentes:

1. Un grupo de funciones primitivas.- Son predefinidas por el lenguaje y pueden ser aplicadas.

2. Un grupo de formas funcionales.- Son mecanismos a través de los cuales las funciones pueden ser combinadas para crear nuevas funciones.

3. La operación de la aplicación.- Es la construcción de un mecanismo para aplicar una función a sus argumentos y producir un valor.

4. Un grupo de datos objetos.- Son los miembros que permiten los grupos de dominio y rango. La característica de los lenguajes funcionales es proveer un grupo muy limitado de datos objetos con una simple y regular estructura. La esencia de la programación funcional es definir nuevas funciones usando las formas funcionales.

En general, la sintaxis de esta clase de lenguajes es similar a:

función (.. función 2(función1(datos))...)

LISP y ML son dos lenguajes funcionales que manejan este modelo. Ambos difieren de otros lenguajes anteriores en dos aspectos importantes:

- Desde el punto de vista de su diseño, están destinados a programarse en forma aplicativa. La ejecución de una expresión es el concepto principal que se necesita para la secuenciación de programas, en vez de los enunciados de lenguajes como C, Pascal o Ada.

- Puesto que son aplicativos (o funcionales), el almacenamiento en montículos y las estructuras de lista se convierten en el proceso natural para gestión de almacenamiento en lugar del mecanismo tradicional de registros de activación de los lenguajes de procedimiento.

Características:

- El valor de una expresión depende sólo de los valores de sus sub-expresiones si las tiene.

- El manejo de almacenamiento es implícito. Ciertas operaciones integradas de los datos asignan almacenamiento en el momento necesario. El almacenamiento que se vuelve inaccesible se libera, en forma automática.

- La programación funcional trata a las funciones como "ciudadanos de primera clase". Las funciones son valores de primera clase. Las funciones tienen la misma jerarquía que cualquier otro valor. Una función puede ser el valor de una expresión, puede pasarse como argumento y puede colocarse en una estructura de datos.

Paradigma funcional

• Aplicación y composición de funciones.

• Ejecutar un programa: **evaluar** una función.

• El orden de evaluación de las funciones **no** depende del programador (con reservas).

• Diferencia clara entre datos de entrada y de salida (una única dirección).

Principales ventajas:

1. Elegancia, claridad, sencillez, potencia y concisión
2. Semánticas claras, simples y matemáticamente bien fundadas
3. Cercanos al nivel de abstracción de las especificaciones formales / informales de los problemas a resolver
4. Referencialmente transparentes: Comportamiento matemático adecuado que permite razonar sobre los programas
5. Soportan técnicas muy avanzadas de desarrollo, mantenimiento y validación de programas
6. Altas dosis de paralelismo implícito
7. Aplicaciones variadas y de gran interés
8. Provee de un paradigma para programar en paralelo.
9. Es ampliamente aplicada en la Inteligencia Artificial.
10. Es bastante útil para el desarrollo de prototipos.

Aplicaciones:

- Interfaces para bases de datos (SQL es “casi” declarativo)
- Ingeniería de lenguaje natural y consulta de bases de datos en lenguaje natural
- Sistemas de planificación y scheduling
- Sistemas de tomas de decisiones basados en el conocimiento
- Sistemas de computación simbólica (Mapple o Mathematica son “casi” declarativos)
- Aplicaciones de aprendizaje y enseñanza
- Descripción de lenguajes y construcción de compiladores
- Verificación de propiedades (demostración de teoremas, verificación de sistemas hardware/software, diagnosis, etc.)

- Enrutado de aviones para Alitalia (Prolog), para Airbone Corps USA (Prolog), asignación de aviones a puertas de embarque - APACHE (CHIP)
- Diseño de circuitos lógicos para protección automática de trenes en la estación de Bolonia.
- Configuración de teléfonos móviles - Nokia (IF-Prolog)
- Configuración de centrales telefónicas – Ericsson (Erlang)
- Acceso a bases de datos en español - Software AG España (Haskell)

4.2. Estructura de los lenguajes de programación funcional

Expresiones y sentencias.

Expresiones:

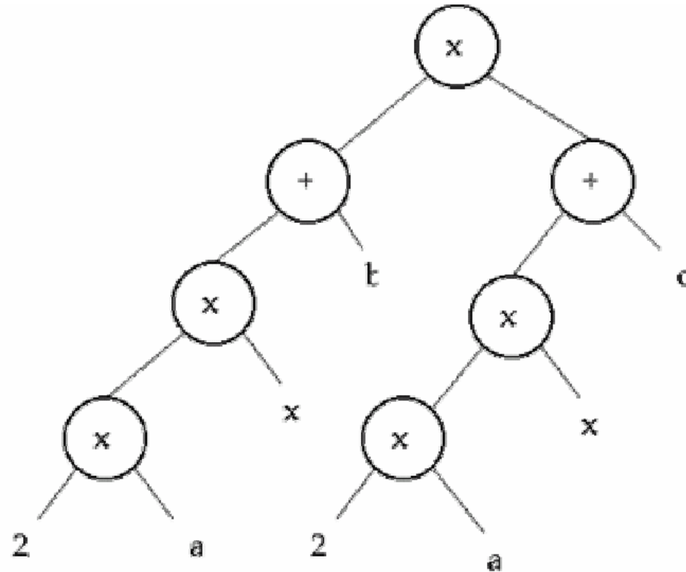
(a+b) x c	aritmética.
(a+b) = 0	relacional.
~(a V b)	lógica

El propósito es asignar un valor a través del proceso de evaluación.

Los lenguajes de programación incluyen también sentencias: unas para alterar el flujo del programa y otras para alterar el estado de la computadora.

```
If x > 0 then
S := 1
Else
S:= -1
End if;
I := I +1;
```

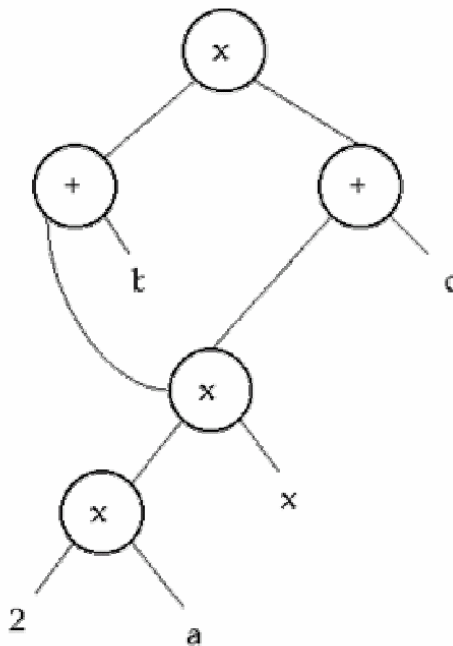
Independencia en el orden de evaluación.
(2 a x + b)(2 a x + c) = 0



La independencia de evaluación es también llamada Propiedad Church – Rosser.
Funciones puras.

```
Int f(int x)
{
return (x + x );
}
```

Transparencia Referencial.



Transparencia Referencial.

Interfaces Manifiestas:

Son las conexiones de entrada /salida entre subexpresiones y sus expresiones allegadas, obviamente son visuales.

Principios de Estructuración de Hoare.

1. Transparencia de Significado.
2. Transparencia de propósito.
3. Independencia de partes.
4. Aplicación recursiva.
5. Interfaces Estrechas.
6. Manifestación de estructuras.

Definición de funciones.

Una función es sólo un conjunto de parejas de entrada-salida.

Ejemplo función

OR(T,T) ≡ T

OR(T,F) ≡ T

OR(F,T) ≡ T

OR(F,F) ≡ F

Composición de funciones.

Implica(x,y) ≡ OR(NOT X,Y).

Estructura del lenguaje:

- Tipos de datos (particularmente secuencias).
- Operaciones
- Todos los lenguajes de programacion comprenden dos partes:
- Uno independiente del dominio de la aplicación intentada.
- Otro dependiente sobre el dominio de la aplicación.

- Parte independiente del dominio(framework).-- (Proveen la forma del lenguaje) Incluye los mecanismos básicos de lingüistica utilizados para la construcción de programas.

Para Lenguajes imperativos:

- Estructuras de control
- Mecanismos de procedimientos
- Operaciones de asignación

Para Lenguajes aplicativos:

- Definición de funciones
- Mecanismos de aplicación de funciones.
- Otras construcciones como condicionales pueden incluirse en las definiciones o en las aplicaciones de las funciones.
- Parte dependiente del dominio. -- (Proveen el contenido del lenguaje)
- Es un conjunto de componentes que son completamente usados en el dominio.

Componentes para programación numérica:

- Números de punto flotante
- Operaciones (+, -, etc)
- Relaciones(=, <, >, etc)
- La selección de frameworks determina la clase de lenguaje(imperativo o aplicativo).
- La selección de componentes orienta a la clase de problemas a los que se aplicará el lenguaje(numéricos o simbólicos).

Es posible aprender una metodología de programación aplicativa sin aprender un lenguaje de programación aplicativa particular. Por una parte, la mayoría de los lenguajes aplicativos tienen el mismo framework, basicamente es necesario definir funciones que incluyan condicionales y funciones recursivas y la llamada a estas funciones.

Por otra parte, muchos lenguajes aplicativos proveen componentes similares. Los componentes de un lenguaje aplicativo son tipos de datos propios de él y las operaciones definidas sobre valores llegan a ser de estos tipos, y estos tipos de datos pueden ser clasificados como atómicos (indivisibles) o compuestos (estructuras de datos).

a)Tipo de datos atómicos

Los tipos de datos atómicos son aquellos que son indivisibles, es decir no compuestos, por ejemplo datos boleanos, enteros, flotantes y cadenas. Para aritmética es son necesarios los tipos de datos atómicos, operadores (+, -, x, /) y las relaciones( ).

Un tipo de dato abstracto, es aquel que se denota en términos de un conjunto (abstracto) de valores y operaciones; mas que en términos de una implementación(concreta).

Usaremos la convención de que todos los tipos son abstractos, a menos que se especifique lo contrario.

b) Secuencias

Las secuencias son un tipo de dato (abstracto) compuesto porque sus valores pueden dividirse en otros mas pequeños.

Valores de las secuencias:

Definimos una secuencia como secuencias finitas de valores de un tipo, delimitados por <>:

Ejemplos:

Secuencias de enteros <1, 8, 9, 3>
Secuencias de strings <'las', 'frases', 'celebres', 'de', 'hoy'>
Secuencias de secuencias <<1, 5>, <2>, <7, 8>>
Secuencias nulas <>

c) Operaciones sobre secuencias

Para determinar las operaciones primitivas sobre secuencias que nos permitan realizar una programación completa es necesario tener presentes los requerimientos fundamentales sobre las operaciones primitivas:

- Constructor, nos permite construir alguna secuencia en un número finito de pasos.
- Selector, nos permite seleccionar un elemento de la secuencia.
- Discriminador, nos permite distinguir entre diferentes clases de secuencias.

Por simplicidad es importante tener un pequeño número de constructores, selectores y discriminadores.

Son necesarios solo dos constructores:

1. nil, con el que podemos crear una secuencia vacía, y
2. prefix, con la que podemos insertar un valor arbitrario dentro de una secuencia arbitraria.

Es natural que exista un discriminador que distinga entre las dos clases de secuencias creadas por los constructores:

null, el cual determina si el argumento es null (generado por el constructor nil).

Hay solo dos clases de secuencias para las que los selectores pueden utilizarse para recuperar componentes:

- first, el cual recupera el primer elemento de una secuencia.
- rest, el cual recupera el resto de la secuencia.

Una secuencia nula no tiene asociado un selector.

d) Arquetipo para tipo de datos abstractos

Para definir un tipo de dato abstracto debemos considerar:

- Primeramente, es necesario conocer la sintaxis de los tipos de datos: las constantes y operaciones primitivas del tipo de dato, y las reglas para combinar estos.
- Segundo, es necesario describir la semántica de los tipos de datos: el significado de las operaciones primitivas, es decir, los valores calculados cuando obtienen una entrada correcta.
- Finalmente, es necesario describir la pragmática del tipo de dato: el propósito, efectos, e implicaciones del uso actual de las operaciones primitivas.

e) Sintaxis

Permite describir la forma correcta en la que las expresiones deben ser construidas.

La sintaxis para una operación primitiva debe especificar:

1. El nombre de la operación,
2. Los tipos de los argumentos.
3. El tipo de valor que regresa.

f) Semántica

Nos permite establecer como deben interpretarse las fórmulas por la computadora. Esto se puede lograr proporcionando un conjunto de axiomas matemáticos que especifiquen los valores que pueden tomar los tipos y que determinen el valor de las formulas. Así, la especificación semántica consiste de dos partes:

1. Un axioma de existencia (nos dice que valores pueden tomar los tipos).
2. Un conjunto de ecuaciones (nos dicen que valores se obtienen de las operaciones).

g) Pragmática

Nos permite evaluar el performance de una función (nos describe una constante de tiempo que tarda una función en ejecutarse). También en esta sección podemos discutir las limitaciones permisibles sobre las implementaciones de los tipos de datos.

4.3. Introducción a LISP

LISP fue proyectado e implementado inicialmente por John McCarthy y un grupo del Massachussets Institute of Technology alrededor de 1960. El lenguaje se ha llegado a usar ampliamente para investigación en ciencias de la computación, en forma destacada en el área de inteligencia artificial (LA: robótica, procesamiento de lenguajes naturales, prueba de teoremas, sistemas inteligentes, etc.). Se han desarrollado muchas versiones de LISP a lo largo de los últimos 30 años y, de todos los lenguajes que se describen en este libro, LISP es el único que no está estandarizado ni dominado por una implementación particular.

LISP es diferente de casi todos los demás lenguajes en varios aspectos. El más notable es la equivalencia de forma entre programas y datos en el lenguaje, la cual permite ejecutar estructuras de datos como programas y modificar programas como datos. Otra característica destacada es la fuerte dependencia de la recursión como estructura de control, en vez de la iteración (formación de ciclos) que es común en casi todos los lenguajes de programación.

Historia

LISP tuvo sus inicios en el MIT alrededor de 1960. Durante los años sesenta y setenta se desarrollaron varias versiones del lenguaje. Éstas incluian el MacLisp del MIT, el Interlisp de WarrenTeitelman para la DEC PDP-10, el Spice LISP y el

Franz LISP. El Interlisp fue el dialecto dominante durante la mayor parte de esta época. Durante la parte final de la década de 1970, Gerald Sussman y Guy Steele desarrollaron una variante, como parte de una investigación sobre modelos de computación, que se conocía como “Schemer”, pero, debido a limitaciones a un nombre de seis caracteres, se abrevió a simplemente Scheme. Ésta es la versión que ha tenido el máximo impacto en el uso universitario del lenguaje.

En abril de 1981, se llevó a cabo una reunión entre las diversas facciones del LISP para tratar de fusionar los diversos dialectos en un solo lenguaje LISP, el cual se difundió con el nombre de Common LISP, a falta de un mejor nombre “común”. El nombre “Standard LISP” ya había sido adoptado por uno de estos dialectos. La estructura básica del CorumonLISP se desarrolló a lo largo de los tres años siguientes.

La época desde 1985 hasta principios de los años noventa representó probablemente la de máxima popularidad para LISP. La investigación en JA prosperaba, y en muchas universidades, quizá el 75% o más de los estudiantes de posgrado declaraban la IA como su área de especialización.

En 1986, el grupo técnico de trabajo X3J13 se reunió para estandarizar el CommonLISP,yse inició un esfuerzo por fusionar los dialectos Scheme y Common LISP. Esto fracasó, y en 1989 el IEEE desarrolló un estándar propio para Scheme. Alrededor de esta época, los efectos de la orientación a objetos en lenguajes como C++ y Smalltalk se estaban haciendo sentir, y se escribió el Common LISPObject System(CLOS; SistemadeObjetosenCommonLISP). Finalmente, en 1992, el X3J13 desarrolló un borrador para Common LISP de más de 1000 páginas, más grande que el estándar para COBOL.

Parece extraño que un lenguaje con un diseño básico tan sencillo y limpio pudiera crecer fuera de control.

Desde un principio, LISP sufrió críticas por ejecución lenta, en especial en la computadora estándar de von Neumann. Así como las funciones originales car y cdr tuvieron como modelo el hardware de la IBM 704, se estaban ideando arquitecturas de máquina altemativas para acelerar la ejecución del LISP. Varias compañías desarrollaron máquinas proyectadas para la ejecución rápida de LISP. Sin embargo, alrededor de 1989 se desarrolló una estrategia de recolección de basura para arquitecturas normales de von Neumann que era competitiva con el hardware de LISP para usos especiales. Por esta razón, ninguna de las compañías de LISP sobrevivió hasta llegar a ser un éxito comercial a largo plazo.

Ejemplo:

LISP, al igual que FORTRAN, es uno de los lenguajes de programación más antiguos.

Constituye un pilar de la comunidad dedicada a la inteligencia artificial y ha estado

evolucionando durante más de 30 años. Sin embargo, LISP ha sido siempre un lenguaje que ocupa un nicho y no es idóneo para la mayoría de las aplicaciones de computadora convencionales.

Aunque las versiones compiladas de LISP permiten que los programas en LISP se ejecuten con un poco más de eficiencia, los avances en el diseño de lenguajes, como el ML, suministran una manera más fácil de escribir programas aplicativos.

Existen numerosos dialectos de LISP, y los intentos por fusionar los formatos de Scheme y Common LISP en un lenguaje común no han tenido tanto éxito como se quisiera.

Tipos, estructuras y objetos en Lisp

a) Tipos de datos primitivos.

Los tipos primarios de objetos de datos en LISP son listas y átomos. Las definiciones de función y las listas de propiedades son tipos especiales de listas de particular importancia. También se suministran ordinariamente arreglos, números y cadenas, pero estos tipos desempeñan un papel inferior.

b) Variables y constantes.

En LISP, un átomo es el tipo elemental básico de un objeto de datos. A un átomo se le suele llamar átomo literal para distinguirlo de un número (o átomo numérico), el cual también es clasificado como átomo por casi todas las funciones en LISP.

Sintácticamente, un átomo es tan solo un identificador, una cadena de letras y dígitos que se inicia con una letra. No se suele tomar en cuenta la distinción de mayúsculas y minúsculas para identificadores. Dentro de las definiciones de función en LISP, los átomos tienen los usos ordinarios de los identificadores; se emplean como nombres de variables, nombres de funciones, nombres de parámetros formales, etcétera.

Sin embargo, un átomo en LISP no es simplemente un identificador en tiempo de ejecución. Un átomo es un objeto de datos complejo representado por una posición en la memoria, el cual contiene el descriptor de tipo para el átomo junto con un apuntador a una lista de propiedades. La lista de propiedades contiene las diversas propiedades asociadas al átomo, una de las cuales es siempre su nombre de impresión, que es la cadena que representa el átomo para entradas y salidas. Otras propiedades representan diversos enlaces para el átomo, las cuales pueden incluir la función nombrada por el átomo y otras propiedades asignadas por el programa durante la ejecución.

c) Tipos de datos estructurados.

Las listas en LISP son estructuras simples con un sólo vínculo. Cada elemento de lista contiene un apuntador a un elemento de datos y un apuntador al elemento de lista siguiente.

El último elemento de lista señala al átomo especial nil (nada) como su sucesor. Los dos apuntadores de una lista de elementos se conocen como elapuntador car y el apuntador cdr. El apuntador cdr señala al sucesor del elemento de lista. El apuntador car señala al elemento de datos.
Un elemento de lista puede contener (tener como apuntador car) un apuntador a un objeto de datos de cualquier tipo, incluso un átomo literal o numérico, o un apuntador a otra lista. Cada caso se distingue por un indicador de tipo de datos guardado en la localidad a la que se señala. Puesto que un elemento de lista puede contener un apuntador a otra lista, es posible construir estructuras de listas de complejidad arbitraria.

d) Control de secuencia.

LISP no distingue entre datos en LISP y programas en LISP, lo que conduce a un diseño sencillo para el intérprete de LISP. La ejecución de LISP es amena, comparada con la de otros lenguajes, en cuanto a que el intérprete básico se puede describir en LISP. A la función apply (aplicar) se le pasa un apuntador a la función por ejecutar, y apply interpreta la definición de esa función, empleando una segunda función eval para evaluar cada argumento.

e) Expresiones y condicionales

Un programa en LISP se compone de una sucesión de definiciones de función, donde cada función es una expresión en notación polaca de Cambridge. Condicionales. La condicional es la estructura principal para proporcionar una secuenciación alternativa de ejecución en un programa en LISP. La sintaxis es: donde cada alternativa, es (predicado expresión,). cond se ejecuta evaluando cada predicado,, por turno, y evaluando la expresión, del primero que devuelva cierto (7'). Si todos los predicados son falsos, se evalúa expresión por omisión.

f) Enunciados

LISP carece de un concepto real de ejecución de enunciados. Este lenguaje se ejecuta evaluando funciones. El grupo de llamadas de función normales representa el conjunto usual de “estructuras de control”. La expresión condes prácticamente el único mecanismo de secuenciación de ejecución de primitivas que se necesita.

La función prog proporciona los medios para la ejecución secuencial, pero no es absolutamente necesaria, aunque facilita la escritura de casi cualquier programa en LISP.

Una prog tiene la sintaxis:

Donde variables es una lista de variables que es conocida dentro de la sucesión de expresiones, y cada expresión, se ejecuta en orden. (progn(expresión1)...) es similar, sin las variables locales. Se puede salir de una prog ya sea completando la evaluación de la última expresión (en cuyo caso la prog tiene en conjunto el valor ni!) o por una llamada a la primitiva return. El argumento para return es el valor que debe ser devuelto como valor de la prog.

g) Entrada y salida

Las entradas y salidas son simplemente llamadas de función. (read) lee el átomo siguiente proveniente del teclado.

h) Definición de funciones

La función defun se usa para crear nuevas funciones. La sintaxis de esta función es:

donde argumentos son los parámetros para la función, y el cuerpo de la misma es una sola expresión (que puede ser una sucesión de progn u otras secuencias de ejecución complejas).

Muchos sistemas en LISP reservan una localidad especial en el átomo (bloque de cabecera) para la definición de la función que se nombra con ese átomo, para evitar la necesidad de buscar la definición de la función en la lista de propiedades cada vez que se llama la función. La localidad especial designa el tipo de función (expr,fexpr, etc.) y contiene un apuntador a la estructura de listas que representa su definición de función. En estas implementaciones, por lo común se suministran primitivas especiales para obtener, insertar y eliminar directamente una definición de función de la localidad especial.

4.4. Introducción a ML

ML es un lenguaje aplicativo con programas escritos en el estilo de C o Pascal. Sin embargo, es un lenguaje aplicativo con un concepto avanzado del tipo de datos. ML maneja polimorfismo y, a través de su sistema de tipos, maneja abstracciones de datos. El lenguaje base es relativamente compacto, en especial si se compara con la definición de un lenguaje como Ada. Sin embargo, la capacidad de ampliación de sus tipos le confiere un gran poder para desarrollar programas complejos.

Incluye excepciones, programación imperativa y funcional, especificaciones con base en reglas y casi todos los conceptos que se han presentado en otros lenguajes de este libro.

Si uno estuviera limitado a un lenguaje en el cual estudiar muchos conceptos de lenguajes, entonces ML sería una opción razonable, en tanto a uno no le importara la viabilidad comercial del lenguaje.

ML ha logrado avances significativos en las comunidades educativas y de investigación en ciencias de la computación. La exposición del mecanismo de tipos en nivel de lenguaje fuente es una característica que no está disponible en otros lenguajes ampliamente utilizados. Sin embargo, las aplicaciones comerciales de los programas en ML son pocas, y hasta ahora continúa principalmente como un vehículo para investigación en ciencias de la computación y uso educativo.

Historia

ML, por las siglas de MetaLanguage, fue desarrollado por Robin Miller, junto con otras personas, como un mecanismo para pruebas formales asistidas por computadora en el sistema de Lógica para Funciones Computables de Edimburgo desarrollado a mediados de los años setenta. Sin embargo, también se le encontró utilidad como lenguaje general para manipulación de símbolos. En 1983, el lenguaje se rediseñó y se amplió con conceptos como los módulos para convertirse en el Standard ML. Aunque por lo común se implementa como un intérprete, el ML se puede compilar con relativa facilidad. Los primeros compiladores aparecieron en 1984.

El uso del ML estándar se extendió a través de la comunidad dedicada a la investigación en lenguajes de programación durante los años finales de la década de 1980.

David Appel, de la Princeton University, y David MacQueen de los Bell Telephone Laboratories de AT&T desarrollaron una versión popular.

Ejemplo:

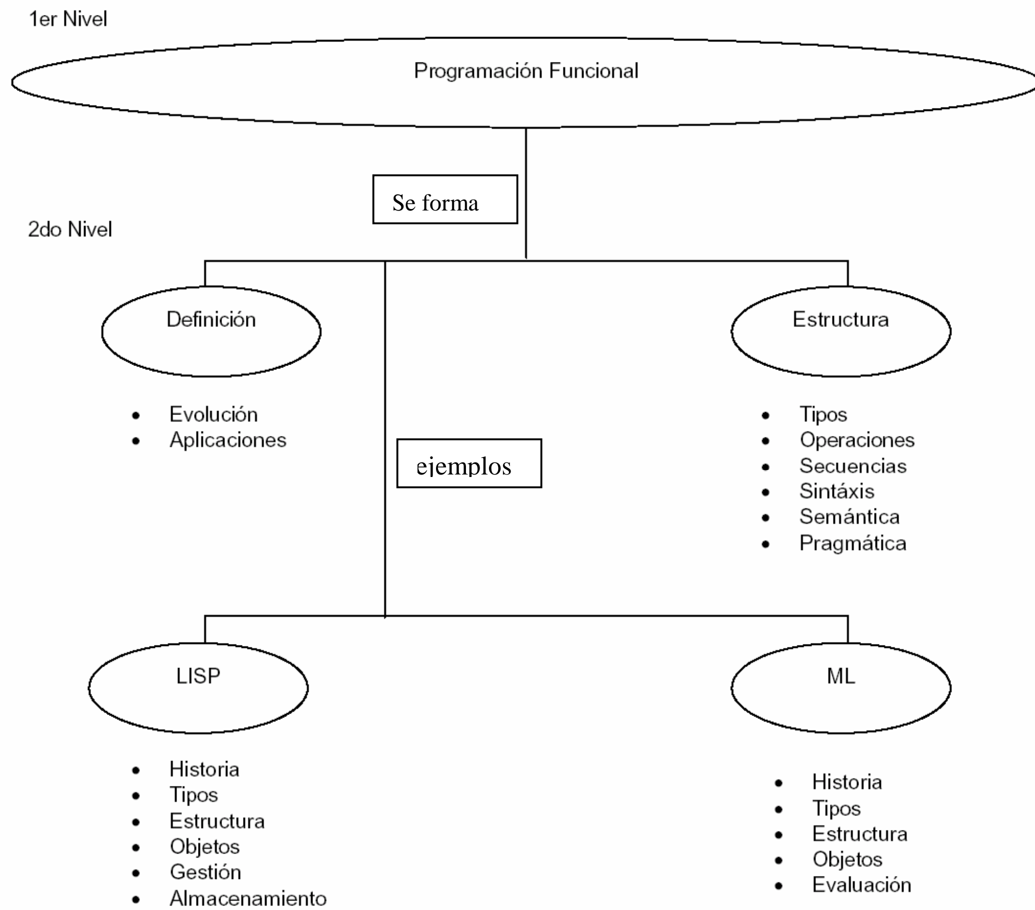
ML es un lenguaje con tipos estáticos, tipificación fuerte y ejecución aplicativa de programas. Sin embargo, difiere de otros lenguajes que hemos estudiado en cuanto a que el programador no necesita especificar los tipos. Existe un mecanismo de inferencia de tipos que determina los tipos de las expresiones resultantes. Esta inferencia de tipos da cabida a homonimia y concordancia de patrones usando unificación, en forma muy parecida a como se utiliza en Prolog.

Al igual que en LISP, un programa en ML se compone de varias definiciones de función. Cada función se tipifica estáticamente y puede devolver valores de cualquier tipo.

Puesto que es aplicativo, el almacenamiento variable se maneja en forma diferente respecto a lenguajes como C o FORTRAN. Se tiene sólo una forma limitada de asignación y, a causa de la naturaleza funcional de la ejecución, todos

los parámetros para funciones son de llamada por valor y ML crea una nueva copia de cualquier objeto complejo usando un almacén de montículo.

1. A partir del estudio de la bibliografía específica sugerida, elabore un mapa conceptual de los temas de la unidad.



2. Utilice el lenguaje LISP y el PASCAL para crear un archivo que contengan los campos siguientes: clave, nombre, dirección y teléfono y estado. Con estos campos construya el programa para abrir la estructura del archivo.

```
program Agenda;
{Implementa operaciones de acceso en ficheros y con ellas
una pequeña agenda de datos personales.}
const NOM_FICH = 'datos.dat'; NO_ENCONTRADO = -1;
type
{tipos dependientes de la aplicación particular}
        tDato= record
        nombre,dir,ciudad,tlf: string[80];
        valido: boolean
        end;
tClaveBusqueda =(nombre,dir,ciudad,tlf,igualdad);

{tipos genéricos}
tFDatos= file of tDato;
tDatoBuscado= record
        dato: tDato;
        claveBusqueda: tClaveBusqueda;
        end;
var miAgenda: tFDatos; opcion: char;
{Operaciones dependientes de la aplicación particular}

procedure EntradaDato(var dato: tDato);
        begin
        write('Nombre : '); readln(dato.nombre);
        write('Dirección: '); readln(dato.dir);
        write('Ciudad : '); readln(dato.ciudad);
        write('Teléfono : '); readln(dato.tlf);
        dato.valido:= true;
        end; {de EntradaDato}

procedure EntradaClaveBusqueda(var clave: tClaveBusqueda);
var op: char;
        begin
        repeat
        write('1->Nombre 2->Dirección 3->Ciudad 4->Teléfono): ');
        readln(op)
        until op in ['1'..'4'];
        case op of
                '1': clave:= nombre; '2': clave:= dir;
```

```
			'3': clave:= ciudad; '3': clave:= tlf
		end
		end; {de EntradaClaveBusqueda}

procedure SalidaDato(var dato: tDato);
		begin
		writeln('Nombre: ',dato.nombre); writeln('Dirección: ',dato.dir);
		writeln('Ciudad: ',dato.ciudad); writeln('Teléfono : ',dato.tlf);
		end; {de SalidaDato}

function EsBuscado(dato:tDato; datoBuscado:tDatoBuscado):boolean;
		begin
		case datoBuscado.claveBusqueda of
		nombre : EsBuscado:=Pos(datoBuscado.dato.nombre,dato.nombre)>0;
		dir : EsBuscado:=Pos(datoBuscado.dato.dir,dato.dir)>0;
		ciudad : EsBuscado:=Pos(datoBuscado.dato.ciudad,dato.ciudad)>0;
		telefono: EsBuscado:=Pos(datoBuscado.dato.tlf,dato.tlf)>0;
		igualdad: EsBuscado:= (datoBuscado.dato.nombre=dato.nombre) and
			(datoBuscado.dato.ciudad=dato.ciudad) and
			(datoBuscado.dato.dir=dato.dir) and
			(datoBuscado.dato.tlf=dato.tlf)
		end;
		end; {de EsBuscado}

{Operaciones básicas de acceso directo}
function Posicion(var f: tFDatos; pos: longint;
datoBuscado: tDatoBuscado): longint;
{busca desde una posición, devuelve la posición o NOENCONTRADO.}
var encontrado: boolean; dato: tDato;
		begin
		encontrado:= false;
		reset(f); seek(f,pos); {nos colocamos en posición de búsqueda}
		while not eof(f) and not encontrado do begin
			read(f,dato);
			encontrado:= EsBuscado(dato,datoBuscado) and dato.valido
			end;
		if encontrado then Posicion:= filepos(f)-1
		else Posicion:= NO_ENCONTRADO;
		close(f)
		end; {de Posicion}

procedure Iniciar(var f: tFDatos);
		begin
		assign(f,NOM_FICH)
		end; {de Iniciar}
```

```
procedure Crear(var f: tFDatos);
      begin
      rewrite(f); close(f);
      end; {de Crear}

procedure Insertar(var f: tFDatos; dato: tDato);
      begin
      reset(f); seek(f,filesize(f)); {inserta al final}
      write(f,dato); close(f);
      end; {de Insertar}

procedure LeerDato (var f:tFDatos; pos: integer; var dato: tDato);
      begin
      reset(f); seek(f,pos); read(f,dato); close(f);
      end; {de LeerDato}

procedure EscribirDato (var f: tFDatos; pos: integer; dato:tDato);
      begin
      reset(f); seek(f,pos); write(f,dato); close(f);
      end; {de EscribirDato}

procedure EliminarDato (var f: tFDatos; pos: integer);
var dato: tDato;
      begin
      reset(f); seek(f,pos); read(f, dato);
      dato.valido:= false; {borrado lógico}
      seek(f,pos); write(f,dato); close(f)
      end; {de EliminarDato}

{Operaciones de alto nivel}
procedure Incorporar(var f: tFDatos);
var dato: tDatoBuscado; pos:longint;
      begin
      EntradaDato(dato.dato); dato.claveBusqueda:= igualdad;
      pos:=Posicion(f,0,dato);
      if pos=NO_ENCONTRADO then Insertar(f,dato.dato)
      else begin
      LeerDato(f,pos,dato.dato);
      if not dato.dato.valido then EscribirDato(f,pos,dato.dato)
      end
      end; {de Incorporar}

procedure Consultar(var f: tFDatos);
var datoEncontrado: tDato; datoBuscado: tDatoBuscado;
```

```
pos,posBusqueda: longint; opcion: char;
          begin
          EntradaDato(datoBuscado.dato);
          EntradaClaveBusqueda(datoBuscado.claveBusqueda);
          posBusqueda:= 0;
          repeat {buscar todas las ocurrencias de ese dato}
          pos:= Posicion(f,posBusqueda,datoBuscado);
          if pos=NO_ENCONTRADO then writeln('No existe dato')
          else begin
          posBusqueda:= pos + 1; {para siguiente búsqueda}
          LeerDato(f,pos,datoEncontrado); SalidaDato(datoEncontrado);
          repeat
          write('1->Modificar 2->Eliminar 3->Buscar Siguiente ');
          write('4->Acabar: '); readln(opcion)
          until opcion in ['1'..'4'];
          case opcion of
                    '1': begin {modificar}
                    EntradaDato(datoEncontrado);
                    EscribirDato(f,pos,DatoEncontrado);
                    end;
                    '2': EliminarDato(f,pos);
                    end
          end
          until (opcion='4') or (pos=NO_ENCONTRADO)
          end;

procedure BorrarBajas(var fDatos: tfDatos);
var faux: tFDatos; dato: tDato;
          begin
          assign(faux,'faux.dat'); rewrite(faux); reset(fDatos);
          while not eof(fDatos) do begin
                    read(fDatos,dato); if dato.valido then write(faux,dato)
                    end;
          close(faux); close(fDatos); erase(fDatos);rename(faux,NOM_FICH);
          end;

begin {de Agenda}
          Iniciar(miAgenda);
          repeat
          writeln;
          repeat
          write('1->Crear 2->Añadir ');
          write('3->Consultar 4->Acabar: ');
          readln(opcion)
          until opcion in ['1'..'4'];
```

```
case opcion of
          '1': Crear(miAgenda);
          '2': Incorporar(miAgenda);
          '3': Consultar(miAgenda);
          '4': BorrarBajas(miAgenda);
          end;
until opcion='4';
end.
```

3. Realice un programa donde se sume recursivamente 10 veces.

```
PROGRAM Sumador;
VAR valor: integer;

PROCEDURE Suma(n: CARDINAL): CARDINAL;
BEGIN
  IF n=0 THEN
    RETURN 0
  ELSE
    RETURN n + Suma(n-1)
  END
END Suma10;

Valor: = Suma(10);

{ Va a sumar  10 numeros recursivamente desde 10, 10-1, 9-1, ... hasta 1-1 que es 0
(Cero) y es cuando regresa la función un valor 0 y ya no consitnua recursivamente}

Writeln('Valor de la suma (10 veces):', valor);

End.
```

Bibliografía

"Lenguajes de programación"
Diseño e implementación
Pratt – Zelkowitz
Ed.Prentice may

"Sistemas de Información para la Administración"
James A. Senn
Ed. Grupo Editorial Iberoamérica,
México, 1990

"Programing language concepts"
Ghezzi Carlo,
John Wiley and Sons
U.S.A., 1976

5. Programación Lógica

Introducción

En Octubre de 1981, el gobierno japonés y más concretamente el Ministerio Japonés de Comercio Internacional e Industria (MITI), anuncia la puesta en marcha de un proyecto revolucionario equiparable a la carrera del espacio norteamericana. Están dispuestos a ofrecer al mundo la siguiente generación, la Quinta Generación de Ordenadores. Unas máquinas de Inteligencia Artificial que pueden pensar, sacar conclusiones, emitir juicios e incluso comprender las palabras escritas y habladas.

Con este fin se crea el ICOT (Institute for New Generation Computer Technology) constituido por cuarenta brillantes investigadores de las más importantes empresas, y se les dota con todos los medios necesarios para construir la nueva clase de supercomputadoras.

La Quinta Generación prevé máquinas diseñadas para el tratamiento lógico, de capacidades análogas a las capacidades de anteriores generaciones de ordenadores para tratar operaciones aritméticas. Se trata de ordenadores que tienen el PROLOG como lenguaje nativo (lenguaje máquina), con capacidad para procesar millones de inferencias lógicas por segundo (LIPS).

PROLOG puede encontrarse en múltiples versiones diferentes. La más popular es la definida por Clocksin y Mellish, y es la más común. Afortunadamente, al ser tan sencilla la sintaxis del PROLOG, las variaciones entre distintas implementaciones son mínimas

Definición

La programación lógica es un paradigma de programación declarativa, que emerge de la fusión de tres campos:

1) La lógica de primer orden de la Lógica Matemática.
2) La demostración automática de teoremas dentro de la Inteligencia Artificial.
3) El estudio de los lenguajes formales como parte de las ciencias de la computación.

La programación lógica los programas se consideran como una serie de aserciones lógicas. De esta forma, el conocimiento se representa mediante

reglas, tratándose de sistemas declarativos. Una representación declarativa es aquella en la que el conocimiento está especificado, pero en la que la manera en que dicho conocimiento debe ser usado no viene dado. El más popular de los sistemas de programación lógica es el PROLOG.

La programación Lógica está basada en la noción de ***Relación***. Debido a que en la relación es un concepto mas general de una aplicación. La programación lógica es potencialmente de alto nivel.

Paradigma lógico

• *Programa*: conjunto de hechos y reglas.
• *Sistema*: demostrador de teoremas mediante un mecanismo de inferencia.
• *Caracterización de propiedades y relaciones.*
• *Doble dirección (E/S) de los datos.*
• *Datos parcialmente construidos.*
• *Teóricamente, el orden no importa.*

Características:

- sintáxis y semántica bien definidas
- reglas de inferencia

Introduccion a PROLOG

Historia

El desarrollo de Prolog se inició en 1970 con Alain Coulmerauer y Philippe Roussel, quienes estaban interesados en desarrollar un lenguaje para hacer deducciones a partir de texto. El nombre corresponde a “PROgramming in LOGic” (Programación en lógica). Prolog fue desarrollado en Marsella, Francia, en 1972. El principio de resolución de Kowalski, de la Universidad de Edimburgo pareció un modelo apropiado para desarrollar sobre él un mecanismo de inferencia.

Con la limitación de la resolución a cláusulas de Horn, la unificación condujo a un sistema eficaz donde el no determinismo inherente de la resolución se manejó por medio de un proceso de exploración a la inversa, el cual se podía implementar con facilidad. El algoritmo para resolución suministró la secuencia

de ejecución que se requería para implementar los ejemplos de especificaciones como la relación de vuelos antes señalada.

La primera implementación de Prolog se completó en 1972 usando el compilador de ALGOL W de Wirth, y los aspectos básicos del lenguaje actual se concluyeron para 1973. El uso del Prolog se extendió gradualmente entre quienes se dedicaban a la programación lógica principalmente por contacto personal y no a través de una comercialización del producto. Existen varias versiones diferentes, aunque bastante similares. Aunque no hay un estándar del Prolog, la versión desarrollada en la Universidad de Edimburgo ha llegado a ser utilizada ampliamente. El uso de este lenguaje no se extendió sino hasta los años ochenta.

La falta de desarrollo de aplicaciones eficaces de Prolog inhibieron su difusión.

Definición

Prolog es un lenguaje de programación seminterpretado. Su funcionamiento es muy similar a Java. El codigo fuente se compila a un código de byte el cuál se interpreta en una máquina virtual denominada Warren Abstract Machine (comúnmente denominada WAM).

PROLOG es un lenguaje de programación para ordenadores que se basa en el lenguaje de la Lógica de Primer Orden y que se utiliza para resolver problemas en los que entran en juego *objetos* y *relaciones* entre ellos. Por ejemplo, cuando decimos "Jorge tiene una moto", estamos expresando una relación entre un objeto (Jorge) y otro objeto en particular (una moto). Más aún, estas relaciones tienen un orden específico (Jorge posee la moto y no al contrario). Por otra parte, cuando realizamos una pregunta (¿Tiene Jorge una moto?) lo que estamos haciendo es indagando acerca de una relación.

Además, también solemos usar reglas para describir relaciones: "dos personas son hermanas si ambas son hembras y tienen los mismos padres". Como veremos más adelante, esto es lo que hacemos en Prolog.

Una de las ventajas de la programación lógica es que se especifica *qué* se tiene que hacer *(programación declarativa*), y no *cómo* se debe hacer *(programación imperativa*). A pesar de esto, Prolog incluye algunos predicados predefinidos meta-lógicos, ajenos al ámbito de la Lógica de Primer Orden, (var, nonvar, ==, ...), otros extra-lógicos, que tienen un efecto lateral, (write, get, ...) y un tercer grupo que nos sirven para expresar información de control de como realizar alguna tarea (el corte, ...).

Por tanto, Prolog ofrece un sistema de programación práctico que tiene algunas de las ventajas de claridad y declaratividad que ofrecería un lenguaje de

programación lógica y, al mismo tiempo, nos permite un cierto control y operatividad.

Prolog representa el lenguaje principal en la categoría de programación lógica. A diferencia de otros lenguajes no es un lenguaje de programación para usos generales, sino que está orientado a resolver problemas usando el cálculo de predicados.

Aplicaciones del Prolog

Provienen en general de dos dominios distintos:

1) Preguntas a bases de datos. Las bases de datos modernas indican típicamente relaciones entre los elementos que están guardados en la base de datos. Pero no todas estas relaciones se pueden indicar. Por ejemplo, en una base de datos de una línea aérea, puede haber entradas que indican números de vuelo, ubicación y hora de salida, y ubicación y hora de llegada. Sin embargo, si un individuo necesita hacer un viaje que requiera cambiar de avión en algún punto intermedio, es probable que esa relación no esté especificada en forma explícita.

2) Pruebas matemáticas. También se pueden especificar las relaciones entre objetos matemáticos a través de una serie de reglas y sería deseable un mecanismo para generar pruebas de teoremas a partir de este modelo. Aunque la búsqueda ascendente inherente en LISP da cabida a la construcción de sistemas generadores de pruebas, sería eficaz un lenguaje orientado en mayor grado hacia la prueba de propiedades de relaciones.

Estas dos aplicaciones son similares y se pueden resolver usando Prolog. El objetivo para Prolog era proporcionar las especificaciones de una solución y permitir que la computadora dedujera la secuencia de ejecución para esa solución, en vez de especificar un algoritmo para la solución de un problema, como es el caso normal de casi todos los lenguajes que hemos estudiado.

Por ejemplo, si se tiene información de vuelos de una línea aérea de la forma: vuelo (número_de_vuelo ciudad_origen, ciudad_destino hora_de_salida hora_de llegada) entonces todos los vuelos de Los Ángeles a Baltimore se pueden especificar ya sea como vuelos directos: vuelo(número_de_vuelo, Los Ángeles, Baltimore hora_de_salida hora de llegada) o como vuelos con una parada intermedia, especificados como:

vuelo(vuelol, Los Ángeles, X, salel, llegal),
vuelo(vuelo2, X,Baltimore, sale2, llega2),
sale2>= llegal+30

lo que indica que se está especificando una ciudad X, a la cual llega un vuelo proveniente de Los Ángeles, y de la que sale un vuelo para Baltimore, y el vuelo con destino a Baltimore sale al menos 30 minutos después de que llega el vuelo de los Ángeles para dar tiempo a cambiar de avión. No se especifica un algoritmo; sólo se han indicado las condiciones para tener una solución correcta.

El lenguaje mismo, si se puede enunciar un conjunto de condiciones de esta naturaleza, suministrará la secuencia de ejecución que se necesita para encontrar el vuelo apropiado.

Un entorno de desarrollo Prolog se compone de:

- **Un compilador**. Transforma el código fuente en código de byte. A diferencia de Java, no existe un standard al respecto. Por eso, el codigo de byte generado por un entorno de desarrollo no tiene por que funcionar en el intérprete de otro entorno.

- **Un intérprete**. Ejecuta el código de byte.

- **Un shell o top-level**. Se trata de una utilidad que permite probar los programas, depurarlos, etc. Su funcionamiento es similar a los interfaces de línea de comando de los sistemas operativos.

- **Una biblioteca de utilidades**. Estas bibliotecas son, en general, muy amplias. Muchos entornos incluyen (afortunadamente) unas bibliotecas standard-ISO que permiten funcionalidades básicas como manipular cadenas, entrada/salida, etc.

Prolog es aproximadamente diez veces más lento que el lenguaje C. Pero también hemos de admitir que un programa en Prolog ocupa aproximadamente diez veces menos, en líneas de código y tiempo de desarrollo, que el mismo programa escrito en C. Además las técnicas de optimización de código en

Objetos de datos

a) Tipos de datos primitivos

Variables y constantes. Los datos en Prolog incluyen enteros (1,2,3, ...), reales (1.2,3.4, 5.4, ...) y caracteres (‘a’, ‘b’, ...). Los nombres que comienzan con letra minúscula representan hechos concretos, si son parte de un hecho de la base de datos, o, si están escritos con mayúsculas, representan variables que se pueden unificar con hechos durante la ejecución de Prolog. El alcance de una variable es la regla dentro de la cual se usa.

b) Tipos de datos estructurados

Los objetos pueden ser átomos (constantes y variables de cadena) o listas. Una lista se representa como [A,B,C,..jJ. La notación [AIB] se usa para indicar A como cabeza de la lista y B como cola de la lista:

La cadena ‘abcd’ representa la lista [‘a’, ‘b’, ‘c’, ‘d’]. La variable _ es una variable arbitraria sin nombre. Así, [A I j hace coincidir A con la cabeza de cualquier lista.

c) Tipos definidos por el usuario

No existen verdaderos tipos definidos por el usuario; sin embargo, las reglas para definir relaciones pueden actuar como si fueran tipos de usuario.

Definiciones y conceptos:

Hechos

Expresan relaciones entre objetos. Supongamos que queremos expresar el hecho de que "un coche tiene ruedas". Este hecho, consta de dos objetos, "coche" y "ruedas", y de una relación llamada "tiene". La forma de representarlo en PROLOG es:

tiene(coche,ruedas).

- Los nombres de objetos y relaciones deben comenzar con una letra minúscula.
- Primero se escribe la relación, y luego los objetos separados por comas y encerrados entre paréntesis.
- Al final de un hecho debe ir un punto (el caracter ".").
- El orden de los objetos dentro de la relación es arbitrario, pero debemos ser coherentes a lo largo de la base de hechos.

Variables

Representan objetos que el mismo PROLOG determina . Una variable puede estar instanciada ó no instanciada. Estar instanciada cuando existe un objeto determinado representado por la variable. De este modo, cuando preguntamos "¨

Un coche tiene X ?",

PROLOG busca en los hechos cosas que tiene un coche y respondería:

X = ruedas.

instanciando la variable X con el objeto ruedas.

Los nombres de variables comienzan siempre por una letra mayúscula. Un caso particular es la variable anónima, representada por el carácter subrayado ("_").

Es una especie de comodín que utilizaremos en aquellos lugares que debería aparecer una variable, pero no nos interesa darle un nombre concreto ya que no vamos a utilizarla posteriormente.

El ámbito de las variables. Cuando en una regla aparece una variable, el ámbito de esa variable es únicamente esa regla. Supongamos las siguientes reglas:

(1) hermana_de(X,Y) :- hembra(X), padres(X,M,P), padres(Y,M,P).
(2) puede_robar(X,P) :- ladron(X), le_gusta_a(X,P), valioso(P).

Aunque en ambas aparece la variable X (y la variable P), no tiene nada que ver la X de la regla (1) con la de la regla (2), y por lo tanto, la instanciación de la X en (1) no implica la instanciacion en (2). Sin embargo todas las X de *una misma regla* sí que se instanciar n con el mismo valor.

Reglas

Las reglas se utilizan en PROLOG para significar que un hecho depende de uno ó mas hechos. Son la representación de las implicaciones lógicas del tipo p ---> q (p implica q).

- Una regla consiste en una cabeza y un cuerpo, unidos por el signo ":-".
- La cabeza está formada por un único hecho.
- El cuerpo puede ser uno o más hechos (conjunción de hechos), separados por una coma (","), que actúa como el "y" lógico.
-Las reglas finalizan con un punto (".").

La cabeza en una regla PROLOG corresponde al consecuente de una implicación lógica, y el cuerpo al antecedente. Este hecho puede conducir a errores de representación.

Supongamos el siguiente razonamiento lógico:

tiempo(lluvioso) ----> suelo(mojado)
suelo(mojado)

Que el suelo esté mojado, es una condición suficiente de que el tiempo sea lluvioso, pero no necesaria. Por lo tanto, a partir de ese hecho, no podemos deducir mediante la implicación, que esté lloviendo (pueden haber regado las calles). La representación *correcta* en PROLOG, sería:

suelo(mojado) :- tiempo(lluvioso).
suelo(mojado).

Adviértase que la regla está "al revés". Esto es así por el mecanismo de deducción hacia atrás que emplea PROLOG. Si cometiéramos el *error* de representarla como:

tiempo(lluvioso) :- suelo(mojado).
suelo(mojado).

PROLOG, partiendo del hecho de que el suelo está mojado, deduciría incorrectamente que el tiempo es lluvioso.

Para generalizar una relación entre objetos mediante una regla, utilizaremos variables.

Por ejemplo:

```
Representación lógica | Representación PROLOG
---------------------------------+-----------------------------------
es_un_coche(X) ----> | tiene(X,ruedas) :-
tiene(X,ruedas) | es_un_coche(X).
```

Con esta regla generalizamos el hecho de que cualquier objeto que sea un coche, tendrá ruedas. Al igual que antes, el hecho de que un objeto tenga ruedas, no es una condición suficiente de que sea un coche. Por lo tanto la representación inversa sería incorrecta.

Operadores

Son predicados predefinidos en PROLOG para las operaciones matemáticas básicas.

Su sintaxis depende de la posición que ocupen, pudiendo ser infijos ó prefijos.

Por ejemplo el operador suma ("+"), podemos encontrarlo en forma prefija '+(2,5)' ó bien infija, '2 + 5'.

También disponemos de predicados de igualdad y desigualdad.

X = Y igual
X \= Y distinto
X < Y menor
X > Y mayor
X =< Y menor ó igual
X >= Y mayor ó igual

Al igual que en otros lenguajes de programación es necesario tener en cuenta la precedencia y la asociatividad de los operadores antes de trabajar con ellos.

En cuanto a precedencia, es la típica. Por ejemplo, 3+2*6 se evalúa como 3+(2*6).

En lo referente a la asociatividad, PROLOG es asociativo por la izquierda. Así, 8/4/4 se interpreta como (8/4)/4. De igual forma, 5+8/2/2 significa 5+((8/2)/2).

El operador 'is'. Es un operador infijo, que en su parte derecha lleva un término que se interpreta como una expresión aritmética, contrastándose con el término de su izquierda. Por ejemplo, la expresión '6 is 4+3.' es falsa. Por otra parte, si la expresión es 'X is 4+3.', el resultado ser la instanciación de X:

X = 7

Una regla PROLOG puede ser esta:

densidad(X,Y) :- poblacion(X,P), area(X,A), Y is P/A.

Estilo de programación en PROLOG

Los lenguajes de Programación Lógica, al igual que los lenguajes procedimentales, necesitan de una metodología para construir y mantener programas largos, así como un buen estilo de programación. Prolog ofrece mecanismos de control típicos de cualquier lenguaje imperativo y no puras propiedades o relaciones entre objetos. Dichos mecanismos contribuyen a dar mayor potencia y expresividad al lenguaje, pero violan su naturaleza lógica.

Adoptando un estilo apropiado de programación podemos beneficiarnos de esta doble naturaleza de Prolog.

- *Metodología de Diseño "top-down*": descomponemos el problema en subproblemas y los resolvemos. Para ello solucionamos primero los pequeños problemas.

- Una vez analizado el problema y separado en trozos, el siguiente paso es decidir como representar y manipular tanto los objetos como las relaciones del problema. Debemos elegir los nombres de los predicados, variables, constantes y estructuras de manera que aporten claridad a nuestros programas (palabras o nombres mnemónicos que estén relacionados con lo que hacen).

- El siguiente paso es asegurarse de que la organización y la sintaxis del programa sea clara y fácilmente legible.

- Llamamos *procedimiento* al conjunto de cláusulas para un predicado dado. Agruparemos por bloques los procedimientos de un mismo predicado (mismo nombre y mismo número de argumentos). Así, cada cláusula de un procedimiento comenzará en una línea nueva, y dejaremos una línea en blanco entre procedimientos.

- Si el cuerpo de una regla es lo bastante corto, lo pondremos en una línea; sino, se escriben los objetivos de las conjunciones indentados y en líneas separadas.

- Primero se escriben los hechos, y luego las reglas.

- Es recomendable añadir comentarios y utilizar espacios y líneas en blanco que hagan el programa más legible. El listado del programa debe estar "autodocumentado". Debemos incluir para cada procedimiento y antes de las cláusulas el *esquema de la relació*n.

- Los argumentos deben ir precedidos por el signos `+', `-' o `?'. `+' indica que el argumento es de entrada al predicado (debe estar instanciado cuando se hace la llamada), `-' denota que es de salida y `?' indica que puede ser tanto de entrada como de salida.

- Agrupar los términos adecuadamente. Dividir el programa en partes razonablemente autocontenidas (por ejemplo, todos los procedimientos de procesado de listas en un mismo fichero).

- Evitar el uso excesivo del corte.

- Cuidar el orden de las cláusulas de un procedimiento. Siempre que sea posible escribiremos las condiciones limite de la recursividad antes de las demás cláusulas. Las cláusulas "recogetodo" tienen que ponerse al final.

Subprogramas y gestión de almacenamiento

Prolog tiene dos modos: modo de consulta y modo de pregunta. En el modo de consulta se introducen nuevas relaciones en el almacenamiento dinámico de la base de datos. Durante el modo de pregunta, se ejecuta un intérprete simplemente basado en pilas para evaluar preguntas.

Ambiente local de referencia. Todos los nombres de variables son locales para la regla en la cual están definidos. La unificación prevé la interacción de nombres locales de una regla con los nombres de otra regla.

Ambiente común de referencia. Todos los datos son compartidos. No existe un verdadero concepto de ambiente local o global de referencia.

Paso de parámetros. La unificación prevé el paso de argumentos entre reglas.

Funciones normales. Casi todos los sistemas en Prolog incluyen varias funciones integradas para ayudar en la generación de programas:

consult(nombredearchivo) lee el archivo nombredearchivo y anexa nuevos hechos y reglas a la base de datos. nombredearchivo también se puede escribir como ‘nombredearchiv ‘si el nombre del archivo. contiene caracteres incrustados no pertenecientes a un identificador. consult(nombredearchivo) se suele poder especificar simplemente como
[nombredearchivo].

· reconsult(nombredearchivo) sobreescribe relaciones en la base de datos.
· fail siempre fracasa.
· see(nombredearchivo) lee entradas desde nombredearchivo como una serie de reglas.
· write(término) escribe término.
· tell(nombredearchivo) reorienta la salida de write a nombredearchivo.
· told cierra el archivo respecto a una tell previa y reorienta la salida de write a la terminal normal de visualización.
· nl pasa al renglón siguiente (de entradas y salidas).
· atom(X) es un predicado que devuelve cierto si X es un átomo (constantes de cadena o nombres de variable).
· var(X) es un predicado que devuelve cierto si Xes una variable.
· integer(X) es un predicado que devuelve cierto si Xes un entero.
· trace activa la depuración porrastreo mostrando cada paso de la ejecución. notrace la desactiva.

Evaluación del lenguaje

Prolog ha alcanzado una medida razonable de éxito como lenguaje para resolver problemas de relaciones, como en el procesamiento de consultas a bases de datos. Ha alcanzado un éxito limitado en otros dominios.

El objetivo original de poder especificar un programa sin proporcionar sus detalles algorítmicos no se ha alcanzado realmente. Los programas en Prolog "se leen secuencialmente", aunque el desarrollo de reglas sigue un estilo aplicativo. Se usan cortes con frecuencia para limitar el espacio de búsqueda para una regla, con el efecto de hacer el programa tan lineal en cuanto a ejecución como un programa típico en C o Pascal. Aunque las reglas se suelen expresar en forma aplicativa, el lado derecho de cada regla se procesa de manera secuencial. Esto sigue un patrón muy similar al de los programas en ML.

Perspectiva del lenguaje

Un programa en Prolog se compone de una serie de hechos, relaciones concretas entre objetos de datos (hechos) y un conjunto de reglas, es decir, un patrón de relaciones entre los objetos de la base de datos. Estos hechos y reglas se introducen en la base de datos a través de una operación de consulta.

Un programa se ejecuta cuando el usuario introduce una pregunta, un conjunto de términos que deben ser todos ciertos. Los hechos y reglas de la base de datos se usan para determinar cuáles sustituciones de variables de la pregunta (llamadas unificación) son congruentes con la información de la base de datos.

Como intérprete, Prolog solicita entradas al usuario. El usuario digita una pregunta o un nombre de función. La verdad ("yes") o falsedad ("no") de esa preguntase imprime, así como una asignación a las variables de la pregunta que hacen cierta la pregunta, es decir , que unifican la pregunta. Si se introduce un ";", entonces se imprime el próximo conjunto de valores que unifican la pregunta, hasta que no son posibles más sustituciones, momento en el que Prolog imprime "no" y aguarda una nueva pregunta. Un cambio de renglón se interpreta como terminación de la búsqueda de soluciones adicionales.

La ejecución de Prolog, aunque se basa en la especificación de predicados, opera en forma muy parecida a un lenguaje aplicativo como LISP o ML. El desarrollo de reglas en Prolog requiere el mismo "pensamiento recursivo" que se necesita para desarrollar programas en esos otros lenguajes aplicativos.

Prolog tiene una sintaxis y semántica simples. Puesto que busca relaciones entre una serie de objetos, la variable y la lista son las estructuras de datos básicas que se usan. Una regla se comporta en forma muy parecida a un procedimiento, excepto que el concepto de unificación es más complejo que el proceso relativamente sencillo de sustitución de parámetros por expresiones.

Prolog emplea la misma sintaxis que C y PL/I para comentarios: /* ... */.

ACTIVIDADES COMPLEMENTARIAS

1- A partir del estudio de la bibliografía sugerida, elabore un mapa conceptual de los temas de la unidad

PARADIGMA LOGICO

Emerger de

LOGICA MATEMATICA

DEMOSTRACIÓN DE TEOREMAS

LENGUAJES FORMALES

Surge

PROGRAMACIÓN LOGICA

Ejemplo

PROLOG

Se forma

DEFINICIONES Y CONCEPTOS

- Hechos
- Variables
- Reglas
- operadores

CARACTERÍSTICAS DEL LENGUAJE

-historia
-definición
-aplicaciones
-entorno
-perspectiva

2. Ejemplo de programa en PROLOG:

```
quiere_a(maria,enrique).

quiere_a(juan,jorge).

quiere_a(maria,susana).

quiere_a(maria,ana).

quiere_a(susana,pablo).

quiere_a(ana,jorge).

varon(juan).

varon(pablo).

varon(jorge).

varon(enrique).

hembra(maria).

hembra(susana).

hembra(ana).

teme_a(susana,pablo).

teme_a(jorge,enrique).

teme_a(maria,pablo).

/* Esta linea es un comentario */

quiere_pero_teme_a(X,Y) :- quiere_a(X,Y), teme_a(X,Y).

querido_por(X,Y) :- quiere_a(Y,X).

puede_casarse_con(X,Y) :- quiere_a(X,Y), varon(X), hembra(Y).

puede_casarse_con(X,Y) :- quiere_a(X,Y), hembra(X), varon(Y).
```

A. Bibliografía

“Lenguajes de programación” Diseño e implementación
Pratt – Zelkowitz
Ed.Prentice may

“Sistemas de Información para la Administración”
James A. Senn
Ed. Grupo Editorial Iberoamérica,
México, 1990

“Enciclopedia Encarta 99”
Microsoft

UNIDAD 6: Matriz comparativa de los paradigmas

Análisis comparativo de los diferentes paradigmas de la programación

PARADIGMAS	CARACTERÍSTICAS	LENGUAJE EJEMPLO
IMPERATIVO Los lenguajes imperativos son lenguajes controlados por mandatos u orientados a enunciados (instrucciones).	- Comandos o Instrucciones. - Orientados a la utilización por programadores profesionales. - Requiere especificación sobre cómo ejecutar una tarea. - Se deben especificar todas las alternativas. - Requiere gran número de instrucciones de procedimiento. - El código puede ser difícil de leer, entender y mantener. - Lenguaje creado originalmente para operación por lotes. - Puede ser difícil de aprender. - Difícil de depurar. - Orientados comúnmente a archivos.	• Fortran (FORmula TRANslation) • Basic (BegInner's AII-purpose Symbolic Instruction Code) • Cobol (COmmon Business-Oriented Language) • Pascal • Ada • ALGOL • Smalltalk • PL/l • C
ORIENTADA A OBJETOS La idea principal de la programación orientada a objetos es construir programas que utilizan objetos de software. Un objeto puede considerarse como una entidad independientemente de cómputo con sus propios datos y programación.	-Se basa en conceptos sencillos: objetos y atributos, el todo y las partes, clases y miembros. - Encapsulación - Herencia - Polimorfismo - Facilita la creación de prototipos - Simplicidad - Modularidad -Facilidad para hacer modificaciones - Posibilidad de extenderlo	• Java • PHP • ASP • PERL • BASH • Visual J ++
FUNCIONAL La programación funcional se basa en el uso de las funciones matemáticas para producir mejores efectos y que sus resultados sean más eficientes.	- Semánticas claras, simples y matemáticamente bien fundadas - Cercanos al nivel de abstracción de las especificaciones formales / informales de los problemas a resolver - Referencialmente transparentes: Comportamiento matemático adecuado que permite razonar sobre los programas -Soportan técnicas muy	• ML (MetaLanguage) • LISP

	avanzadas de desarrollo, mantenimiento y validación de programas -Altas dosis de paralelismo implícito - Aplicaciones variadas y de gran interés - Provee de un paradigma para programar en paralelo. - Es ampliamente aplicada en la Inteligencia Artificial. - Es bastante útil para el desarrollo de prototipos.	
LOGICA La programación lógica los programas se consideran como una serie de aserciones lógicas. De esta forma, el conocimiento se representa mediante reglas, tratándose de sistemas declarativos	- Emerge de la lógica de primer orden de la Lógica Matemática. - Busca la demostración automática de teoremas dentro de la Inteligencia Artificial, mediante un mecanismo de inferencia. - Se basa en el estudio de los lenguajes formales como parte de las ciencias de la computación. - Caracterización de propiedades y relaciones. - Doble dirección (E/S) de los datos. - Datos parcialmente construidos. -Teóricamente, el orden no importa. -Sintáxis y semántica bien definidas -Reglas de inferencia	• PROLOG

UNIVERSIDAD NACIONAL AUTONOMA DE MÉXICO.

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN.

SISTEMA DE UNIVERSIDAD ABIERTA.

INFORMATICA IV.

UNIDAD VI.

ANÁLISIS COMPARATIVO DE LOS DIFERENTES PARADIGMAS DE LA PROGRAMACIÓN.

LIC. VICENTE HERNANDEZ.

6.1 DEFINICION DE PARADIGMA.

Según el Diccionario El Pequeño Larousse Ilustrado, edición 1999, en su pagina 759 anota que Paradigma viene de la palabra griega *paradeigma:* Ejemplo que sirve de norma. En la Filosofía Platónica: El mundo de las ideas, Prototipo del mundo sensible que vivimos.
Como se puede apreciar es un ejemplo a seguir por cuanto a lo valioso que es y representa.
El Paradigma en el ambiente de programación es la plataforma teórica-conceptual a través de la cual, el planteamiento de un problema se puede resolver relacionando sus elementos con características diferentes como son la definición y tipo de los datos, la arquitectura del compilador, entre otros elementos.

6.2 PARADIGMAS DE ALGUNOS LENGUAJES DE PROGRAMACIÓN.

Aunado a los matices del pensamiento para resolver los problemas a través de un algoritmo se agregan las restricciones de la arquitectura de la computadora, la arquitectura del compilador, los tiempos de ejecución, la eficiencia del empleo del sistema en general para combinar posteriormente combinar estos elementos para investigar la creación de nuevas plataformas teóricas en el manejo de las variables, su definición, la definición de la estructura de datos, la técnica recursiva, y eventualmente encontrar un estilo nuevo en la solución de problemas. La concepción de paradigmas no solo trasciende en la búsqueda de configuraciones más eficientes sino también en el estudio teórico de la computación y consecuentemente encontrar un modelo representativo de la realidad.

PARADIGMAS	CARACTERÍSTICAS	LENGUAJE EJEMPLO
IMPERATIVO Los lenguajes imperativos son lenguajes controlados por mandatos u orientados a enunciados (instrucciones).	- Comandos o Instrucciones. - Orientados a la utilización por programadores profesionales. - Requiere especificación sobre cómo ejecutar una tarea. - Se deben especificar todas las alternativas. - Requiere gran número de instrucciones de procedimiento. - El código puede ser difícil de leer, entender y mantener. - Lenguaje creado originalmente para operación por lotes. - Puede ser difícil de aprender. - Difícil de depurar. - Orientados comúnmente a archivos.	• Fortran (FORmula TRANslation) • Basic (Beginner's All-purpose Symbolic Instruction Code) • Cobol (COmmon Business-Oriented Language) • Pascal • Ada • ALGOL • Smalltalk • PL/l • C
ORIENTADA A OBJETOS La idea principal de la programación orientada a objetos es construir programas que utilizan objetos de software. Un objeto puede considerarse como una entidad independientemente de cómputo con sus propios datos y programación.	-Se basa en conceptos sencillos: objetos y atributos, el todo y las partes, clases y miembros. - Encapsulación - Herencia - Polimorfismo - Facilita la creación de prototipos - Simplicidad - Modularidad -Facilidad para hacer modificaciones - Posibilidad de extenderlo	• Java • PHP • ASP • PERL • BASH • Visual J ++
FUNCIONAL La programación funcional se basa en el uso de las funciones matemáticas para producir mejores efectos y que sus resultados sean más eficientes.	- Semánticas claras, simples y matemáticamente bien fundadas - Cercanos al nivel de abstracción de las especificaciones formales / informales de los problemas a resolver - Referencialmente transparentes: Comportamiento matemático adecuado que permite razonar sobre los programas -Soportan técnicas muy avanzadas de desarrollo, mantenimiento y validación de programas -Altas dosis de paralelismo implícito - Aplicaciones variadas y de gran interés - Provee de un paradigma para programar en paralelo. - Es ampliamente aplicada en la	• ML (MetaLanguage) • LISP

	Inteligencia Artificial. - Es bastante útil para el desarrollo de prototipos.	
LOGICA La programación lógica los programas se consideran como una serie de aserciones lógicas. De esta forma, el conocimiento se representa mediante reglas, tratándose de sistemas declarativos	- Emerge de la lógica de primer orden de la Lógica Matemática. - Busca la demostración automática de teoremas dentro de la Inteligencia Artificial, mediante un mecanismo de inferencia. - Se basa en el estudio de los lenguajes formales como parte de las ciencias de la computación. - Caracterización de propiedades y relaciones. - Doble dirección (E/S) de los datos. - Datos parcialmente construidos. -Teóricamente, el orden no Importa. -Sintaxis y semántica bien definidas -Reglas de inferencia	• PROLOG

6.3 PARADIGMA DEL ALGORITMO DE EUCLIDES.

Método básico:

En esta lección vamos a ver el Algoritmo de Euclides que es un método muy astuto para calcular el divisor común entre dos números. Una de sus principales ventajas es que no es necesario calcular los divisores ni los factores de los números, por lo que anda muy rápido, incluso cuando alguno de los números es un primo o tiene algún factor primo grande.

Este método se basa en la siguiente propiedad:

- **si A y B son enteros entonces DCM(A,B) = DCM(A-B,B)**

A continuación se puede ver el desarrollo.

La idea es ir achicando los números tanto como se pueda hasta que el DCM sea obvio, por ejemplo que sean ambos números iguales, o que uno de los dos sea cero.
Por ejemplo: para calcular el DCM(8069,5209)
DCM(8069,5209) =DCM(8069-5209,5209) =DCM(2860,5209) =DCM(5209-2860,2860) =DCM(2349,2860) =DCM(2860-2349,2349) =DCM(511,2349) =DCM(2349-511,511) =DCM(1838,511) =DCM(1838-511,511) =DCM(1327,511) =DCM(1327-511,511) =DCM(816,511) =DCM(816-511,511) =DCM(305,511) =DCM(511-305,305) =DCM(206,305) =DCM(305-206,206) =DCM(99,206) =DCM(206-99,99) =DCM(107,99) =DCM(107-99,99) =DCM(8,99) =DCM(99-8,8) =DCM(91,8) =DCM(91-8,8) =DCM(83,8) =DCM(83-8,8) =DCM(75,8) =DCM(75-8,8) =DCM(67,8) =DCM(67-8,8) =DCM(59,8) =DCM(59-8,8) =DCM(51,8) =DCM(51-8,8) =DCM(43,8)

=DCM(43-8,8) =DCM(35,8) =DCM(35-8,8) =DCM(27,8) =DCM(27-8,8) =DCM(19,8) =DCM(19-8,8) =DCM(11,8) =DCM(11-8,8) =DCM(3,8) =DCM(8-3,3) =DCM(5,3) =DCM(5-3,3) =DCM(2,3) =DCM(3-2,2) =DCM(1,2)= DCM(2-1,1) =DCM(1,1) =DCM(1-1,1) =DCM(0,1) =1
Si revisan con cuidado, van a ver que a pesar de que son muchas cuentas, todas son muy simples (solo hay que restar y restar y restar). En cambio si uno trata de calcular el divisor común mayor de la manera usual, puede pasarse bastante tiempo buscando cuales son los divisores de 8069 y 5209.
Veamos otro ejemplo: Calcular el DCM(1651,2353)
DCM(1651,2353) =DCM(702,1651) =DCM(949,702) =DCM(247,702) =DCM(455,247) =DCM(208,247) =DCM(208,247) =DCM(39,208) =DCM(169,39) =DCM(130,39) =DCM(91,39) =DCM(52,39) =DCM(13,39) =DCM(26,13) =DCM(13,13) =DCM(0,13) =13
Se puede hacer una función que haga estas cuentas para calcular el DCM entre dos números con el mismo formato que usábamos en las lecciones anteriores. (En realidad esta función anda bien si a y b son enteros positivos o cero.)

```
DefIng A-Z 'Todas las variables son enteros largos
Function DCMRestando(a,b)
   CopiaA=a 'Copia a A (ver nota mas abajo)
   CopiaB=b 'Copia a B (ver nota mas abajo)
   'Print "DCM(" ; CopiaA ; "," ; CopiaB ; ")"
   Do While CopiaB <> 0
      If CopiaA < CopiaB Then 'si están al revés los doy vuelta
         Temp = CopiaA
         CopiaA = CopiaB
         CopiaB = Temp
      End If
      CopiaA = CopiaA - CopiaB
      'Print "=DCM(" ; CopiaA ; "," ; CopiaB ; ")"
   Loop
   DCMRestando = CopiaA
   'Print "=" ; CopiaA
End Function
```

(Todas las salidas a pantalla están comentadas para que no molesten en un programa normal, pero puede ser útil activarlas para entender como funciona exactamente la función.)

Un poco más rápido:

Para acelerar un poco las cuentas se puede utilizar la división entera para hacer varios de estos pasos al mismo tiempo. Al dividir A por B el cociente representa la cantidad de veces que entra B completamente en A y el resto es lo que sobra para que la cuenta sea exacta, por lo que después de restarle B a A varias veces vamos a terminar obteniendo el resto.
Por ejemplo en el primer ejemplo aparece
DCM(2349,511) =DCM(1838,511) =DCM(1327,511) =DCM(816,511) =DCM(305,511)

y al hacer la división entera de 2349 dividido 511 se obtiene como resultado 4 y resto 305. Por esto se ve en el ejemplo hay que restarle 4 veces 511 y se obtiene como resultado 305.
De esta manera los ejemplos quedan:
DCM(8069,5209) =DCM(8069-5209,5209) =DCM(2860,5209) =DCM(5209-2860,2860) =DCM(2349,2860) =DCM(2860-2349,2349) =DCM(511,2349) =DCM(2349-4*511,511) =DCM(1838,511) =DCM(305,511) =DCM(511-305,305) =DCM(206,305) =DCM(305-206,206) =DCM(99,206) =DCM(206-2*99,99) =DCM(8,99) =DCM(99-12*8,8) =DCM(3,8) =DCM(8-2*3,3) =DCM(2,3) =DCM(3-2,2) =DCM(1,2)= DCM(2-2*1,1) =DCM(0,1) =1
DCM(1651,2353) =DCM(702,1651) =DCM(949,702) =DCM(247,702) =DCM(208,247) =DCM(208,247) =DCM(39,208) =DCM(13,39) =DCM(0,13) =13
Con esta observación puede mejorar un poco la función de la siguiente manera usando el resto de la división entera. En esta función no se necesita verificar si el valor de a es mayor que el de b, porque si a fuera más chico la primera división de 0 y sólo se dan vuelta los números.

```
DefIng A-Z 'Todas las variables son enteros largos
Function DCMEuclides(a,b)
   CopiaA=a 'Copia a A (ver nota mas abajo)
   CopiaB=b 'Copia a B (ver nota mas abajo)
   Do While CopiaB<>0'Repite hasta que el resto de cero
      Temp = CopiaA Mod CopiaB
      'Como el resto de la división es siempre menor que
      'CopiaB entonces los doy vuelta
      CopiaA = CopiaB
      CopiaB = Temp
   Loop
   DCMEuclides = CopiaA
End Function
```

Es sorprendente lo corta que es esta función y además anda realmente muy rápido, así que en (¿casi?) todos los casos es preferible usar esta función a todas las que vimos antes.
Nota: En ambos programas se hacen copias de los parámetros a y b para operar con las copias y no modificarlos. Esto es necesario porque sino al cambiarse los valores dentro de la función, se cambian los valores de las variables en el programa principal (o de donde se llame a la función). Esto sólo es necesario en QB y VB, pero no en Pascal, C, etc. porque en estos lenguajes la función recibe normalmente sólo el valor de las variables. En general uno ni se preocupa por todo esto, hasta que el programa comienza a hacer cosas inexplicables.

Comentarios:

Cuando hay que hacer este tipo de cuentas a mano, a veces se pueden utilizar otras propiedades, como por ejemplo, es obvio cuando un número es múltiplo de 10, o cuando es par. Así que también se podrían resolver los ejemplo anteriores de la siguiente manera:
DCM(8069,5209) =DCM(2860,5209) =DCM(286*10,5209) =DCM(286,5209) =DCM(143*2,5209) =DCM(143,5209) =DCM(143,5209) =DCM(5209-143*3,143) =DCM(4780,143) =DCM(239*20,143) =DCM(239,143) =DCM(96,143) =DCM(49,96) =DCM(7*7,3*31) =1

DCM(1651,2353) =DCM(702,1651) =DCM(351,1651) =DCM(1300,351)=DCM(13,351)=13
Sin embargo, la mayoría de estos trucos que se pueden hacer a ojo usando las reglas de divisibilidad más sencillas no son tan útiles para utilizar en un programa.

6.4 LOS PARADIGMAS NUMERICOS.

Los algoritmos se clasifican en numéricos y los no numéricos.

Los algoritmos numéricos emplean la definición de los datos, el tipo de dato, variables y constantes; la utilización de procedimientos y funciones; los parámetros; las etiquetas; las instrucciones, las declaraciones y sentencias; en casos recientes los objetos y los eventos.

Todo lo anterior mencionado se anoto como si fuese estático, pero no lo es, al momento de la codificación, cada algoritmo toma personalidad y el estilo dl programador le va dando el verdadero alcance de acuerdo a la aplicación especifica; no obstante los estios de programacon como lo son: el secuencial, el modular, el orientado a eventos le van dando a la construcción de las instrucciones las características specilaes en la definición de ls instrucciones. Afortunadamente los valore se van almacenando en variables las cuales se pueden cambiar de acuerdo a otros parametros

6.4.1 LA INTERPOLACIÓN.

El siguiente algoritmo es la página del Instituto Universitaria Politécnica.

Consta de las siguientes partes:

- **Introducción a la teoría de interpolación. Interpolación polinomial, casos particulares.**
- **Polinomio de interpolación de Lagrange.**
- **Construcción del polinomio de interpolación de Lagrange: Fórmula de Lagrange.**
- **Construcción del polinomio de interpolación de Lagrange: Fórmula de Newton.**
- **Polinomio de interpolación de Lagrange: Cotas de error.**
- **Algoritmo de Aitken.**

1.- INTRODUCCIÓN A LA TEORÍA DE INTERPOLACIÓN.
INTERPOLACIÓN POLINOMIAL: CASOS PARTICULARES.

Un problema clásico de la matemática, utilizado con frecuencia en la práctica, es el de calcular el valor de una función dada en un punto cuando no se puede o no se desea, por las razones que sean, evaluar directamente la expresión de la función. Supóngase que, por ejemplo, se han efectuado experiencias que permiten conocer una función para determinados valores de su variable y se desea

conocer el valor de la función en puntos intermedios entre aquellos.
La resolución aproximada del problema consiste en encontrar una función fácil de construir y,
a la vez, fácil de evaluar y que coincida con la función objeto del problema en algunos datos que
conocemos sobre ésta. Entonces se dice que la función así construida interpola a la función respecto
a los datos.
Ahora bien, aquí hay que concretar dos cosas principalmente:
1.- Los datos que han de ser comunes a la función dada y a la que la va a interpolar.
2.- Qué tipo de función se va a usar como función interpoladora o función de interpolación.
El ejemplo más típico –que va a ser el objetivo central de este tema– es el siguiente: sea f(x)
una función cuyo valor se conoce en N + 1 puntos $x_0, x_1, ..., x_N$ y deseamos encontrar su valor
aproximado para un valor x cualquiera. Para ello intentaremos encontrar un polinomio, de grado no
mayor que N, que tome los valores $f(x_k)$ en los puntos x_k para k = 0, 1, . . . , N. Con esto último,
hemos concretado los datos comunes y qué función vamos a usar como interpoladora: un polinomio de
grado no mayor que N. Este ejemplo da lugar a lo que se conoce como problema de interpolación
Polinomial clásico.
En la situación anterior, la función interpoladora que se quiere utilizar es un polinomio. Sin
embargo, podríamos proponer, por ejemplo, que la función de interpolación fuese una función del
tipo
$g(x) = c_0Ö_0(x) + c_1Ö_1(x) + \cdots + c_NÖ_N(x)$.
Es decir, que la función de interpolación sea una combinación lineal de unas funciones dadas $Ö_0$,
$Ö_1, ..., Ö_N$, que verifique que g(xk) = (xk), k = 0, 1, . . . , N. De forma análoga, se podrían haber
impuesto otras condiciones a los datos que deben ser comunes a la función f(x) y a la función
Interpoladora g(x).
Así, además del problema de interpolación polinomial clásico, existen otros muchos casos particulares e interesantes de interpolación. De entre ellos, y dentro de los que utilizan los polinomios
como funciones de interpolación, son usuales los siguientes
a) Conocido el valor de una función f(x) y de sus N primeras derivadas sucesivas en un determinado
punto x0, encontrar un polinomio P(x), de grado menor o igual que N, tal que
P(x0) = f(x0), $P^{(k)}(x_0) = f^{(k)}(x_0)$, **k= 1, 2, . . . N.**

Este planteamiento se denomina problema de interpolación de Taylor.
b) Conocido el valor de una función f(x) y de su primera derivada $f_0(x)$, buscar un polinomio
P(x), de grado menor o igual que 2N + 1, tal que coincida él y su primera derivada con los
valores de la función f(x) y $f_0(x)$ en unos puntos x_0, x_1, ..., x_N. Es decir,
$P(x_k) = f(x_k)$, $P_0(x_k) = f_0(x_k)$, k= 0, 1, . . . N.
Este planteamiento es conocido por problema de interpolación de Hermite.
Lógicamente, en todas las situaciones que se planteen, especificada el tipo de función de interpolación
que se va a utilizar y establecidas las condiciones que han de ser comunes a la función dada
2
y a la que la va a interpolar, surgen, de manera inmediata, dos cuestiones básicas que tendrán que
contestarse: ¿Existe tal función de interpolación que verifica las condiciones impuestas? En caso de
existir, ¿es única?

2.- POLINOMIO DE INTERPOLACIÓN DE LAGRANGE.

Definición.
Sean N+1 puntos x_0, x_1, . . . , x_N, distintos dos a dos, y los correspondientes valores de una función
$y_0 = f(x_0)$, $y_1 = f(x_1)$, . . . , $y_N = f(x_N)$. Se denomina polinomio de interpolación de Lagrange
a un polinomio $P_N(x)$, de grado menor o igual a N, tal que
$P_N(x_0) = y_0$, $P_N(x_1) = y_1$, ... , $P_N(x_N) = y_N$.
Los valores x_i (i = 0, . . . , N) se denominan abscisas o nodos de la interpolación.
Teorema.
Dados N + 1 puntos x_0, x_1, . . . , x_N, distintos dos a dos, y los correspondientes valores de una
función $y_0 = f(x_0)$, $y_1 = f(x_1)$, . . . , $y_N = f(x_N)$, existe un único polinomio de interpolación de
Lagrange.
El procedimiento de cálculo directo del polinomio de interpolación de Lagrange (establecido en la
Demostración del teorema anterior) es laborioso. Buscamos otras formas de calcular más fácilmente
el polinomio de interpolación de Lagrange.

3.- CONSTRUCCIÓN DEL POLINOMIO DE INTERPOLACIÓN DE LAGRANGE: FÓRMULA DE LAGRANGE.

Para construir el polinomio $P_N(x)$, de grado menor o igual que N, que pase por los puntos
(x_0, y_0), . . . , (x_N, y_N), puede utilizarse la fórmula
$P_N(x) =$
$\sum_{k=0}^{N}$
$y_k L_{N,k}(x)$, (1)

siendo $L_{N,k}(x) =$
$(x - x_0) \dots (x - x_{k-1})(x - x_{k+1}) \dots (x - x_N)$
$(x_k - x_0) \dots (x_k - x_{k-1})(x_k - x_{k+1}) \dots (x_k - x_N)$
.
Es decir, $P_N(x)$ se expresa como una combinación lineal de N + 1 polinomios de grado N.
Un cálculo directo prueba que, para cada k fijo,
$L_{N,k}(x_i) = ($ 1 si i = k
0 si i 6= k
3
con lo que $P_N(x_i) =$
NX
k=0
$y_k L_{N,k}(x_i) = y_i$ y, puesto que el polinomio de interpolación de Lagrange es Único, la fórmula (1) nos proporciona un procedimiento válido de cálculo. A esta forma de expresión
De $P_N(x)$ se la denomina fórmula de Lagrange del polinomio de interpolación.
4.- CONSTRUCCIÓN DEL POLINOMIO DE INTERPOLACIÓN DE LAGRANGE: FÓRMULA DE NEWTON.
Los procedimientos anteriores para la construcción del polinomio interpolador de Lagrange tienen
el inconveniente de que no existe relación entre la construcción de $P_{(N-1)}(x)$ y $P_N(x)$. Ahora, vamos
a seguir un procedimiento de construcción distinto debido a Newton y que tiene como idea básica la
de construir la solución en pasos sucesivos.
Consideremos N + 1 puntos $x_0, x_1, \dots, x_N$, distintos dos a dos, y los correspondientes valores de
una función $y_0 = f(x_0), y_1 = f(x_1), \dots, y_N = f(x_N)$ para los que se quiere determinar el polinomio
de interpolación de Lagrange. Se va a proceder de la siguiente forma: Primero, se construye el
polinomio $P_0(x)$, de grado menor o igual que 0 (es decir, constante) que coincida con la función en
el punto x_0. Luego, se construye el polinomio $P_1(x)$, de grado menor o igual que 1, que coincida con
la función en x_0, x_1. A continuación, el polinomio $P_2(x)$, de grado menor o igual que 2, que coincida
con la función en x_0, x_1, x_2, y así hasta llegar al polinomio $P_N(x)$, de grado menor o igual que N,
que satisfaga todas las condiciones establecidas.
El procedimiento descrito en el párrafo anterior da lugar al siguiente esquema recursivo:
$P_0(x) = c_0$,
$P_1(x) = c_0 + c_1(x - x_0)$,
$P_2(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1)$,
$P_3(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1)$

$+c_3(x - x_0)(x - x_1)(x - x_2)$,
...
$P_N(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1)$
$+c_3(x - x_0)(x - x_1)(x - x_2)$
$+c_4(x - x_0)(x - x_1)(x - x_2)(x - x_3) + \cdots$
$+c_N(x - x_0)(x - x_1)(x - x_2)(x - x_3) \cdots (x - x_{N-1})$,
a lo largo del cual se irían imponiendo las condiciones de interpolación
$P_0(x_0) = f(x_0)$, $P_1(x_1) =$
$f(x_1)$, ..., $P_N(x_N) = f(x_N)$, para calcular sucesivamente los números $c_0, c_1, ..., c_N$.
Obsérvese que, entonces, el polinomio $P_N(x)$ se obtiene a partir de $P_{N-1}(x)$
usando la recurrencia
$P_N(x) = P_{N-1}(x) + c_N(x - x_0)(x - x_1)(x - x_2)(x - x_3) \cdots (x - x_{N-1})$,
4
y queda ahora escrito como una combinación lineal de
$(x - x_0), (x - x_0)(x - x_1), \ldots, (x - x_0)(x - x_1) \cdots (x - x_{N-1})$.
A esta forma de expresión de $P_N(x)$, esto es
$P_N(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \ldots + c_N(x - x_0)(x - x_1) \cdots (x - x_{N-1})$
se la denomina fórmula de Newton del polinomio de interpolación de
Lagrange.
Ahora, vamos a dar un medio de calcular las constantes c_i que aparecen en el proceso recursivo de
construcción del polinomio de interpolación de Lagrange. Para ello, introducimos, en primer lugar,
una definición.
Definición.
Se llama diferencia dividida de una función f(x) en un punto z_i, y escribimos $f[z_i]$, al valor de
la función f(x) en el punto z_i. Es decir,
$f[z_i] = f(z_i)$.
La diferencia dividida de f(x) en dos o más puntos distintos dos a dos, $z_1, z_2, \ldots, z_k$, se denota
por $f[z_1, z_2, \ldots, z_k]$ y se define como
$f[z_1, z_2, \ldots, z_k] =$
$f[z_2, \ldots, z_k] - f[z_1, \ldots, z_{k-1}]$
$z_k - z_1$
.
Teorema.
Dados N + 1 puntos $x_0, x_1, \ldots, x_N$, distintos dos a dos, y los correspondientes valores de una
función $y_0 = f(x_0), y_1 = f(x_1), \ldots, y_N = f(x_N)$, la fórmula de Newton del único polinomio de
interpolación de Lagrange viene dada por
$P_N(x) = c_0 + c_1(x - x_0) + \cdots + c_N(x - x_0)(x - x_1) \cdots (x - x_{N-1})$,
siendo $c_k = f[x_0, x_1, \ldots, x_k]$ para $k = 0, 1, \ldots, N$.
Teniendo en cuenta el teorema anterior, los coeficientes c_k, que permiten construir el polinomio

interpolador de Lagrange siguiendo la fórmula de Newton, son diferencias divididas de la función f(x)
las cuales pueden ser fácilmente calculadas utilizando la definición recurrente dada y disponiendo los
Cálculos como se indican en la siguiente tabla para el caso N = 4.
5

x_k	$f[x_k]$	$f[\ ,\ ]$	$f[\ ,\ ,\ ]$	$f[\ ,\ ,\ ,\ ]$	$f[\ ,\ ,\ ,\ ,\ ]$
x_0	$f[x_0]$				
x_1	$f[x_1]$	$f[x_0, x_1]$			
x_2	$f[x_2]$	$f[x_1, x_2]$	$f[x_0, x_1, x_2]$		
x_3	$f[x_3]$	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$	$f[x_0, x_1, x_2, x_3]$	
x_4	$f[x_4]$	$f[x_3, x_4]$	$f[x_2, x_3, x_4]$	$f[x_1, x_2, x_3, x_4]$	$f[x_0, x_1, x_2, x_3, x_4]$

Se observa que, en general, construida la tabla correspondiente a los datos $(x_0, y_0), ..., (x_N, y_N)$,
los elementos diagonales nos dan los coeficientes por los que hay que multiplicar los polinomios
$(x - x_0), (x - x_0)(x - x_1), . . . , (x - x_0)(x - x_1) \cdot\cdot\cdot (x - x_{N-1})$,
para formar el polinomio de interpolación de Lagrange en su forma de Newton.

5.- POLINOMIO DE INTERPOLACIÓN DE LAGRANGE: COTAS DE ERROR.

Formado, por el procedimiento que sea, el polinomio de interpolación de Lagrange, surge la
pregunta: ¿Qué diferencia hay entre su valor P(x) en un punto x dado y el verdadero valor f(x) de
la función?

Teorema.

Sea f(x) una función con N = 1 derivadas continuas en un intervalo [a, b] y tal que $f^{(N+1)}(x)$
existe en (a, b). Sean $x_0, x_1, ..., x_N$, N + 1 nodos en [a, b], distintos dos a dos, y sea $P_N(x)$ el
correspondiente polinomio interpolador de Lagrange. Entonces, para cada punto x en [a, b],

$$E_N(x) = f(x) - P_N(x) = \frac{(x - x_0)(x - x_1) \cdot\cdot\cdot (x - x_N)}{(N + 1)!} f^{(N+1)}(î)$$

para algún valor î = î(x) del intervalo (a, b).

Si, a las hipótesis del teorema anterior, se le añade que, para todo x . [a, b], $|f^{(N+1)}(x)| = K_{N+1}$
entonces

$$|E_N(x)| = |f(x) - P_N(x)| = \frac{|x - x_0||x - x_1| \cdot\cdot\cdot |x - x_N|}{(N + 1)!} K_{N+1}, .x . [a, b].$$

Además, en el caso especial de que los N + 1 nodos estén equiespaciados pueden darse resultados
más específicos para la cota del error. Es decir, sí
$a = x_0 < x_1 < \cdot\cdot\cdot < x_N = b$, siendo $x_k = x_0 + hk$, para k = 0, 1, . . . , N,

6

entonces, los términos del error correspondientes a los casos N = 1, N = 2 y N = 3 admiten las
siguientes cotas

$|E_1 x)| = |f(x) - P_1(x)| =$
h_2
8
K_2, válida para x . [x_0, x_1],

$|E_2 x)| = |f(x) - P_2(x)| =$
h_3
9v3
K_3, válida para x . [x_0, x_2],

$|E_3 x)| = |f(x) - P_3(x)| =$
h_4
24
K_4, válida para x . [x_0, x_3].

Por otro lado, también es conveniente plantearse si el error de interpolación $f(x) - P_N(x)$ puede
Hacerse arbitrariamente pequeño incrementando el grado del polinomio. Más precisamente: dada
una función f(x) definida en un intervalo [a, b], elijamos un punto $x^{(0)}_0$ e interpolemos en él por un
polinomio constante $P_0(x)$; elijamos dos puntos distintos entre sí $x^{(1)}_0$, $x^{(1)}_1$ e interpolemos en ellos
por una recta $P_1(x)$; elijamos tres puntos, distintos dos a dos, $x^{(2)}_0$, $x^{(2)}_1$, $x^{(2)}_2$ e interpolemos por
una parábola cuadrática, etc. ¿Será cierto que $\lim_{N.8} P_N(x) = f(x)$? La respuesta es, en general,
negativa, con la conclusión de que incrementar el grado de los polinomios de interpolación no siempre
es aconsejable.

6.- POLINOMIO DE INTERPOLACIÓN DE LAGRANGE: ALGORITMO DE AITKEN.

En esta sección vamos a dar un procedimiento recurrente para calcular el polinomio de interpolación
de Lagrange para el conjunto de datos $\{(x_k, y_k\}^N_{k=0}$. Se basa en un resultado que se expondrá
a continuación.

Previamente, hay que indicar que si consideramos el conjunto de datos $\{(x_k, y_k)\}^N_{k=0}$, el conjunto
de nodos $A = \{x_0, x_1, . . . , x_N\}$ y un subconjunto no vacío B de A, denotaremos por $P_B(x)$ el polinomio

de intepolación de Lagrange correspondiente a los nodos que forman el conjunto B.

Teorema. (Lema de Aitken)

Sean B y C dos subcojuntos no vacíos de A, con todos sus nodos comunes excepto x_j. B que no está en C y x_i. C que no está en B. Entonces,

$$P_{B.C}(x) = \frac{(x_i - x)P_B(x) - (x_j - x)P_C(x)}{x_i - x_j}.$$

Del teorema se desprende que es posible construir el polinomio que interpola a una función en N+1

7

nodos utilizando dos polinomios que la interpolan en N nodos. Por ejemplo, para conocer el polinomio de interpolación correspondiente a los nodos $\{x_0, x_1, x_2, x_3\}$ podemos utilizar los polinomios asociados a $\{x_0, x_1, x_2\}$, $\{x_0, x_1, x_3\}$, y para éstos los de $\{x_0, x_1\}$, $\{x_0, x_2\}$ y $\{x_0, x_1\}$, $\{x_0, x_3\}$, respectivamente.

Por ello, es conveniente disponer los cálculos como aparece en la siguiente tabla

x_0	$x_0 - x$	y_0			
x_1	$x_1 - x$	y_1	$P_{\{x_0,x_1\}}(x)$		
x_2	$x_2 - x$	y_2	$P_{\{x_0,x_2\}}(x)$	$P_{\{x_0,x_1,x_2\}}(x)$	
x_3	$x_3 - x$	y_3	$P_{\{x_0,x_3\}}(x)$	$P_{\{x_0,x_1,x_3\}}(x)$	$P_{\{x_0,x_1,x_2,x_3\}}(x)$

Obsérvese, en la tabla, que a partir de y_0 y de cada uno de los y_1, y_2, y_3 se construye, mediante la fórmula que nos proporciona el lema de Aitken, la columna de los $P_{\{x_0,x_i\}}(x)$. Después, a partir de $P_{\{x_0,x_1\}}(x)$ y de cada uno de los elementos de su columna se construye la siguiente columna, etc.

Para esto viene bien poner una columna previa con los $x_k - x$ ya que para obtener, por ejemplo, $P_{\{x_0,x_1,x_3\}}(x)$ necesitamos manejar $P_{\{x_0,x_1\}}(x)$, $x_1 - x$, $P_{\{x_0,x_3\}}(x)$ y $x_3 - x$.

Si se desea calcular únicamente el valor concreto de $P_{\{x_0,x_1,x_2,x_3\}}(x)$ en un punto $x = c$, basta sustituir x por c en todos los cálculos hechos en la tabla anterior. Al final se obtendrá $P_{\{x_0,x_1,x_2,x_3\}}(c)$; es decir, el valor deseado.

Por último hay que indicar que, lógicamente, la elección que se ha hecho de B y C para aplicar el lema de Aitken no es única. La que aquí se ha elegido es la que da lugar al algoritmo de Aitken:

$$P_{\{x_0,x_1,\ldots,x_{i-1},x_i\}}(x) = \frac{(x_i - x)P_{\{x_0,x_1,\ldots,x_{i-1}\}}(x) - (x_{i-1} - x)P_{\{x_0,x_1,\ldots,x_{i-2},x_i\}}(x)}{x_i - x_{i-1}}$$

.
8

6.4.2 CALCULOS MATRICIALES.

Algoritmo utilizando los Indices de Welch

Una aproximación tentativa en tratar de utilizar los resultados de trabajos de autores anteriores como los del Dr. Harold V. McIntosh y Jose Manuel Gómez Soto; este algoritmo consiste en aprovechar los conceptos de multiplicidad uniforme e índices de Welch para la construcción de las posibles matrices de evolución de ACLR; por ejemplo, para un ACLR(5,h) se tiene la siguiente ``plantilla'' para generar las matrices de evolución:

	0	1	2	3	4
0	0				
1		1			
2			2		
3				3	
4	4	4	4	4	4

Tabla: **Plantilla para generar matrices de evolución de ACLR(5,h).**
El resto de los lugares vacíos se llenaban con todas permutaciones de estados posibles de tal manera que un elemento de cada columna fuera diferente al resto, esto es ya que al tener cinco nodos el diagrama de de Bruijn asociado, el valor de LMR=5, por lo que se fija a M=1 y L=1, lo que implica que no debe existir dos elementos iguales por columna.
La razón por la cual el último renglón de la matriz estaba lleno del estado 4 era para tener un estado en el cual fuera seguro que el conjunto de todas sus extensiones compatibles derechas tuviera una cardinalidad igual a R, cumpliendo para ese estado que LMR=5, una vez llena la matriz, se generaba una evolución de este ACL y se observaba si cada posible vecindad de longitud 2 tenía un único estado para el cual evolucionar hacia atrás, de cumplir ésto, el ACL se tomaba como reversible, de no ser así se verificaban si las vecindades de longitud 3 si cumplían con tener un único estado en el pasado; este proceso continuaba hasta revisar vecindades de longitud 4.

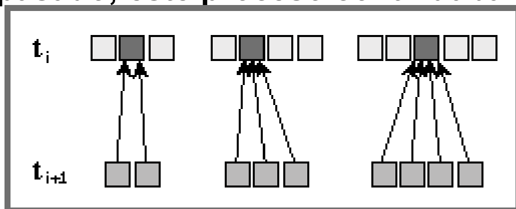


Figura: **Verificación de que cada cadena tenga un único elemento en común en el pasado para cadenas de longitud 2, 3 y 4.**
La ventaja de este planteamiento es que ya no hace la revisión de cada posible regla de evolución para un ACL(k,h) sino que solo analiza aquellas matrices de evolución en donde se tenga una diagonal principal donde cada estado sea diferente al resto y un renglón formado por un mismo valor para asegurar que la cardinalidad del conjunto de extensiones compatibles por la

derecha del mismo fuera R, con lo que el tiempo de computación que toma hacer el cálculo se reduce de manera significativa.

BIBLIOGRAFÍA.

Diccionario El Pequeño Larousse Ilustrado, Edit. Larousse, edición 1999.