“UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO”

FACULTAD DE ESTUDIOS SUPERIORES

CAMPUS ARAGÓN.

“LENGUAJES DE PROGRAMACIÓN”.

Trabajo de Examen General de Conocimientos

Para obtener el título de:

INGENIERO EN COMPUTACIÓN.

Presenta:

VANESSA ALEJANDRA CAMACHO VÁZQUEZ.

Director de Titulación:

M. en C. Marcelo Pérez Medel.

México, D.F. 2007

ÍNDICE

LENGUAJES DE PROGRAMACIÓN

Pág.

1- INTRODUCCIÓN A LOS LENGUAJES DE PROGRAMACIÓN. 3

1.1- Concepto de Lenguajes de Programación. 4

1.2- Clasificación y generaciones de los Lenguajes de Programación. 4

1.3- Principios de los Lenguajes de Programación. 7

2 ASPECTOS DE DISEÑO DE LOS LENGUAJES DE PROGRAMACIÓN. 13

2.1- Organización y operación de una computadora. 14

2.2- Máquinas virtuales. 19

2.3- Consecuencias para la traducción de los distintos momentos de ligado. 20

2.4- Paradigmas de los Lenguajes de Programación. 21

3- ASPECTOS DE TRADUCCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN.

3.1- Sintaxis de los lenguajes de programación. 26

3.2- Etapas de traducción. 27

4- TIPOS Y OBJETOS. 31

4.1- Propiedades. 32

4.2- Tipos Elementales. 33

4.3- Tipos Estructurados. 39

5- ENCAPSULACIÓN. 61
5.1- Tipos abstractos de datos. 62
5.2- Encapsulación mediante subprogramas. 64
5.3- Definición de tipos. 69

6- HERENCIA. 74
6.1- Concepto, beneficios y facetas de la Herencia. 75
6.2- Mecanismos y clasificación de la Herencia. 78
6.3- Objetos y mensajes. 80
6.4- Metaclases y Herencia Múltiple. 81

7- CONTROL DE SECUENCIA. 84
7.1- Secuencia de expresiones aritméticas y no aritméticas. 85
7.2- Control de secuencia entre enunciados. 86

8- CONTROL EN SUBPROGRAMAS. 91

9-ADMINISTRACIÓN DE ALMACENAMIENTO. 108

CONCLUSIONES 117

BIBLIOGRAFÍA Y REFERENCIAS 120

CAPÍTULO 1:

“INTRODUCCIÓN A LOS LENGUAJES DE PROGRAMACIÓN”.

1.1- CONCEPTO DE LENGUAJES DE PROGRAMACIÓN.

Lenguaje de Programación es cualquier lenguaje artificial que puede especificar y generar la comunicación con la computadora, a través de la definición adecuada de una secuencia de instrucciones (software) que serán interpretadas y ejecutadas por ésta (hardware).

Los Lenguajes de Programación intentan conservar una similitud con el lenguaje humano para ser más naturales así, permiten a los programadores especificar de forma precisa los datos sobre los que se va a actuar, su almacenamiento, transmisión y demás acciones a realizar.

Los Lenguajes de Programación tienen un léxico (vocabulario) que es un conjunto de símbolos permitidos, una sintaxis (instrucciones con palabras reservadas y reglas - gramática) que indica como realizar las construcciones del lenguaje y una semántica que son reglas que permiten determinar el significado de cualquier construcción del lenguaje (el programador sabe lo que hace el programa).

1.1- CLASIFICACIÓN Y GENERACIONES DE LOS LENGUAJES DE PROGRAMACIÓN.

Los Lenguajes de Programación pueden clasificarse de acuerdo a su semejanza con el Lenguaje Máquina o el Lenguaje Humano (generalmente inglés). Los lenguajes que tiene mayor semejanza con el Lenguaje Humano se les llama “Lenguajes de Alto Nivel”, mientras que los lenguajes más parecidos al Lenguaje de Máquina son conocidos como “Lenguajes de Bajo Nivel”.

TABLA 1: “GENERACIONES DE LOS LENGUAJES DE PROGRAMACIÓN”

Generación	Nombre	Particularidad
Primera	De máquina	Específico para cada microprocesador, uso de código binario
Segunda	Ensamblador	Uso de nemotécnicos que abstraen del lenguaje máquina.
Tercera	De procedimientos	Lenguajes estructurados: comandos cercanos al lenguaje común.
Cuarta	Orientados a procesos	Programas orientados a problemas específicos.
Quinta	Natural	Incluye inteligencia artificial y sistemas expertos.

LENGUAJES DE BAJO NIVEL. Los microprocesadores necesitan que se les envíen instrucciones binarias compuestas de series de unos y ceros, espaciadas en el tiempo de una forma determinada. Esta secuencia de señales se denomina “Lenguaje de Máquina” y representa normalmente datos y números así, como instrucciones para manipularlos. Para hacer más fácil el manejo del Código Máquina, se dio a cada instrucción un mnemónico. Ejemplo: STORE (Almacenar), ADD (Adicionar), SUB (Restar), MUL (Multiplicar), DIV (dividir), JUMP (Saltar), etc. Tal abstracción da como resultado el “Lenguaje Ensamblador” basado en mnemónicos y que es específico para cada microprocesador. Está constituido por un Editor (especie de procesador de palabras) donde se crea un Archivo Fuente con las instrucciones que la computadora va a ejecutar. El Código Fuente es traducido al Lenguaje Máquina mediante Compiladores (programas traductores). El Lenguaje Ensamblador no se utiliza para programas grandes porque siguen siendo altamente detallados (instrucciones básicas), sólo se usa en los casos donde la velocidad es clave (en la programación de juegos de video) y para afinar partes importantes de los programas que son escritos en lenguajes superiores.

Además, los programas escritos en un Bajo Nivel son específicos para cada procesador, ya que si se quiere ejecutar el programa en otra máquina de arquitectura diferente es necesario volver a escribir el programa desde el principio.

LENGUAJES DE ALTO NIVEL. Sus instrucciones son independientes de la máquina y más fáciles de aprender porque están formadas por elementos de Lenguajes Naturales (del inglés). Los lenguajes de programación buscan facilitar el proceso de programación y también necesitan de un Editor (especie de procesador de palabras), donde se crea un Archivo Fuente que debe ser traducido al Lenguaje Máquina mediante Programas Traductores o Compiladores.

Los “Lenguajes de Procedimientos” tienen programas capaces de soportar programación estructurada, porque pueden usar estructuras de programación específicas para ramificaciones y ciclos en el flujo del programa. Ejemplos:
-FORTRAN (Traductor de Fórmulas). Utilizado en equipos de cómputo para Investigación y Educación con Programas Matemáticos y de Ingeniería.
-COBOL (Lenguaje Simbólico de Programación). Para Aplicaciones Comerciales y de Gestión. Fue un lenguaje estandarizado en negocios, pero perdió seguidores.
-BASIC (Código de Instrucción Simbólica Universal para Principiantes). Lenguaje de enseñanza, no es un lenguaje viable para Aplicaciones Comerciales porque no posee un amplio repertorio de herramientas. Además, sus compiladores no producen Archivos Ejecutables tan compactos, rápidos y eficientes como otros.
-PASCAL (introducido por Niklaus Wirth - 1971). Excelente lenguaje de enseñanza y fácil para implementar algoritmos. Su compilador es estricto con el uso de programación estructurada y asegura que los errores sean señalados al principio.
-C (desarrollado por B. Kernighan y D. Ritchie – 60’s). Consigue realizar casi cualquier cosa en una computadora, es mejor lenguaje de integrar con el sistema operativo UNIX para que los usuarios puedan hacer modificaciones y adaptaciones con facilidad. Su ejecutable es rápido y eficiente.
-C++ (80’s). Implica aprender C, además de la Programación Orientada a Objetos y su aplicación mediante el C++.

-Java (desarrollado por Sun Microsistema – 1991). Ambiente de Programación que permite crear Sistemas Interactivos y Dinámicos (denominados applets) para las páginas Web. Gana popularidad por la demanda de aplicaciones para Internet.

Los “Lenguajes de Propósito Especial” permiten la creación de prototipos de una aplicación rápidamente, dan idea del aspecto y funcionamiento de la aplicación antes que el código sea terminado. Al principio del proceso se puede aportar retroalimentación en la estructura y diseño, pero se pierde flexibilidad. Muchos tienen capacidad para Bases de Datos y permiten crear programas que sirvan de enlace con las mismas. Los programas incluyen formas y cuadros para introducir datos, así como para solicitar reportes de información. Ahorran tiempo porque el código requerido para “conectar” los cuadros de diálogo y las formas se genera automáticamente. Ejemplos:
-Visual BASIC o Visual C. Soportan características y métodos orientados a objetos que permiten crear Programas en un Ambiente Visual. Facilitan el desarrollo de la Interfaz de Usuario y aceleran la creación de programas.
.Macromedia Director. Ambientes de Desarrollo Visuales, mucho código se escribe automáticamente y la mayoría incluyen sus propios lenguajes de escritura para un control extra sobre el producto final. Crean Multimedia, capacitación basada en computadora, páginas Web, etc.

Los “Lenguajes Naturales” incluyen la Inteligencia Artificial y Sistemas Expertos. Su objetivo es “pensar” y anticipar las necesidades de los usuarios. Continúan desarrollándose por su complejidad, pronto serán capaces de tomar como entrada “hechos” y luego usarán un procesamiento de datos que formule una respuesta adecuada, de modo similar a como responden los humanos. Ejemplo: Prolog.

1.2- PRINCIPIOS DE LOS LENGUAJES DE PROGRAMACIÓN

ABSTRACCIÓN. Se debe evitar que un proceso se inicie más de una vez. Una abstracción es una representación de un objeto sólo con los atributos importantes

del objeto original, ignora atributos irrelevantes para el propósito. El Lenguaje de Programación ofrece 2 tipos de Abstracción: las “características” del lenguaje que son un conjunto útil de abstracciones y “facilidades” que ayudan al programador a construir sus propias abstracciones. Subprogramas, bibliotecas de subprogramas, definiciones de tipo y paquetes, son facilidades proporcionadas por diferentes lenguajes para soportar las abstracciones definidas por el programador.
Una abstracción es una herramienta intelectual, permite trabajar con los conceptos independientemente de las particularidades de estos y los separa de un sistema para implementarlo después. La “Abstracción Funcional” (de procedimientos) aplica un conjunto de entradas a un conjunto de salidas, pudiendo en algún momento modificar algunas de las entradas. Una “Abstracción de Datos” está formada por un conjunto de objetos y un conjunto de operaciones (abstracciones funcionales) que manipulan esos objetos.

AUTOMATIZACIÓN. Al programar, se debe cuidar el manejo de los apuntadores para no provocar problemas con la memoria, también actividades muy repetitivas y tediosas se prestan a que hayan posibles errores. La Automatización propone que estas actividades se implementen con mecanismos automatizados para evitar lo más posible los problemas que puedan causar. Ejemplo: uso de una Máquina Virtual al programar en Ensamblador, porque podríamos obtener por error la dirección del BIOS y guardarla en una variable la cual pudiéramos utilizar para guardar algunos datos en un registro, por consecuencia borraríamos el BIOS.
DEFENSA EN PROFUNDIDAD. Tener una serie de defensas tal que si un error no es detectado por uno, éste probablemente sea detectado por otro. Los errores deben ser detectados por el compilador, si un mecanismo no es capaz de detectar un error es necesario implementar otro que lo detecte, pero nunca ignorarlo.

OCULTACIÓN DE INFORMACIÓN. Los módulos de un sistema deben diseñarse de modo que la información contenida en ellos sea inaccesible a todos aquellos módulos que no la necesiten. Establece las restricciones de acceso a los detalles

internos de cualquier módulo y a cualquier estructura de datos utilizada localmente en el mismo. El módulo oculta su funcionalidad interna y se comunica con otros mediante interfaces. El lenguaje debe permitir módulos diseñados para que el usuario y el desarrollador del lenguaje tengan toda la información necesaria para usar el módulo correctamente y nada más. Este ocultamiento de información se puede ver en varios casos. Ejemplo: uso de software mediante interfaz, no importa cómo el programa maneja internamente la información sino el resultado obtenido. “Información Oculta” es un principio central en el diseño de abstracciones definidas por el programador, tales como subprogramas y tipos de datos nuevos: cada componente del programa debe ocultar tanta información como sea posible al usuario del componente. Ejemplo: función de la raíz cuadrada proporcionada por el lenguaje, es una exitosa abstracción que oculta al usuario los detalles de la representación del número y el algoritmo de computación para resolverla.
Cuando la información es encapsulada en una abstracción, significa que el usuario de la abstracción no necesita saber la información oculta para usarla, y no es permitido usar o manipular la información oculta, aún cuando lo desee.

ETIQUETADO (labeling). Evitar secuencias arbitrarias más que unos pocos items largos. No requiere que el usuario conozca la posición absoluta en una lista. En lugar de esto, se debe asociar una etiqueta significativa con cada item (pedazo de código que realiza una tarea determinada) y dejar que el item ocurra en cualquier orden. El etiquetado ayuda para realizar saltos dentro de un programa, porque a veces no es factible poner todo el código seguido y que se ejecute en forma lineal. Las etiquetas ayudan a no estar guardando o calculando la localidad absoluta de memoria a donde queremos que continúe la ejecución del programa, sino que esta continuará donde encuentre la etiqueta indicada. Así, se logra facilitar la lectura de los programas y se reutiliza código escribiéndolo sólo una vez comportándose esto como una función.

LOCALIZACIÓN DE COSTOS. Los usuarios sólo deben pagar por lo que usan, deben evitar costos distribuidos. Un buen lenguaje de programación permite al

programador determinar los costos, elegir que parte del lenguaje usar y pagar menos por él. La determinación se basa en el Costo de Ejecución, Costo de Compilación, Costo de Creación de Programas, Costo de Uso - Prueba y Costo de Mantenimiento.

INTERFACE MANIFIESTA. Las interfaces deben ser aparentes (manifiestas) en la sintaxis, reconocibles fácilmente, lo más visibles, claras y descriptivas posibles. La interface con un objeto es su protocolo, el conjunto de mensajes a los cuales el objeto responde.

ORTOGONALIDAD (Orthogonality). Las funciones independientes deben ser controladas por mecanismos independientes. La ortogonalidad se da por las rectas que se interceptan y forman ángulos rectos entre sí (como los ejes en un plano cartesiano), son independientes. El lenguaje de programación tiene funciones independientes, las cuales van en el mismo camino en cualquier circunstancia. Trata de juntar varias funciones (independientes) en distintos grupos para después poder hacer referencia a estas con mayor facilidad. La ortogonalidad ayuda a reducir el número de códigos que el programador debe aprender.
PORTABILIDAD. Un lenguaje es portable si un programa puede ser compilado en diferentes computadoras y ejecutable en diferente hardware. Ejemplo: permitir el uso de librerías para definir los tipos de datos.

PRESERVACIÓN DE LA INFORMACIÓN. El lenguaje permite la representación de información que el usuario debe saber y que el compilador puede necesitar. Proporciona las herramientas necesarias para que tanto el usuario del lenguaje como el desarrollador del mismo puedan realizar su trabajo correctamente.

REGULARIDAD (Regularity). Reglas regulares, sin excepción son fáciles de aprender, usar, describir e implementar. También se conoce como Uniformidad (uniformity). Quiere decir que notaciones similares deben verse y comportarse parecido. Se enfoca a definir las fronteras de una instrucción o un conjunto de

ellas agrupadas en un bloque de código para que el lenguaje conozca donde comienza y en donde termina cada una y así poder interpretarlas correctamente.

SEGURIDAD. Ningún programa que viole la definición del lenguaje o su propia estructura propuesta debe escapar a la detección.

SIMPLICIDAD. Un lenguaje debe ser lo más simple posible, con un número mínimo de conceptos y reglas simples para su combinación. También debe esforzarse en la Simplicidad Sintáctica y Semántica. La “Simplicidad Semántica” implica que el lenguaje contiene un mínimo número de conceptos y estructuras, los conceptos son naturales, rápidamente aprendidos y fácilmente entendidos. La “Simplicidad Sintáctica” requiere que la sintaxis represente cada concepto en una única forma y que ésta interpretación sea muy legible. El lenguaje debe ser una ayuda para el programador antes de que alcance el estado real de codificación en programación. Debe darle un conjunto de conceptos claro, simple y unificado para que pueda usarlos como primarios en el desarrollo de lenguajes. Para ello es deseable tener un número mínimo de conceptos diferentes, con las reglas de su combinación lo más simples y regulares posibles. Esta claridad semántica y de conceptos es el factor determinante del valor de un lenguaje.

ESTRUCTURA (structure). La estructura estática de un programa debe corresponder con la estructura dinámica de los cómputos correspondientes. Debe ser posible visualizar el comportamiento dinámico de un programa fácilmente desde su forma estática. El comportamiento dinámico es el comportamiento que tiene el programa en tiempo de ejecución y su forma estática es el programa escrito. Entender el comportamiento del programa al ser ejecutado simplemente con ver el código y seguirlo instrucción por instrucción, sirve para detectar errores.

CONSISTENCIA SINTÁCTICA. Las cosas similares deben lucir similares y las cosas diferentes deben lucir diferentes. Todas las palabras reservadas y las

funciones definidas en el lenguaje deben evitar confundir al programador con la sintaxis de sus declaraciones.

UNO, CERO E INFINITO (ninguno). Los únicos números razonables son uno, cero e infinito. No es conveniente diseñar un lenguaje de programación que requiera que el programador memorice valores diferentes a estos tres valores. Ejemplo: para denotar valor vacío se usa cero, la violación seria representar el valor con otro número que no sea el cero, ya que obligará a que el aprendizaje del lenguaje dependa de la memorización de diferentes valores.

CAPÍTULO 2:

“ASPECTOS DE DISEÑO DE LOS LENGUAJES DE PROGRAMACIÓN”.

Los primeros Lenguajes de Programación se ejecutaron en equipos costosos, producían un Código de Máquina eficiente y la escritura de los programas era difícil (Fortran y LISP). Actualmente, los equipos son de bajo costo y permiten el desarrollo de programas que son fáciles de escribir correctamente aunque se ejecuten con lentitud algo mayor (C++, Java). Para desarrollar un Lenguaje de Programación existen 3 influencias que afectan su diseño: La “Computadora Subyacente” que es donde se van a ejecutar los programas escritos en el lenguaje, el “Modelo de Ejecución” ó “Computadora Virtual” que apoya a ese lenguaje en el equipo real y el “Modelo de Computación” que el lenguaje implementa.

2.1- ORGANIZACIÓN Y OPERACIÓN DE UNA COMPUTADORA.

Una computadora es un conjunto integrado de algoritmos y estructuras de datos, capaz de almacenar y ejecutar programas. Podemos construir una “Computadora Real” o “Computadora de Hardware” como un dispositivo físico real utilizando circuitos integrados, tarjetas, etc. También se puede construir una “Computadora Simulada por Software” por medio de programas que se ejecuten en otra computadora. Un Lenguaje de Programación se implementa construyendo un Traductor, el cual traduce los programas que están en el lenguaje a programas en Lenguaje de Máquina que pueden ser ejecutados directamente por alguna computadora. Un ejemplo de computadora es la figura que viene a continuación:

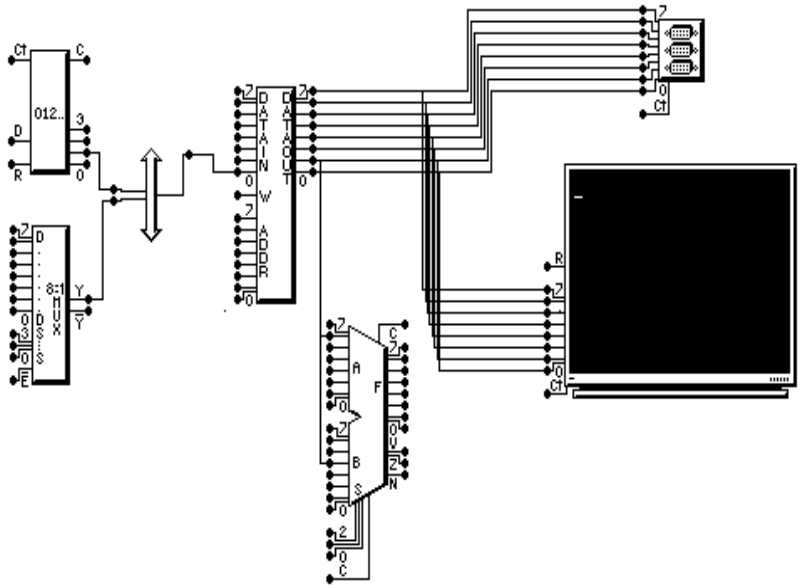


FIG 1: Organización de la Computadora.

COMPONENTES DE UNA COMPUTADORA. La computadora consta de 6 componentes fundamentales que corresponden estrechamente a los aspectos principales de un Lenguaje de Programación y son:

-DATOS. Elementos de información integrados (enteros, reales o cadenas de caracteres) que se manipulan con operaciones primitivas de hardware. Los principales Medios de Almacenamiento de Datos son la Memoria Principal organizada como una secuencia lineal de bits subdividida en palabras de longitud fija, los Registros Internos de Alta Velocidad, compuestos de secuencias de bits con sub-campos especiales que son directamente accesibles, la Memoria Caché y los Archivos Externos.

-OPERACIONES. Conjunto de operaciones primitivas inter-construidas y apareadas una a una con los Códigos de Operación que pueden aparecer en las instrucciones de Lenguaje Máquina. Un conjunto típico incluiría primitivas para aritmética sobre cada tipo numérico de datos (suma, resta, etc.), primitivas para probar propiedades de un conjunto de datos (cero, positivo, negativo), primitivas para controlar dispositivos.

-CONTROL DE SECUENCIA. Mecanismos que controlan el orden en el que se van a ejecutar las operaciones primitivas. La instrucción siguiente que se debe

ejecutar en cualquier punto durante la ejecución de un programa en lenguaje máquina está determinada por el contenido de un registro de direcciones de programa, el cual contiene la dirección de memoria de la próxima instrucción.

-ACCESO A DATOS. Recursos para incorporar algún medio para designar operandos y un mecanismo para recuperar operandos de un designador de operandos dado. De manera similar, el resultado de una operación primitiva se debe guardar en alguna localidad designada. El esquema convencional consiste en asociar direcciones de enteros con localidades de memoria y suministrar operaciones para recuperar el contenido de una localidad dada.

-GESTIÓN DE ALMACENAMIENTO. El Sistema Operativo usa Multiprogramación para acelerar el desequilibrio entre acceso a datos externos y el procesador central. La computadora cuando espera muchos milisegundos para leer los datos, también ejecuta otro programa. A fin de que muchos programas puedan residir conjuntamente en la memoria al mismo tiempo, es común incluir recursos para paginación o reubicación dinámica de programas directamente en el hardware. Existen algoritmos de paginación que intentan prever cuales direcciones de programas y datos tienen más probabilidades de ser utilizados en futuro cercano, con el propósito de que el hardware pueda ponerlas a disposición del procesador central. La Memoria Caché es un pequeño almacén de datos de alta velocidad que se encuentra entre la memoria principal y el procesador central, sirve para acelerar el desequilibrio entre estos, contiene los datos e instrucciones que el procesador central ha usado más recientemente, también incluye los datos e instrucciones que es más probable que se van a necesitar en el futuro cercano.

-ENTORNO DE OPERACIÓN. Conjunto de dispositivos periféricos de almacenamiento y Entrada - Salida. Estos representan el mundo exterior para la computadora y la comunicación con ella debe tener lugar a través de estos. Ejemplo: discos magnéticos, CD-ROM, cintas, etc.

ARQUITECTURAS ALTERNATIVAS DE COMPUTADORA. La “Arquitectura Von Neumann” desarrollada por John Von Neumann como parte de la ENIAC. Consta de una UCP (Unidad Central de Proceso), compuesta de las operaciones

primitivas, el control de secuencia y registros internos para guardar los resultados de las operaciones primitivas, una memoria principal más grande y un proceso para recuperar y guardar palabras de datos entre la UCP y la memoria más grande. Esta arquitectura genera desigualdad de velocidades entre los datos externos y la alta velocidad de los registros de la UCP. La alternativa de este problema es la “Arquitectura de Multiprocesadores” que usa múltiples UCP en un sistema dado. El sistema operativo ejecuta distintos programas en UCP diferentes en el sistema y el rendimiento global mejora.

ESTADOS DE COMPUTADORAS. Entender la computadora se basa en conocer su Operación Estática y Dinámica. El concepto de Estado de Computadora es un medio conveniente para visualizar el Comportamiento Dinámico de ésta. El proceso de ejecución de programas por la computadora tiene lugar a través de una serie de estados, cada uno definido por el contenido de la memoria, los registros internos y los almacenes externos en algún punto durante la ejecución. El Estado Inicial de la computadora es el contenido inicial de estas áreas de almacenamiento. Cada paso en la ejecución del programa, transforma el estado existente en un nuevo estado a través de la modificación del contenido de una de estas áreas de almacenamiento o más. Transición de Estado es la transformación de éste. Al concluir la ejecución del programa, se define el Estado Final por el contenido final de estas áreas de almacenamiento.

FIRMWARE. Cualquier algoritmo o estructura de datos definida con precisión se puede producir en hardware. Una alternativa común a la producción estricta en hardware de una computadora es la “Computadora de Firmware”, simulada por un microprograma que se ejecuta en una computadora microprogramable de hardware especial. Su lenguaje de máquina consiste en simples transferencias de datos entre memoria principal y registros internos de alta velocidad, entre los registros mismos, y desde los registros a otros registros a través de procesadores como sumadores y multiplicadores. El microprograma simula la operación de la

computadora deseada en la computadora microprogramable anfitrión. El microprograma reside generalmente en memoria de sólo lectura.

TRADUCTORES Y COMPUTADORAS SIMULADAS POR SOFTWARE. La programación se realiza más en Lenguajes de Alto Nivel alejados del Lenguaje de Máquina mismo del hardware. Las soluciones para implementar son la Traducción y la Simulación de Software.

TRADUCCIÓN (compilación).

-Traductor. Procesador de lenguajes que acepta programas en cierto lenguaje fuente (que puede ser de alto o bajo nivel) como entrada y produce programas funcionalmente equivalentes en otro lenguaje objeto.

-Ensamblador. 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.

-Compilador. Traductor cuyo Lenguaje Fuente es un Lenguaje de Alto Nivel y cuyo Lenguaje Objeto se aproxima al Lenguaje Máquina de una Computadora Real, ya sea que se trate de un Lenguaje Ensamblador o alguna variedad de Lenguaje Máquina.

-Cargador. Traductor cuya entrada es un Lenguaje Objeto y la entrada es un programa en Lenguaje Máquina en manera reubicable; las modificaciones que realiza son a tablas de datos que especifican las direcciones de memoria, donde el programa necesita estar para ser ejecutable.

-Preprocesador: Editor de texto, toma como entrada una forma ampliada de un Lenguaje Fuente y su salida es una forma estándar del mismo Lenguaje Fuente.

SIMULACIÓN DE SOFTWARE (interpretación). En vez de traducir los programas de Alto Nivel a programas equivalentes en Lenguaje Máquina, se pueden simular a través de programas ejecutados en otra Computadora Anfitrión, cuyo Lenguaje

Máquina sea de Alto Nivel. Esto se construye con software que se ejecuta en la Computadora Anfitrión, la computadora con Lenguaje de Alto Nivel que de otra manera se podría haber construido en hardware.
La traducción y la simulación dan ventajas diferentes. La traducción cuando tiene códigos referentes a ciclos que se repitan gran cantidad de veces, los ejecuta muchas veces, pero se traducen sólo una a diferencia de los simulados que tienen que decodificarlo la misma cantidad de veces que se ejecute. La simulación en caso de un error proporciona más información acerca de donde fue la falla que en un Código Objeto traducido.

2.2- COMPUTADORAS VIRTUALES

Computadora virtual son las estructuras de control y algoritmos de un lenguaje que se emplean durante el Tiempo de Ejecución de un programa. Su Lenguaje de Máquina es el Programa Ejecutable que produce el traductor del lenguaje, el cual puede adoptar la forma de Código de Máquina auténtico si el lenguaje se compila o puede tener alguna estructura arbitraria si el lenguaje se interpreta.

SINTAXIS Y SEMÁNTICA. Es el aspecto que ofrece el programa. Las reglas de sintaxis para un Lenguaje de Programación es decir cómo se escriben los enunciados, declaraciones y otras construcciones del lenguaje. La semántica de un Lenguaje de Programación es el significado que se da a las diversas construcciones sintácticas. Típico es describir la sintaxis de una construcción del lenguaje con la notación BNF (Backus Naur Form) y luego se da también la semántica para esa construcción.

COMPUTADORAS VIRTUALES E IMPLANTACIÓN DE LENGUAJES. Cuando se implementa un Lenguaje de Programación en una computadora particular, el Implementador determina primero la Computadora Virtual que representa una interpretación de la semántica del lenguaje y luego la construye a partir de los

elementos de hardware y software que suministra la Computadora Subyacente, así como los costos de uso del lenguaje.

JERARQUÍAS DE COMPUTADORAS VIRTUALES. La Computadora Virtual utilizada por el programador para hacer un programa en algún Lenguaje de Alto Nivel, esta formada por una jerarquía de Computadoras Virtuales. Hasta abajo está una Computadora de Hardware Real, que se transforma sucesivamente a través de capas de software (o microprogramas) en una Computadora Virtual que puede ser radicalmente distinta. El segundo nivel de Computadora Virtual (o tercero si un microprograma forma el segundo nivel) está definido por el Sistema Operativo (compleja colección de rutinas).
El Sistema Operativo provee simulaciones de un cierto número de operaciones y estructuras de datos nuevas que no proporciona directamente el hardware, por ejemplo, estructuras de archivos externos y primitivas de administración de archivos. La Computadora Virtual definida por el Sistema Operativo es la que está disponible para el implementador de un Lenguaje de Alto Nivel. El Implementador, también suministra un Traductor para traducir programas de usuario al Lenguaje de Máquina de la Computadora Virtual definida por el Lenguaje de Alto Nivel. Los programas que el programador construye añaden un nivel más a la jerarquía. El Lenguaje de Máquina se compone de los datos de entrada para estos programas. Lo que es programa en un contexto es probable que se convierta en datos en otro.

2.3- CONSECUENCIAS PARA LA TRADUCCIÓN DE LOS DISTINTOS MOMENTOS DE LIGADO

ENLACE Y TIEMPO DE ENLACE. El Enlace es la elección de una propiedad para un elemento de programa, de entre un conjunto de propiedades posibles. El Tiempo de Enlace es el momento durante el procesamiento del programa en el que se hace esta elección. Las clases de tiempo de enlace son:

-Tiempo de ejecución. Incluye enlaces a variables con sus valores y a variables con localidades particulares de almacenamiento. Se divide en 2 categorías: al entrar a un subprograma o bloque y en puntos arbitrarios durante la ejecución.
-Tiempo de traducción (tiempo de compilación): Enlaces elegidos por el programador son nombres de variables, tipos para las variables, estructuras de enunciados de programa; Enlaces elegidos por el traductor que es cómo se guardan los arreglos y cómo se crean descriptores para los arreglos; Enlaces elegidos por el cargador que es cuando se tienen varios subprogramas que se deben de fusionar en un programa ejecutable único, en el cual cada una de las variables definidas en los subprogramas deben de tener asignadas direcciones reales de almacenamiento. Esto ocurre durante el tiempo de carga o tiempo de vinculación.
-Tiempo de implantación del lenguaje. Son representaciones de números y operaciones aritméticas.
-Tiempo de definición del lenguaje. Son las formas opcionales de enunciados, tipos de estructuras de datos, estructuras de programa, etc.

IMPORTANCIA DE LOS TIEMPOS DE ENLACE. Las distinciones entre los Lenguajes de Programación se basan en diferencias en cuanto a Tiempos de Enlace. Ejemplo: no todos los lenguajes son igualmente idóneos para problemas de programación que impliquen grandes cantidades de aritmética. Un diseño de lenguaje especifica el momento más cercano durante el procesamiento del programa, en el cual es posible un enlace particular. Los Tiempos de Enlace dependen de la implementación del lenguaje.

2.4- PARADIGMAS DE LOS LENGUAJES DE PROGRAMACIÓN.

Paradigmas son formas o maneras de ver las cosas. Los describen 4 modelos:
LENGUAJES IMPERATIVOS O DE PROCEDIMIENTOS. Se componen de una serie de enunciados imperativos combinados entre sí 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 y alcance un resultado deseado. Su sintaxis es: enunciado 1

enunciado 2…

Estos se adhieren a la arquitectura convencional de una computadora, que realiza operaciones de manera secuencial. Ejemplos: Algol 60, Pascal, C, y Fortran. También tienen la imposibilidad de demostrar que los programas estén correctos, porque lo correcto de un programa depende del contenido de todas y cada una de las celdas de memoria. Para poder observar un programa a través del tiempo debemos tomar "fotos instantáneas" de la memoria antes y después de ejecutar cada paso. Si el programa maneja una cantidad grande de memoria esto se vuelve tedioso e imposible en general.

LENGUAJES APLICATIVOS O FUNCIONALES. Examinan la función que el programa representa, y no sólo los cambios de estado conforme el programa se ejecuta, enunciado por enunciado. Esto se consigue viendo el resultado deseado en vez de los datos disponibles. Sus características generales son:

-Un conjunto de funciones primitivas que son predefinidas en el lenguaje y pueden ser aplicadas.

-Un conjunto de formas funcionales que son los mecanismos mediante los cuales podemos combinar funciones para crear funciones nuevas.

-Las operaciones de aplicación es el mecanismo construido en el lenguaje para aplicar una función a sus argumentos y obtener un valor.

-Un conjunto de objetos de datos son los objetos permitidos del dominio y el rango.

Los lenguajes funcionales están muy restringidos en cuanto a la variedad de objetos de datos que permiten, siendo éstos conjuntos de una estructura simple y regular. Su sintaxis es similar a: $función_n(...función_2(función_1(datos))...)$ Ejemplos: LISP, ML, etc. LISP es el lenguaje funcional más antiguo y popular, se aplica en la Inteligencia Artificial. Programar en éstos lenguajes, significa construir funciones a partir de las ya existentes. La desventaja del modelo es que está bastante alejado

del Modelo de la Máquina de Von Neumann, por lo tanto, la eficiencia de ejecución de los intérpretes de lenguajes funcionales no es comparable con la ejecución de los programas imperativos precompilados. Para solucionar la deficiencia, se busca utilizar arquitecturas paralelas que mejoren el desempeño de los programas funcionales.

LENGUAJES CON BASE EN REGLAS O LÓGICO. Ejecutan una acción cuando se verifica y satisface una condición habilitadora. Las condiciones habilitadoras determinan el orden de ejecución, su sintaxis: condición$_1$ entonces acción$_{1...}$ Ejemplos: PROLOG que es el primer lenguaje lógico más conocido, aplicado a la Inteligencia Artificial. En la programación lógica, el trabajo del programador se restringe a la buena descripción del problema en forma de hechos y reglas. Así, se pueden encontrar muchas soluciones dependiendo de cómo se formulen las preguntas (metas), que tienen sentido para el problema. Si el programa está bien definido, el sistema encuentra automáticamente las respuestas a las preguntas formuladas. Entonces, ya no es necesario definir el algoritmo de solución (como en la programación imperativa) sino expresar bien el conocimiento sobre el problema mismo. La eficiencia de la ejecución no puede ser comparable con la de un programa equivalente escrito en un lenguaje imperativo.

LENGUAJES ORIENTADOS A OBJETOS. Utilizan y manejan entes (objetos) a través de sus propiedades y características de objetos (entes reales). Los noruegos Dahl y Nygaard crearon las Clases de Objetos, que son el concepto de objeto y sus colecciones, éstas permitieron introducir abstracciones de datos a los Lenguajes de Programación. También, las Jerarquías de Herencia de Clases sirven para reutilizar y modificar el código. Además, les debemos el concepto de polimorfismo introducido vía procedimientos virtuales. Todos estos conceptos fueron presentados en el lenguaje Simula 67, desde el año 1967. En los años 80, los Lenguajes de Programación con conceptos de objetos fueron Smalltalk, C++, Eiffel, Modula-3, Ada 95 y terminando con Java. La moda de objetos se ha extendido de los Lenguajes de Programación a la Ingeniería de Software.

El modelo de objetos y los lenguajes que lo usan, facilitan la construcción de sistemas en forma modular. Los objetos ayudan a expresar programas en términos de abstracciones del mundo real, lo que aumenta su comprensión. La clase ofrece cierto tipo de Modularización que facilita las modificaciones al sistema. La reutilización de clases es otro punto a favor. Sin embargo, el modelo de objetos, al interpretarse en la arquitectura Von Neumann tiene un excesivo manejo dinámico de memoria debido a la constante creación de objetos, así como a una carga de código fuente causada por la constante invocación de métodos. Por lo tanto, los programas en Lenguajes Orientados a Objetos siempre pierden en eficiencia, en tiempo y memoria, contra los programas equivalentes en Lenguajes Imperativos. Pero ganan en la Comprensión de Código.

LENGUAJES CONCURRENTES, PARALELOS Y DISTRIBUIDOS. Busca aprovechar al máximo la arquitectura Von Neumann y sus modalidades reflejadas en conexiones paralelas y distribuidas. Encontramos soluciones conceptuales y mecanismos como semáforos, regiones críticas, monitores, envío de mensajes (CSP), llamadas a procedimientos remotos (RPC), que se incluyeron como partes de los Lenguajes de Programación en Concurrent Pascal, Modula, Ada, OCCAM, y últimamente en Java. Es difícil evaluar las propuestas existentes de lenguajes para la Programación Concurrente, Paralela y Distribuida. Primero, porque los programadores están acostumbrados a la Programación Secuencial y cualquier uso de estos mecanismos les dificulta la construcción y el análisis de programas. Por otro lado, este tipo de conceptos en el pasado fue manejado principalmente a nivel de Sistemas Operativos o Protocolos de Comunicación, donde la eficiencia era crucial, y por lo tanto no se utilizaban Lenguajes de Alto Nivel para la Programación. Ahora, la programación de sistemas complejos tiene que incluir las partes de Comunicaciones, Programación Distribuida y Concurrencia. Esto lo saben los creadores de los lenguajes más recientes, que integran conceptos para manejar: los Hilos de control, Comunicación, Sincronización y No Determinismo; el hardware y las aplicaciones se los exigen.

CAPÍTULO 3:

“ASPECTOS DE TRADUCCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN”.

3.1- SINTAXIS DE LOS LENGUAJES DE PROGRAMACIÓN.

El estudio de los lenguajes de programación se divide en el análisis de la sintaxis y la semántica.

SINTAXIS. Forma que tienen los elementos principales del programa (identificadores, palabras claves, etc), expresiones, sentencias y unidades del programa.

SEMÁNTICA. Es el significado que tienen los elementos principales del programa, expresiones, sentencias y unidades del programa.
La descripción y entendimiento completo de un lenguaje requiere de ambos aspectos.

OBJETIVO DE LA SINTAXIS. Es proveer una notación para comunicación entre el programador y el procesador del lenguaje de programación.
-Legibilidad: Un programa es legible en la medida que es autodocumentado.
-Facilidad para escribir programas: busca estructuras sintácticas concisas, regulares y poderosas, mientras que la legibilidad busca construcciones más verborragias.
-Facilidad de Traducción: si más regular es la estructura de un programa, más simple es de traducir.
-Facilidad de Verificación: Prueba de programas o corrección de programas.
-Falta de ambigüedad: La ambigüedad es un problema importante en todo diseño de lenguajes. Una construcción ambigua permite dos o más interpretaciones diferentes.

ELEMENTOS SINTÁCTICOS DE UN LENGUAJE. Son factores sintácticos básicos que afectan el estilo sintáctico general.
-Conjunto de caracteres: como representar los caracteres que su secuencia es un programa.

-Identificadores: la sintaxis básica para identificadores, una cadena de letras y dígitos comenzando con letra, es ampliamente aceptada.
-Símbolos de operadores. Las operaciones primitivas pueden ser representadas por: Símbolos especiales e identificadores.
-Palabras claves y palabras reservadas: una palabra clave es un identificador que se usa como una parte fija de la sintaxis de una sentencia. Una palabra clave es reservada sino puede también ser usada como un identificador elegido por el programador.
-Palabras opcionales: son palabras que son insertadas en sentencias para mejorar la legibilidad.
Comentarios: la inclusión de comentarios en un programa es una parte importante de su documentación.
-Blancos: las reglas sobre el uso de los blancos varían entre los lenguajes.
-Delimitadores y corchetes: los delimitadores marcan el comienzo o fin de alguna unidad sintáctica o expresión. Los corchetes son delimitadores apareados.
Formato libre: las sentencias van en cualquier posición en las líneas de entrada sin importar cortes de línea.
Formato fijo: la posición dentro de la línea de entrada es importante.
Expresiones: construcciones sintácticas que acceden objetos de datos y retornan un valor (son funciones).
Sentencias: componente sintáctica principal en los lenguajes imperativos, la clase dominante de lenguajes en uso en la actualidad.

3.2- ETAPAS DE TRADUCCIÓN.

El proceso de traducción de un programa escrito en su sintaxis original, en la forma ejecutable es fundamental en toda implementación del lenguaje de programación. Se puede dividir la traducción en dos partes principales: el análisis del programa fuente dado como entrada y la síntesis del programa objeto ejecutable.

MÓDULOS DE UN COMPILADOR. Los traductores son agrupados de acuerdo al número de pasadas que hacen sobre el programa fuente. El compilador más común hace dos pasadas sobre el programa fuente.

La primera fase del análisis descompone el programa en sus componentes constituyentes y obtiene información, como por ejemplo el uso de un nombre de variable en el programa.

La segunda fase genera un programa objeto a partir de la información recolectada. Si la velocidad de compilación es importante, se puede usar la estrategia de una pasada. La figura siguiente va a mostrar la estructura de un compilador y ayuda a identificarlo de una forma más sencilla:

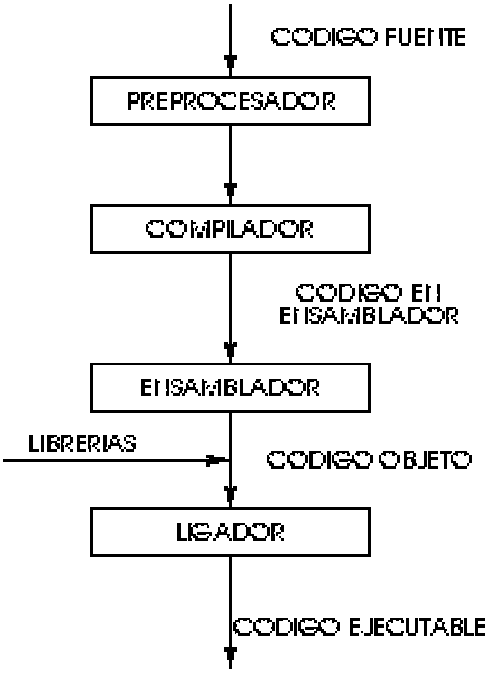


FIG 2: Estructura de un Compilador.

ANÁLISIS DEL PROGRAMA FUENTE. Para un traductor el programa fuente es una secuencia de cientos, miles o más caracteres no diferenciados. Un análisis de la estructura del programa se debe construir caracter a caracter durante la traducción.

-Análisis lexicográfico: fase básica del proceso de traducción, agrupa una secuencia de caracteres en sus componentes elementales: identificadores, delimitadores, símbolos de operadores, números, palabras claves, blancos, comentarios. Los elementos básicos del programa que resultan del análisis se llaman tokens (o items lexicográficos). El modelo básico que se usa para diseñar analizadores lexicográficos es el autómata finito.

-Análisis Sintáctico: se identifican las estructuras del programa (sentencias, expresiones, etc) usando los ítems producidos por el analizador lexicográfico. El análisis sintáctico interactúa con el análisis semántico, el analizador sintáctico identifica una secuencia de items lexicográfico formando una unidad sintáctica y luego el analizador semántico es llamado para procesar esta unidad.

-Análisis Semántico: las estructuras sintácticas reconocidas por el analizador sintáctico son procesadas, y la estructura del código objeto ejecutable comienza a tomar forma. El analizador semántico puede producir directamente el código objeto ejecutable, pero lo más común es que la salida de esta fase sea alguna forma intermedia del programa ejecutable final, el cual es manipulado por la fase de optimización del traductor, antes que el código ejecutable sea realmente generado. Las funciones del analizador semántico pueden variar, dependiendo del lenguaje y la organización lógica del traductor. Algunas de las funciones más comunes pueden ser: mantenimiento de la tabla de símbolos, inserción de información implícita, detección de errores, operaciones de tiempo de compilación y macro-procesamiento.

Las fases finales de la traducción tienen que ver con la construcción del programa ejecutable a partir de la salida producida por el analizador semántico. Esta fase involucra la generación de código y puede también incluir optimización del programa generado. Si los subprogramas son traducidos en forma separada, o si subprogramas de librerías son usados, una fase de linkeo y carga es necesaria para producir el programa completo ejecutable.

MODELOS DE TRADUCCIÓN FORMALES. Un símbolo es una entidad abstracta. Letras y dígitos son ejemplos de símbolos frecuentemente usados. Una cadena o palabra es una secuencia finita de símbolos. El conjunto de símbolos usados para formar las cadenas se denomina alfabeto, (_: símbolo usual para denotar un alfabeto). Por lo tanto, se define como lenguaje, al conjunto de cadenas formadas a partir de un determinado alfabeto.

DISPOSITIVOS DESCRIPTORES DE LENGUAJES MÁS AVANZADOS.

-Dispositivos generadores - Gramática: Es un modelo matemático que permite generar a través de reglas sintácticas o gramaticales, cadenas miembros de un leguaje específico.

-Dispositivos reconocedores - Autómata o Máquina Abstracta: Es un modelo matemático que representa la idea de computación o manipulación de cadenas, vía la aplicación de acciones preestablecidas. Tiene como objetivo, determinar la pertenencia de una cadena a un lenguaje específico.

GRAMÁTICAS. Son el método más popular utilizado para describir la sintaxis de los lenguajes de programación. Existen distintos tipos de gramáticas y asociado a cada uno de ellos hay un autómata. Chomsky define cuatro tipos: Gramáticas regulares, Gramáticas libres del contexto, Gramáticas dependientes del contexto y las Gramáticas irrestrictas.

Las más usadas en el área de compiladores son las *gramáticas libres del contexto y las* gramáticas regulares. Existe también otra forma de describir un lenguaje de programación que se denomina BNF (Backus Naur Form). BNF es un metalenguaje, o sea un lenguaje que se usa para describir otro lenguaje.

CAPÍTULO 4:

“TIPOS Y OBJETOS”.

4.1- PROPIEDADES.

Un programa es un conjunto de operaciones que son aplicadas a ciertos datos en una secuencia específica. Hay diferencias básicas de un lenguaje a otro, como los tipos de datos permitidos, tipos de operaciones válidas y los mecanismos para establecer un control en la secuencia de ejecución del programa. Las áreas de almacenamiento de datos de una Computadora Virtual para un lenguaje de programación tienen una organización muy compleja, con arreglos, stacks, números, caracteres, cadenas, y otras estructuras de datos que existen durante la ejecución del programa. Por eso, es necesario emplear el término Objeto de Datos que es la ejecución en tiempo de un grupo o más estructuras de datos en una Computadora Virtual. En el Almacenado Estático de datos en áreas específicas en la computadora, estos datos cambian, al igual que su interrelación durante la Ejecución Dinámica o Proceso del Programa. Algunos Objetos de Datos que existen durante la ejecución del programa, son definidos por el programador y pueden ser variables, constantes, arreglos, archivos, etc. Todos los Objetos de Datos podrán ser manipulados a través de declaraciones dentro del mismo programa, otros pueden ser creados, manejados y modificados dinámicamente por el sistema, y eventualmente el programador o usuario tendrán acceso a ellos.

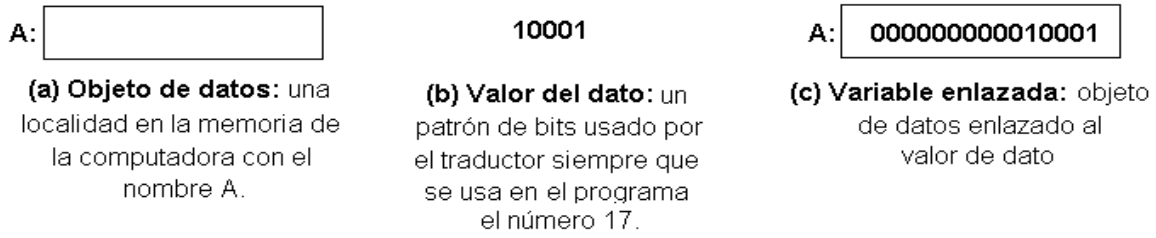


FIG3: Objetos de datos.

OBJETO DE DATOS. Es un contenedor de valores de datos, donde éstos pueden ser almacenados para después ser llamados. Uno objeto de datos se caracteriza por un Set (o conjunto) de Atributos, el más importante de ellos es el Tipo de Dato. El atributo determina el número y tipo de valores que el objeto de datos puede contener, también determina la organización lógica de estos valores.

Una “variable” simple es un tipo de objeto elemental de datos etiquetado con un nombre. Una “constante” es un objeto de datos etiquetado con un nombre y está limitado a un valor permanente durante toda su existencia.

TIPO DE DATOS. Es una clase de objeto de datos, junto con un set de operaciones para su creación y manipulación. Puede ser estudiado en términos de su especificación (u organización lógica) y en términos de su implementación.
-ELEMENTOS BÁSICOS PARA ESPECIFICAR UN TIPO DE DATOS SON: Los atributos que distingue al tipo del objeto de datos, el valor que puede tener el objeto de datos, y las operaciones que define la manipulación del objeto de datos.
-ELEMENTOS BÁSICOS PARA IMPLEMENTAR UN TIPO DE DATOS SON: La representación de almacenado que es usado para representar el objeto de dato del tipo de dato en el área de almacenado en la computadora durante la ejecución del programa y; la manera en la cual están definidas las operaciones para el tipo de dato están representadas en términos de un algoritmo o procedimiento particular que manipula la opción de la representación de almacenado para el objeto de dato. Las convenciones locales suelen afectar la manera como los datos se guardan y se procesan.

4.2- TIPOS ELEMENTALES.

Los Tipos Elementales son los Objetos Elementales de datos, definidos por los Lenguajes de Programación, conocidos como Tipos Nativos. Son los tipos de datos numéricos, enumeraciones, booleanos, caracteres y los problemas de Internacionalización que se presentan al manejar algunos tipos de datos.

TIPOS DE DATOS NUMÉRICOS. Tipos enteros y de números reales son los más comunes porque se manejan directamente en el hardware de la computadora. Las propiedades de las representaciones de datos numéricos y la aritmética en las computadoras difieren de manera sustancial de los números y operaciones aritméticas que se estudian en las matemáticas ordinarias.

ENTEROS. Un objeto de datos de tipo entero no tiene atributos además de su tipo. El conjunto de valores enteros definidos para el tipo forma un subconjunto ordenado, dentro de ciertos límites finitos del conjunto infinito de enteros que se estudia en matemáticas. El valor entero máximo depende del número de bits que se destinen a contener un número. Las operaciones sobre éstos, son:

-Operaciones aritméticas: BinOp: entero x entero = entero

Operaciones aritméticas unarias tienen la especificación: UnaryOp: entero= entero

-Operaciones relacionales: RelOp : entero x entero = booleano

-Asignación: entero x entero = vacío ó entero x entero = entero

El tipo entero de datos definido por el lenguaje se implementa con mayor frecuencia usando una representación de almacenamiento de enteros definida por el hardware y un conjunto de aritmética de hardware y operaciones primitivas relacionales sobre enteros. Ésta representación utiliza una palabra de memoria (o serie de bytes) completa para guardar un entero.

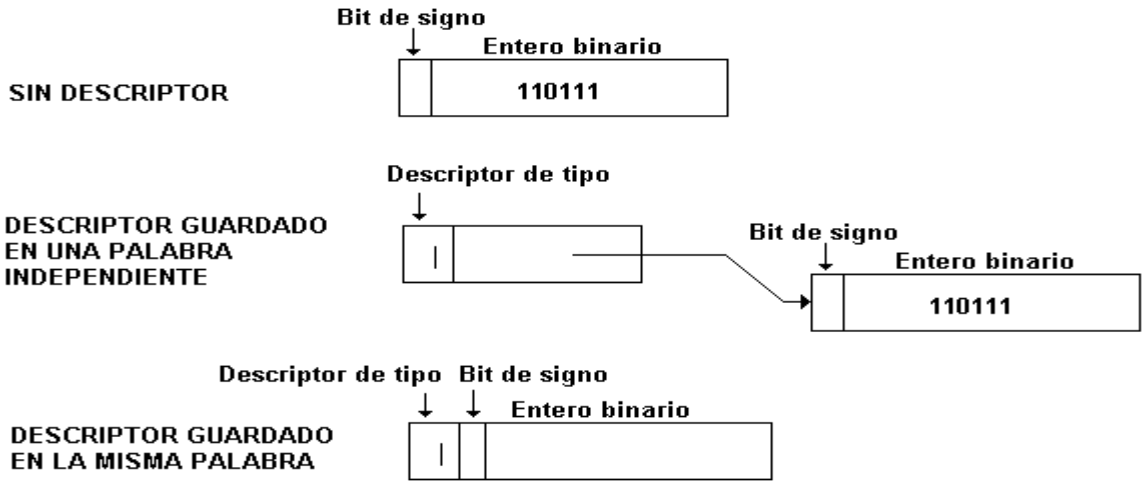


FIG 4: Almacenar entero.

SUBINTERVALO. Es un subtipo del Tipo Entero de Datos y consiste en una serie de valores enteros dentro de cierto intervalo restringido. Ejemplo: enteros en intervalo de 1 a 10. Su declaración es: A: 1... 10 (Pascal). Un tipo de subintervalo permite usar el mismo conjunto de operaciones que para el tipo entero ordinario;

por tanto, un subintervalo se puede designar como un subtipo del tipo base entero. Los tipos de subintervalo tienen 2 efectos sobre las implementaciones:

-Requerimientos de almacenamiento más reducidos: Porque es posible un intervalo más pequeño de valores, un valor de subintervalo se puede guardar ordinariamente en menos bits que un valor entero general.

-Mejor verificación de tipos: La declaración de una variable como perteneciente a un tipo de subintervalo permite llevar a cabo una verificación de tipos más precisa sobre los valores asignados a esa variable.

NÚMEROS REALES DE PUNTO FLOTANTE. Se especifican con sólo el atributo individual de tipo de datos real, como en FORTRAN, o float, como en C. Al igual que en el caso del tipo entero, los valores forman una serie ordenada desde cierto valor mínimo negativo hasta un valor máximo determinados por el hardware, pero los valores no están distribuidos de manera uniforme a través de este intervalo. De manera alternativa, la precisión que se requiere para números de punto flotante, en términos del número de dígitos que se usan en la representación decimal, puede ser especificada por el programador, como en Ada. La misma aritmética (relacional) y las operaciones de asignación descritas para enteros también se suministran comúnmente para números reales. Los programas que verifican en busca de igualdad para salir de una iteración pueden nunca concluir. Por esta razón, la igualdad entre dos números reales puede ser prohibida por el diseñador del lenguaje para impedir esta forma de error. Además, casi todos los lenguajes suministran otras operaciones como funciones integradas, tales como:

sen : real - real (función seno)

máx : real x real - real (función de valor máximo)

Las representaciones de almacenamiento para tipos reales de punto flotante se basan en una representación de hardware subyacente, en la cual una localidad de almacenamiento se divide en una mantisa (los dígitos significativos del número) y un exponente. Este modelo emula la notación científica, donde cualquier número N se pueden expresar como $N= m \times 2k$ para m entre 0 y 1 y para cierto entero k. El estándar o norma 754 del IEEE se ha convertido en la definición

aceptada para formato de punto flotante en muchas implementaciones. También suele estar disponible una forma de doble precisión de número de punto flotante, en la cual se usa una palabra adicional de memoria para guardar una mantisa extendida. La exponenciación se simula por software. El programador puede declarar simplemente que una variable real es “doble o real larga” para especificar el uso de doble precisión como representación de almacenamiento. Ejemplo de su forma de almacenar:

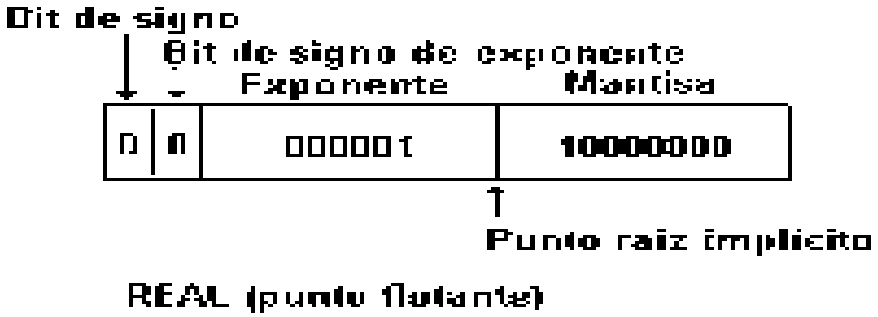


FIG 5: Almacenar número de punto flotante.

NÚMEROS REALES DE PUNTO FIJO. Se aplican en los objetos de datos que representan dinero, contienen pesos y centavos, que son números racionales escritos con dos cifras decimales. Éstos no se pueden escribir como enteros y como valores de punto flotante pueden tener errores de redondeo. Se puede usar una forma de datos de punto fijo para representar esta clase de valores. Un número de punto fijo se representa como una serie de dígitos de longitud fija, con el punto decimal situado en un punto dado entre dos dígitos.

OTROS TIPOS DE DATOS NUMÉRICOS. Son los “Números complejos” que pueden suministrarse con facilidad representando cada objeto de datos como un bloque de dos ubicaciones de almacenamiento que contienen un par de valores reales. Las operaciones sobre éstos, se pueden simular por software.
Los “números racionales” buscan evitar los problemas de redondeo y truncado que se encuentran en las representaciones de punto flotante y fijo de números reales. Como consecuencia, va a ser recomendable representar los números racionales

como pares de enteros de longitud limitada. Estos enteros largos se suelen representar usando una representación vinculada.

ENUMERACIONES. Una variable adopta sólo uno de un número reducido de valores simbólicos. El programador define los nombres de literales que se van a usar para los valores y su ordenamiento usando una declaración como la siguiente en C: enum ClaseEstudiante {Primero, Segundo, Tercero, Cuarto};

enum SexoEmpleado {Masculino, Femenino};

Las operaciones básicas sobre tipos de enumeración son las operaciones relacionales (igual, menor que, mayor que, etc.), las de asignación y las operaciones sucesor y predecesor, las cuales proporcionan el valor siguiente y el anterior, respectivamente, en el orden de las literales que definen la enumeración (y que no están definidas para los valores último y primero, respectivamente).

La representación de almacenamiento para un objeto de datos en un tipo de enumeración es sencilla. Cada valor de la serie de enumeración está representado en tiempo de ejecución por uno de los enteros 0, 1,2,... Ya que sólo interviene un conjunto pequeño de valores y los mismos nunca son negativos, la representación usual de enteros se suele acortar para omitir el bit de signo y usar sólo los bits suficientes para el intervalo de valores que se requiere, como en el caso de un valor de subintervalo.

BOOLEANOS. Sirven para representar cierto y falso. Su representación de almacenamiento es un solo bit, siempre y cuando no se necesite un descriptor que designe el tipo de datos. Como los bits individuales pueden no ser direccionables por separado en la memoria, ésta representación de almacenamiento se suele ampliar para que sea una sola unidad direccionable, como un byte o una palabra. type booleano = (falso, cierto); A veces se incluyen otras operaciones booleanas como equivalencia, o exclusivo, implicación, ny (no-y) y no (no-o).

CARACTERES. Las series de caracteres (cadenas de caracteres) se suelen procesar como una unidad. El conjunto de posibles valores de carácter se toma

como una enumeración definida por el lenguaje que corresponde a los conjuntos normales de caracteres ASCII. El ordenamiento de los caracteres en este conjunto se conoce como la secuencia ordenada del conjunto de caracteres. La secuencia ordenada es importante porque determina el ordenamiento alfabético que las operaciones relacionadas asignan a las cadenas de caracteres. Las operaciones sobre datos de caracteres incluyen sólo relaciones, la asignación y a veces operaciones para probar si un valor de caracteres pertenece a las clases especiales "letra", "dígito" o "caracteres especiales". Los valores de datos de carácter casi siempre son manejados directamente por el hardware y el sistema operativo subyacentes a causa de su uso en la entrada - salida.

INTERNACIONALIZACIÓN. Las convenciones locales suelen afectar la manera como los datos se guardan y se procesan.

-Ordenamiento. Posición de los caracteres no romanos como Å, Ø, ß, ö y otros, no está definida de manera uniforme y puede tener distintas interpretaciones en diferentes países.

-Mayúsculas/minúsculas: El japonés, árabe, hebreo y tailandés carecen de distinción entre mayúsculas y minúsculas.

-Dirección de lectura: Casi todos los idiomas se leen de izquierda a derecha.

-Formatos de fecha específicos del país: 11/26/95 en Estados Unidos, es 26/11/95 en Inglaterra, 26.11.95 en Francia, 26-XI-95 en Italia, etc.

-Formatos de hora específicos del país: Las 5:40 p.m. en Estados Unidos son las 17:40 en Japón, las 17:40 en Alemania, las 17h40 en Francia, etc.

-Usos horarios: separados por un núm. entero de hrs, unos varían 15-30 minutos.

-Sistemas ideográficos: Ciertos idiomas escritos no se basan en un número pequeño de caracteres que forma un alfabeto, sino que usan un gran número de ideogramas. Suelen ser necesarios 16 bits para representar texto en esos idiomas.

-Moneda: ejemplo, $, L, Y, varía según el país.

4.3- TIPOS ESTRUCTURADOS.

Los “Componentes” son un objeto de datos (estructura de datos) creado de un agregado de otros objetos de datos. Los Componentes de una estructura de datos pueden ser elementales u otras estructuras de datos. Los tipos de estructuras de datos también implican puntos de especificaciones de tipo, implantación de tipos, declaración y verificación de tipos, a un nivel más complejo. Consideraciones importantes de los Tipos Estructurados son:

-Especificación e Implantación de Información Estructural: cómo indicar los objetos de datos que componen una estructura de datos y sus relaciones en una forma tal que la selección de un componente de la estructura sea sencilla.

-Gestión de Almacenamiento: debido a que muchas operaciones sobre estructuras de datos sacan a colación cuestiones de almacenamiento que no están presentes para objetos elementales de datos.

ATRIBUTOS PRINCIPALES PARA ESPECIFICAR ESTRUCTURAS DE DATOS.

-NÚMERO DE COMPONENTES. Es de tamaño fijo si el número de componentes no es invariable durante el tiempo de vida (arreglos y registros). Es de tamaño variable si el número de componentes cambia en forma dinámica (listas, conjuntos, tablas y archivos) y éstos emplean un tipo de dato apuntador, además de definir operaciones que inserten y eliminen componentes de la estructura.

-TIPO DE CADA COMPONENTE. Una estructura de datos es homogénea si todos sus componentes son del mismo tipo, de otro modo es heterogénea.

-NOMBRES USADOS PARA SELECCIONAR COMPONENTES: mecanismo de selección para identificar componentes individuales. En arreglos puede ser un subíndice y en registros el nombre propio de cada campo.

-NÚMERO MÁXIMO DE COMPONENTES: tamaño de la estructura basado en el número de componentes.

-ORGANIZACIÓN DE LOS COMPONENTES: es una serie lineal sencilla de componentes. Hay estructuras que abarcan formas multidimensionales que

pueden tratarse como el tipo secuencial básico en el cual los componentes son estructuras de datos de tipo similar.

OPERACIONES SOBRE TIPOS ESTRUCTURADOS. Tienen la misma manera de especificación del dominio y ámbito de operaciones que los tipos elementales. Las nuevas operaciones de interés son:

-Operaciones de Selección de Componentes: Permiten acceder a componentes de la estructura y ponerlos a disposición para ser procesados por otras operaciones. Existe la Selección Directa que permite acceder a un componente arbitrario y la Selección Secuencial, donde los componentes acceden en un orden determinado. Para seleccionar un componente se realizan dos operaciones, uno el refinamiento que determina la localidad del nombre del objeto y regresa un apuntador a esa localidad; y la selección, lleva el apuntador al vector junto con el subíndice del componente y devuelve un apuntador a la localidad de ese componente particular.

-Operaciones con Estructuras de Datos Completas: pueden tomar estructuras de datos enteras como operandos y producir nuevas estructuras de datos como resultados.

-Inserción y eliminación de componentes: para estructuras de tamaño variable y tienen un impacto importante sobre la representación y gestión de almacenamiento, porque cambian el tamaño de la estructura.

-Creación y Destrucción de Estructuras de Datos: impacto importante sobre la gestión de almacenamiento.

IMPLANTACIÓN DE TIPOS DE ESTRUCTURAS DE DATOS. Son los mismos puntos que en el caso de los tipos elementales, pero surgen cuestiones:

-Representación de Almacenamiento: consta del almacenamiento para los componentes de la estructura y un descriptor optativo que guarda algunos de los atributos de ésta. Existe la Representación Secuencial, donde la estructura se guarda en un solo bloque contiguo que incluye al descriptor y sus componentes, la Representación Vinculada, donde la estructura se guarda en varios bloques no contiguos enlazados entre sí a través de un apuntador, llamado vínculo.

La Operación de Selección de Componentes es muy importante para implantar operaciones sobre estructuras de datos, y se hace de manera diferente para cada representación de almacenamiento: en la Representación Secuencial para la selección directa se calcula la dirección base más el desplazamiento mediante una fórmula de acceso; y en la Representación Vinculada, la selección vinculada implica seguir una cadena de apuntadores desde el primer bloque de almacenamiento.

En las Gestiones de Almacenamiento se presentan dos problemas a causa de la acción reciproca entre el tiempo de vida de un objeto y las rutas de acceso al mismo: la Basura, surge cuando se destruyen todas las rutas de acceso a un objeto de datos pero éste continúa existiendo, es decir, no se ha roto el enlace del objeto con la localidad de memoria; y las Referencias Desactivadas, se tienen cuando una ruta de acceso continúa existiendo después de tiempo de vida del objeto de datos asociado.

DECLARACIÓN Y VERIFICACIÓN DE TIPOS DE ESTRUCTURAS DE DATOS. Sus conceptos y preocupaciones son similares que los de los tipos de objetos elementales, aunque las estructuras son más complejas porque hay más atributos por especificar. La verificación de tipos es más difícil porque se deben tomar en cuenta las operaciones de selección de componentes y surgen estos problemas:

-Existencia de un Componente Seleccionado: se puede intentar seleccionar un componente que no exista en la estructura de datos. No será un problema de verificación de tipos, siempre y cuando la operación de selección fracase y el error se maneje mediante una excepción. Sin embargo, si se inhabilita la verificación de tipos en tiempo de ejecución y no se plantea la excepción, el efecto del error de selección es similar al de un error de verificación de tipos (valor inválido). Las operaciones de selección de componentes que usan una fórmula de acceso para el cómputo del desplazamiento del componente, deben verificar en tiempo de ejecución la existencia del componente antes de usar la fórmula para evitar este tipo de error.

-Tipo de un Componente Seleccionado: para realizar la verificación estática de tipos, se debe poder determinar en tiempo de compilación el tipo del componente seleccionado por cualquier selector compuesto válido, aún cuando la secuencia de selección haya definido una ruta compleja a través de la estructura de datos hacia el componente deseado. La selección estática de tipos garantiza la existencia del componente, y si es del tipo correcto.

VECTORES Y ARREGLOS. Son estructuras de datos muy comunes. El Vector (Arreglo Unidimensional o Lineal), es una estructura de datos integrada por un número fijo de componentes del mismo tipo, organizados como una serie lineal simple. Su componente se selecciona dado su subíndice, que indica la posición del componente en la serie. El Arreglo Multidimensional se compone de arreglos de dimensiones menores. Ejemplo: un arreglo bidimensional es una matriz compuesta de filas y columnas, construida como un arreglo de arreglos; un arreglo tridimensional compuesto de planos de filas y columnas, construido como un arreglo de matrices. Los atributos de un vector son:

-Número de Componentes: da serie de intervalos de subíndices a cada dimensión.

-Tipo de Datos de cada Componente: es un solo tipo de datos, si todos los componentes son del mismo tipo.

-Subíndice de Selección de cada Componente: intervalo de enteros, donde el primer entero designa el primer componente, el segundo designa al segundo componente, etc. Puede ser un intervalo de valores o un límite superior con un límite inferior implícito.

“Subindización” es la operación de Selección de un Componente del Vector y es el nombre del vector seguido del subíndice del componente por seleccionar. Otras operaciones son Crear o Destruir Vectores, Asignación de Componentes a un Vector, y Operaciones Aritméticas sobre Parejas de Vectores del mismo tamaño, como la Suma y Resta de dos vectores. Puesto que los vectores son estructuras de tamaño fijo, no se permite la Inserción ni la Eliminación de Componentes; sólo se puede Modificar el valor de un componente.

La homogeneidad de los componentes y el tamaño fijo de un vector simplifica el almacenamiento de componentes individuales y el acceso a los mismos. Estas características implican que el tamaño y estructura de cada componente son los mismos, y que el número y posición de cada componente son invariables a lo largo de su tiempo de vida. Existen tres representaciones para la Implantación de Vectores: Representación de Almacenamiento Secuencial, Representación de Almacenamiento Empacado y Representación de Almacenamiento No Empacado. Además de considerar el tipo de representación, se deben considerar las operaciones con vectores completos.

REPRESENTACIÓN DE ALMACENAMIENTO SECUENCIAL DE VECTORES. Los componentes se guardan en forma contigua. Puede incluir un descriptor para guardar algunos atributos, en especial si toda esta información no se conoce sino hasta el tiempo de ejecución. Los límites superior e inferior se guardan si se requiere verificación de intervalo para valores de subíndices calculados. Los otros atributos no se guardan en el descriptor en tiempo de ejecución, sólo se requieren durante la traducción para la verificación de tipos y establecer la representación de almacenamiento.

La fórmula para calcular la dirección de almacenamiento de un componente y accederlo surge porque a partir del componente inicial, la dirección del i_ésimo componente se puede obtener saltando I-1 componentes; si E = Tamaño de cada Componente, entonces es necesario saltar (i-1)*E localidades de memoria. Si Li = Límite Inferior del Intervalo de Subíndices, entonces hay que saltar (I - Li)*E posiciones de memoria. Si el primer elemento del vector comienza en la localidad “ ”, se tiene la fórmula de acceso para el i_ésimo componente, dada por:

Valor_i (A [i]) = + (i – Li) * E

En algunos lenguajes (FORTRAN), K es una constante y el acceso es más rápido, pero en otros (Pascal), cada argumento de K puede ser variable y es necesario hacer el cálculo una vez cuando se asigna memoria para el vector. Debido a que el tamaño de cada componente es conocido en tiempo de traducción, si el

traductor también conoce el valor del subíndice, entonces la fórmula de acceso se reduce a un valor obtenido en tiempo de compilación.
Si utilizamos la Fórmula de Acceso para obtener la dirección del elemento con subíndice 0, obtenemos la constante K, que representaría la dirección que ocuparía el elemento 0 del vector, si existiera. Dado que el elemento que ocupa la posición cero no puede formar parte del arreglo, puesto que éste puede tener un límite inferior mayor que 0, a esta dirección se le conoce como el Origen Virtual (OV). Se puede observar que la Fórmula de Acceso se puede reescribir usando el Origen Virtual. Si el Origen Virtual se guarda en el descriptor, entonces el arreglo mismo no necesita ser contiguo al descriptor, y de esta manera se puede omitir el tipo de descriptor y el tipo de componentes en el descriptor.

REPRESENTACIÓN DE ALMACENAMIENTO EMPACADO Y NO EMPACADO. Dependen de la forma en que se almacenen los valores de los datos de un arreglo en las unidades de almacenamiento (palabras o bytes). La Representación de Almacenamiento Empacado es cuando los componentes de un vector (u otra estructura) están empacados en la memoria de manera secuencial, sin atender a colocar cada componente al principio de una unidad de almacenamiento. Permite ahorrar la cantidad de memoria requerida, pero el acceso a un componente es más costoso, porque no se puede usar la Fórmula de Acceso. En su lugar se requiere una serie más compleja de cálculos para acceder a la palabra de memoria que contiene el componente, luego quitar los otros componentes y recorrer el componente a uno de los extremos de la palabra, etc. Si un componente puede cruzar un límite de palabra, el acceso es aún más difícil.
La Representación de Almacenamiento No Empacado es cuando se guarda cada componente comenzando en el límite de una unidad de almacenamiento, y entre cada par de componentes puede quedar memoria no usada que representa Relleno. Usa la Fórmula de Acceso para acceder a los componentes, pero el costo de almacenamiento se incrementa; es la forma más común de representación de almacenamiento.

OPERACIONES DE VECTORES COMPLETOS. La asignación de un vector a otro con los mismos atributos se implanta copiando el contenido del bloque de almacenamiento y usan la representación secuencial de almacenamiento. No es necesario copiar el descriptor. Las operaciones aritméticas o especializadas sobre vectores, se implementan a manera de iteraciones procesando los elementos de los vectores en serie. Un problema es el almacenamiento requerido para el resultado, porque puede requerirse de almacenamiento temporal incrementándose la complejidad y el costo de la ejecución.

IMPLANTACIÓN DE ARREGLOS MULTIDIMENSIONALES. La Matriz, considera un vector de vectores; un arreglo tridimensional se implanta como un vector cuyos componentes son vectores de vectores; siguiendo este análisis en forma sucesiva, se puede concluir que un arreglo n_dimensional se implanta en base a arreglos de dimensiones menores. Un arreglo n_dimensional está organizado en orden por filas cuando el arreglo se divide en un vector de sub_vectores, y cada sub_vector también es un arreglo de sub_vectores. El orden por columnas es la representación en la cual la matriz se trata como una sola fila de columnas, y de forma semejante, al orden por filas, se puede generalizar el concepto de orden por columna para arreglos n_dimensionales.

REPRESENTACIÓN DE ALMACENAMIENTO DE UN ARREGLO BIDIMENSIONAL. Considerando el orden por filas para un vector n_dimensional, se guardan los objetos de datos de la primera fila, seguidos de los de la segunda fila, y así sucesivamente. El resultado es un bloque secuencial de memoria que contiene todos los componentes del arreglo en serie. El descriptor es similar al de un vector, pero necesita un límite inferior y superior para cada dimensión. La operación de Subindización, usando la Fórmula de Acceso para calcular el desplazamiento de un componente con respecto a la dirección base del arreglo, es similar a la que se usa para vectores: primero se determina el número de filas que hay que saltar, se multiplica por la longitud de la fila para obtener la localidad del

principio de la fila, y luego se encuentra la localidad del componente en esa fila como para un vector.

REBANADA. Es una subestructura (arreglo) de un arreglo y se puede pasar como argumento a subprogramas. Su manejo en un Lenguaje de Programación sirve para que los programadores desarrollen algoritmos de matriz y manipulen partes de una matriz más grande. El uso de descriptores permite la implantación eficiente de una rebanada; si el descriptor incluye los multiplicadores M_i, esto permite que los elementos de una dimensión puedan no ser contiguos, pero tiene la propiedad de estar igualmente espaciados.

APL (Programmong Language). Lenguajes especializados con características particulares y eficientes para describir algoritmos, surgen por el desarrollo en los sistemas computacionales de tipo remoto. Son la alternativa en Lenguajes de Programación (60's) para el manejo algebraico y uso de funciones muy útiles que no se pueden expresar en forma concisa con los símbolos convencionales.

REGISTROS. Estructura de datos compuesta de un número fijo de componentes (Campos) de distintos tipos. Son estructuras lineales y de longitud fija, al igual que los vectores, pero difieren en que sus componentes pueden ser heterogéneos y se designan con nombres simbólicos. Los atributos de un registro son: el Número de Componentes, el Tipo de Dato de cada Componente y el Selector que se usa para nombrar cada componente. La operación básica sobre un registro es la selección de componentes llamada Subindización para Arreglos, aunque ahora el "subíndice" es siempre el nombre literal del componente y nunca es un valor computado. Las operaciones sobre registros completos son pocas, la más común es la de Asignación de Registros de Estructura Idéntica.

IMPLANTACIÓN DE REGISTROS. La representación de almacenamiento para un registro consiste de un solo bloque secuencial de memoria en el cual se guardan los componentes en serie. Los componentes individuales pueden requerir descriptores para indicar su tipo de dato y otros atributos, pero en tiempo de

ejecución no se requiere del descriptor del registro. La selección de componentes se implanta con facilidad porque los subíndices (nombres de campos) se conocen durante la traducción en vez de calcularse en tiempo de ejecución. La declaración del registro también permite determinar el tamaño de cada componente y su posición dentro del bloque de almacenamiento durante la traducción. La operación de Asignación de un Registro Completo a Otro de Estructura Idéntica se puede implantar como una simple copia del contenido del bloque de almacenamiento que representa el primer registro en el bloque de almacenamiento que representa el segundo registro. Operaciones complejas como MOVE CORRESPONDING (de COBOL) se pueden implantar como una serie de asignaciones de componentes individuales de un registro a otro. Un registro es un grupo de valores tales que, cuando son agrupados forman una descripción de una entidad tal como un objeto o persona.

REGISTROS Y ARREGLOS CON COMPONENTES ESTRUCTURADOS. Cuando se tiene un arreglo de registros, sus componentes se seleccionan usando una serie de operaciones de selección para elegir primero un componente del vector y luego un componente del registro. También se pueden tener registros cuyos componentes sean arreglos u otros registros, lo cual conduce a registros con una estructura jerárquica consistente en un nivel superior de componentes, algunos de los cuales pueden ser arreglos o registros. Los componentes de estos componentes de segundo nivel también pueden ser arreglos o registros. Las representaciones de almacenamiento desarrolladas para vectores y registros simples se extienden sin cambio a vectores y registros cuyos componentes son a su vez vectores y registros. Un vector de registros tiene una representación semejante a la de un vector de vectores, pero por cada fila sustituida por la representación de almacenamiento de un registro. Un registro de registros (o vectores) conservan la misma representación secuencial de almacenamiento, pero con cada componente representado por un sub-bloque que puede ser él mismo la representación de un registro (o vector) completo.

Los Registros Variantes son un tipo especial de los tipos de datos registros, contienen uno o más componentes que son comunes a las variantes, además de otros componentes con nombres y tipos de datos que son exclusivos de cada variante. También incluyen un campo llamado Marca o Discriminante, usado para determinar qué variante se debe considerar en un tiempo de ejecución dado. Como los Registros Variantes se determinan en tiempo de ejecución, presentan el problema de seleccionar una variante no existente en el tiempo de ejecución en que se está tratando de accederlo. Los tipos de Registros Variantes algunas veces son tratados como tipos de unión, por que cada variante se puede considerar como una clase individual de objeto de datos de registro, y entonces el tipo de registro global se presenta como la unión de esos conjuntos de objetos de datos. Si no existe un campo marca, entonces se trata de un tipo de Unión Libre, de otro modo se trata de un tipo de Unión Discriminada.

SELECCIÓN DE COMPONENTES VARIANTES. En un componente de una variante, el componente puede existir en un tiempo dado durante la ejecución y más tarde dejar de existir, y aún más tarde volver a existir. Este problema de seleccionar un componente no existente de un registro con variantes es similar al error de intervalo de subíndices, hay 2 posibles soluciones:

-Verificación dinámica: El componente marca se puede verificar en tiempo de ejecución antes de accederlo, para asegurar que el componente deseado existe. Si la marca tiene el valor apropiado, entonces se procede al acceso de los componentes de la variante, de otro modo se tiene un error en tiempo de ejecución.

-Ninguna verificación: El diseño del lenguaje permite definir registros variantes sin un componente marca que se pueda verificar en tiempo de ejecución. Recupera valores no deseados y sobre-escribe valores en componentes no deseados.

La implantación de registros variantes es más sencilla que su uso correcto. Durante la traducción, se determina la cantidad de almacenamiento que requieren los componentes de cada variante, y se asigna almacenamiento en el registro para

la variante más grande. Dentro del bloque, cada variante describe una disposición distinta para el bloque en términos del número y tipo de componentes. También se determinan las disposiciones y se usan para calcular desplazamientos para la selección de componentes. No se necesita un descriptor especial para la variante porque el componente marca se considera simplemente como otro componente del registro. Además, se calcula el desplazamiento del componente seleccionado dentro del bloque de almacenamiento y, durante la ejecución, el desplazamiento se suma a la dirección base del bloque para determinar lo localidad del componente. Si se suministra verificación dinámica, entonces en tiempo de ejecución, el cálculo de dirección base más desplazamiento para localizar el componente es el mismo, pero primero se debe verificar el valor del campo de marca para asegurar que la marca indica que la variante apropiada existe actualmente.

LISTAS. Estructura de datos compuesta de una serie ordenada de estructuras de datos. Así, se puede hacer referencia al primer elemento de la lista, al segundo y así sucesivamente hasta llegar al último elemento de la lista. El primer elemento de la lista se llama Cabeza y el último elemento Cola. Sin embargo, las listas difieren de los arreglos en los siguientes aspectos: son de longitud variable y se usan para representar estructuras de datos arbitrarias. Sus componentes pueden ser heterogéneos, es decir, el tipo de dato de cada miembro de una lista puede diferir de su “vecino”. Los lenguajes que manejan listas, declaran estos datos de manera implícita, es decir sin atributos explícitos para los miembros de la lista.
La naturaleza dinámica de casi todas las implantaciones de listas y el hecho de que los elementos rara vez son homogéneos, significa que se requiere de una organización de Gestión de Almacenamiento de Lista Vinculada. Un elemento de lista es un elemento primitivo y consiste comúnmente de un objeto de datos de tamaño fijo. Generalmente se requieren de tres campos de información para poder manejar una lista: uno de tipo y dos de apuntadores de lista. En lenguajes como LISP, ML y Prolog, las listas son tipos de datos primitivos. Mientras que en los lenguajes compilados como C, Pascal o Ada, entre otros, las listas no son objetos

primitivos. La gestión dinámica de almacenamiento que se necesita para mantener listas se contrapone con la eficiente gestión de almacenamiento normal que se suele usar en este tipo de lenguajes (los compilados). No obstante, estos lenguajes tienen acceso a listas, pero deben hacerlas tipos de datos definidos por el programador. En ciertos lenguajes se presentan variaciones sobre la estructura típica de las listas, entre las cuales tenemos:

-Pila: Es una lista donde la selección, inserción y eliminación de componentes están restringidas a un extremo. En tiempo de ejecución es un objeto de datos medular definido por el sistema.

-Cola: Es una lista en la cual la selección y eliminación de componentes están restringidas a un extremo y la inserción esta restringida al otro extremo. Se usan en la organización y sincronización de subprogramas concurrentes. Son comunes las representaciones secuenciales tanto para pilas como para colas.

-Árbol: Es una lista donde los componentes pueden ser listas y objetos de datos elementales, siempre y cuando cada lista sea un componente de cuando mucho otra lista. Se suelen usar para representar tablas de símbolos en un compilador.

-Gráfica Dirigida: Es una estructura de datos en la cual los componentes se pueden vincular entre sí usando patrones de vinculación arbitrarios (en vez de series lineales de componentes).

-Lista de Propiedades: También conocida como Lista de Valores de Atributos, Lista de Descripción ó Tabla, es un registro con un número variable de componentes. En una lista de componentes se debe guardar tanto los nombres de los componentes como sus valores. Cada nombre de campo se conoce como Nombre de Propiedad y el valor correspondiente se conoce como Valor de Propiedad.

GESTIÓN DE ALMACENAMIENTO ASOCIADA A LISTAS. La gestión de almacenamiento asociada presenta un problema para la implantación de un lenguaje, por lo que el tratamiento de estos tipos de datos en cualquier lenguaje está ligado a las estructuras de gestión de almacenamiento básicas subyacentes en la implantación del lenguaje. Hay 2 enfoques hacia estos tipos de datos:

-El lenguaje suministra un tipo de datos de estructura de lista, lista de propiedades, pila o cola; y proporciona un sistema oculto de gestión de almacenamiento que determina de forma automática la asignación y recuperación de almacenamiento para estas estructuras (ML, LISP).
-El lenguaje suministra un tipo de dato apuntador, junto con las facilidades para la asignación dinámica de almacenamiento en forma explícita por parte del programador, y éste construye sus propias estructuras vinculadas (C, Pascal y Ada).

CADENAS DE CARACTERES. Es un objeto de datos compuesto de una serie de caracteres, sirve para la representación de datos de caracteres para entrada y salida. Existen 3 tratamientos de los tipos de este tipo de datos:
-Longitud Fija Declarada. Es un objeto de datos de cadena de caracteres con una longitud fija que se declara en el programa. La asignación de un nuevo valor de cadena al objeto de datos da por resultado un ajuste de longitud de la nueva cadena pero truncado de los caracteres en exceso o la adición de caracteres en blanco para producir una cadena de longitud correcta.
-Longitud Variable hasta un Límite Declarado. Es un objeto de datos de cadena de caracteres con una longitud máxima que se declara en el programa, pero el valor real que se guarda en el objeto de datos puede ser una cadena de longitud más corta, posiblemente incluso la cadena vacía. Durante el tiempo de ejecución, la longitud del valor de cadena del objeto de datos puede variar, pero es truncada si sobrepasa la longitud máxima establecida.
-Longitud Ilimitada. Es un objeto de datos de cadena de caracteres que puede tener un valor de cadena de cualquier longitud, y la longitud puede variar en forma dinámica durante la ejecución sin límite alguno.
Los 2 primeros métodos determinan la asignación de almacenamiento para cada objeto de datos en tiempo de traducción; si las cadenas son de longitud ilimitada, entonces se requiere de asignación dinámica de almacenamiento en tiempo de ejecución. Las operaciones comunes en las cadenas de caracteres son:
-Concatenación: unión de dos cadenas para hacer una más larga.

-Operaciones Relacionales sobre Cadenas: como igual, mayor que, menor que, etc., operan en base al ordenamiento lexicográfico (alfabético), en ocasiones es necesario agregar espacios en blanco a una de las cadenas que se van a comparar para que sean del mismo tamaño y así poder comparar caracter a caracter.

-Selección de Subcadenas usando Subíndices de Posición: permite trabajar con una subcadena contigua de la cadena global o principal. Para facilitarla, algunos lenguajes suministran esta operación dando las posiciones del carácter inicial y final de la subcadena.

-Establecer Formato de Entrada-Salida: dan formato a datos de salida o descomponen datos de entrada con formato en elementos de datos más pequeños.

-Selección de Subcadenas usando Equiparamiento de Patrones: es útil cuando no se conoce la posición de la subcadena dentro de la cadena, pero sí su relación con otras subcadenas. Así, esta instrucción requiere de una estructura de datos como Patrón, el cual especifica la forma de la subcadena deseada, y una cadena de caracteres que va a examinar para encontrar una subcadena que coincida con el Patrón.

Para una cadena de longitud fija declarada, la representación es esencialmente la misma que la de un vector. En la cadena de longitud variable hasta un límite declarado, la representación de almacenamiento usa un descriptor que contiene tanto la longitud máxima declarada como la longitud presente de la cadena guardada en el objeto de datos. Para cadenas de longitud ilimitada, se puede usar una representación de almacenamiento vinculada de objetos de datos de longitud fija o un arreglo continuo de caracteres para contener la cadena. Por lo común, se dispone de apoyo de hardware para la representación simple de longitud fija, pero en general, las otras representaciones se deben simular por software. Ordinariamente, las operaciones sobre cadenas, como concatenación, selección de subcadenas y concordancia de patrones se simulan por completo por software.

APUNTADORES DE OBJETOS DE DATOS CONSTRUIDOS POR EL PROGRAMADOR. Un tipo de datos Apuntador (Tipo de Referencia o de Acceso), define una clase de objetos de datos cuyos valores son la ubicación de otros objetos de datos. Los lenguajes de programación que no manejan tipos de datos vinculados de tamaño variable, pero permiten la construcción de cualquier estructura usando apuntadores para vincular objetos de datos entre sí, deben tener las siguientes características:

-Un Apuntador de Tipo Elemental de Datos, el cual contiene la localidad de otro objeto de datos. Es un objeto de datos ordinario que puede ser una variable simple, o un componente de arreglo o de registro.

-Una Operación de Creación para objetos de datos de tamaño fijo, la cual asigna un bloque de almacenamiento para el nuevo objeto de datos y crea un apuntador al nuevo objeto de datos. Se caracteriza porque los objetos creados no necesitan nombre, ya que se acceden mediante apuntadores, y los objetos de datos pueden crearse en cualquier punto de la ejecución de un programa.

-Una Operación de Referencia para valores de apuntador, permite seguir a un apuntador hasta el objeto de datos hacia el cual apunta.

Un único objeto de datos de tipo apuntador se puede tratar de dos maneras:

-Los apuntadores únicamente pueden hacer referencia a objetos de un solo tipo; en este enfoque, se utilizan declaraciones de tipos y verificación estática de tipos.

-Los apuntadores pueden hacer referencia a objetos de cualquier tipo; en este enfoque, los objetos de datos tienen descriptores de tipo durante la ejecución y se efectúa verificación dinámica de tipos.

Un apuntador se representa como una localidad de memoria que contiene la dirección de otra localidad de memoria. La dirección es la dirección base del bloque de memoria que representa el objeto de datos al que el apuntador apunta. Hay dos representaciones de almacenamiento importantes:

-Dirección Absoluta: El valor del apuntador se representa como la dirección de memoria real del bloque de almacenamiento para el objeto de datos. Al usar este tipo de almacenamiento, se puede asignar almacenamiento a los objetos de datos creados, en cualquier parte de la memoria.

-Dirección Relativa: El valor del apuntador se representa como un desplazamiento respecto a la dirección base de algún bloque de memoria más grande dentro del cual el objeto de datos está asignado. El uso de este tipo de almacenamiento requiere la asignación inicial de un bloque de almacenamiento dentro del cual tiene lugar la subsiguiente asignación de objetos de datos.

El problema principal es la asignación de almacenamiento asociada con la operación de creación. La verificación estática de tipos es posible para referencias que emplean valores de apuntador que hacen referencia a objetos de datos de un solo tipo, de otro modo se requiere verificación dinámica. La selección a través del empleo de direcciones absolutas es eficiente porque el valor de apuntador mismo proporciona acceso directo al objeto de datos usando la operación de hardware para acceder la memoria. La desventaja es que la gestión de almacenamiento es más difícil por que ningún objeto de datos puede moverse dentro de la memoria si existe un apuntador a él guardado en otra parte. La recuperación de memoria para objetos que se han convertido en basura también es difícil, por que cada objeto de datos se recupera en forma individual.

En el uso de dirección relativa, puede haber un área para cada tipo de objeto de datos por asignar, o una sola área para todos los objetos. Si se supone un área para cada tipo de objeto de datos, puede asignarse almacenamiento en bloques de tamaño fijo dentro del área, simplificando la gestión de almacenamiento. La selección es más costosa que para una dirección absoluta, porque se debe sumar el desplazamiento a la dirección base del área para obtener una dirección absoluta antes de que se pueda tener acceso al objeto de datos. Sin embargo la ventaja, radica en que permite mover el bloque de área como un todo en cualquier momento sin invalidar ninguno de los apuntadores.

CONJUNTOS. Objeto de datos que contiene una colección No Ordenada de valores distintos. Operaciones sobre conjuntos:

-Pertenencia: Es el valor X un miembro del conjunto C?

-Inserción y eliminación de valores individuales: Insertar el valor de datos X en el conjunto C, siempre y cuando no sea ya miembro de C. Eliminar el valor de datos X de C si es miembro.

-Unión, intersección y diferencia de conjuntos: Dados dos conjuntos, C_1 y C_2, crear el conjunto C_3 que contenga todos los miembros de C_1 y C_2 eliminando los duplicados (operación de unión), crear C_3 de modo que contenga sólo valores que sean miembros tanto de C_1 como de C_2 (operación de intersección), o crear C_3 de modo que contenga sólo valores que estén en C_1 pero no en C_2 (operación de diferencia). El acceso a componentes de un conjunto por subíndice o posición relativa no interviene el procesamiento de conjuntos.

El término Conjunto se aplica a veces a una estructura de datos que representa un conjunto ordenado. Un Conjunto Ordenado es una lista de la que se han eliminado los valores duplicados, no necesita trato especial. El Conjunto No Ordenado, tiene 2 representaciones de almacenamiento especializadas y merecen atención.

-Representación de Conjuntos por Cadenas de Bits: se usa cuando se sabe que el tamaño del universo subyacente de valores (los valores que pueden aparecer en objetos de datos de conjuntos) es pequeño. Las operaciones de unión, intersección y diferencia sobre conjuntos completos se puede representar por medio de las operaciones booleanas sobre cadenas de bits que comúnmente suministra el hardware. El apoyo de hardware para operaciones de cadenas de bits, hace eficiente la manipulación de la representación de conjuntos por cadenas de bits. Las operaciones de hardware se aplican ordinariamente sólo a cadenas de bits de una cierta longitud fija (por ejemplo, la longitud de palabra de la memoria central). Para cadenas mayores de este máximo, se debe usar simulación por software para descomponer la cadena para descomponer la cadena en unidades más pequeñas que puedan ser procesadas por el hardware.

ARCHIVOS DE ENTRADA – SALIDA. Estructura de datos con 2 propiedades:

-Se representa ordinariamente en un dispositivo de almacenamiento secundario (cinta o disco), y puede ser más grande que otras estructuras de datos.

-Su tiempo de vida puede abarcar un intervalo de tiempo mayor que el del programa que lo crea.

Los Archivos Secuenciales son el tipo más común de archivo, muchos lenguajes también suministran Archivos de Acceso Directo y Archivos Secuenciales Indizados. Existen 2 usos generales para los archivos: para entrada y salida de datos a un entorno operativo externo, y como almacenamiento borrable temporal para datos cuando no se dispone de suficiente memoria de alta velocidad. Los componentes de un archivo se llaman Registros. Un Archivo Secuencial es una estructura de datos formada por una serie lineal de componentes del mismo tipo, es de longitud variable y no tiene límite máximo fijo (más allá del almacenamiento disponible). Ejemplo: en Pascal, un archivo se declara dando su nombre y el tipo de componente que contiene → Maestro: file of RegEmpleado; define un archivo llamado Maestro cuyos componentes son del tipo RegEmpleado. El tipo de componentes puede ser un tipo elemental o un tipo de estructura de datos de tamaño fijo, como un arreglo o un registro. Cuando los datos del archivo se leen mas tarde, las ubicaciones de almacenamiento a las que hacen referencia los valores de apuntador pueden estar en uso para otro propósito.

Para Entrada - Salida, los datos se representan en forma de caracteres. Cada componente de un archivo de esta clase es por tanto un solo carácter, y el archivo se conoce en Pascal, como un Textfile. Típicamente, se puede tener acceso al archivo ya sea en el Modo de Lectura o en Modo de Escritura. En cualquiera de ellos existe un Apuntador de Posición de Archivo que designa una posición ya sea antes del primer componente de archivo, entre dos componentes, o después del último componente. En el modo de escritura, el apuntador de posición de archivo siempre está situado después del último componente, y la única operación posible es la de asignar (escritura) un nuevo componente a esa posición, con los que se amplia el archivo en un componente. En el modo de lectura, el apuntador de posición de archivo se puede situar en cualquier parte del archivo, y el único acceso que se proporciona es el acceso (lectura) al componente que está en (sigue inmediatamente a) la posición designada. No se proporciona asignación de

nuevos componentes o valores de componentes. En cualquiera de los modos, una operación de Lectura o Escritura hace avanzar el apuntador de posición de archivo a la posición que sigue inmediatamente al componente al que se tuvo acceso o que se asignó. Si el apuntador de posición de archivo está situado después del último componente, se dice que el archivo está situado en el Final de Archivo (o en la marca de final de archivo).

-Abrir: sirve para usar un archivo, se le proporciona el nombre de un archivo y el modo de acceso (lectura o escritura). Si el Modo es de Lectura, entonces se supone que el archivo ya existe. La operación abrir solicita información del sistema operativo acerca de la localidad y propiedades del archivo, asigna el almacenamiento interno requerido para buffers (memorias temporales), fija el apuntador de posición de archivo en el primer componente del archivo. Si el Modo es de Escritura, entonces se hace una solicitud al sistema Operativo para crear un nuevo archivo vacío o, si ya existe un archivo con el nombre dado, para borrar todos los componentes existentes del archivo para dejarlo vacío. El apuntador de posición se fija en el principio del archivo vacío.
-Lectura: transfiere el contenido de componente de archivo actual (designado por el apuntador de posición de archivo) a una variable designada en el programa.
-Escritura: crea un nuevo componente en la posición actual en el archivo (siempre al final) y transfiere el contenido de una variable de programa designada al nuevo componente. Una vez más, esta transferencia se define comúnmente como una forma de asignación.
-Prueba final de archivo: Una operación de lectura falla si el apuntador de posición de archivo designa el final del archivo. Puesto que el archivo es de longitud variable, se necesita una prueba explícita de la posición de final de archivo para que el programa pueda adoptar una acción especial.
-Cerrar: sirve cuando el procesamiento de un archivo ha terminado y notifica al Sistema Operativo que el archivo se puede separar del programa (y ponerse a disposición de otros programas), y posiblemente desasignar el almacenamiento interno usado para el archivo (como buffers o variables buffers). Los archivos se

pueden cerrar de manera implícita cuando el programa concluye, sin una acción específica por parte del programador. Sin embargo, para cambiar el modo de acceso a un archivo de escritura a lectura, o viceversa, con frecuencia es necesario cerrar explícitamente el archivo y luego volver a abrirlo en el nuevo modo.

ARCHIVOS TEXTFILES. El término proviene de Pascal, es un archivo de texto y de caracteres. Son la forma primaria de archivo para Entrada - Salida al usuario en casi todos los lenguajes, porque se pueden imprimir y crear a partir de entradas de teclado. Los textfiles son una forma de archivo secuencial ordinario y se pueden manipular en las mismas formas. Sin embargo, se suelen suministrar operaciones especiales para textfiles que permiten la conversión automática de datos numéricos (y a otros tipos de datos) a representaciones de almacenamiento interno. Permiten el Formateo de Salida para implementar operaciones de salida.

ENTRADA – SALIDA INTERACTIVA. Un textfile es una terminal interactiva a la cual está sentado un programador. Durante la ejecución del programa, una Operación de Lectura sobre este archivo se interpreta como un mandato para desplegar los caracteres en la pantalla en la terminal. Una Operación de Escritura es un mandato que solicita alimentación de datos desde el teclado, y que por lo común se inicia con el despliegue de un carácter de “señal de entrada” en la pantalla. Ahora, cambian aspectos de la perspectiva de los archivos secuenciales:

-El archivo debe estar tanto en el modo lectura como de escritura, al mismo tiempo, puesto que las operaciones ordinarias de Lectura y Escritura se alternan. Primero se despliegan ciertos datos; luego, se solicitan ciertos datos de entrada; y así sucesivamente.

-El uso de memoria temporal (buffer) para los datos de entrada y de salida está restringido. Rara vez se puede reunir más de un renglón de datos en el buffer de entrada antes que se procesen. Los datos que se recogen en un buffer de salida se deben desplegar antes de que se haga una solicitud de lectura a la terminal.

-El apuntador de posición de archivo y la prueba de final de archivo tienen poca importancia. Un archivo interactivo carece de posición y no tiene final, puesto que el programador puede continuar introduciendo datos de manera indefinida. El programador puede usar un carácter especial de control para marcar el final de una porción de su entrada desde la terminal, pero las nociones usuales de prueba de final de archivo y procesamiento de final de archivo suelen ser inadecuadas. A causa de estas diferencias considerables entre archivos interactivos y archivos secuenciales ordinarios, muchos diseños de lenguajes han experimentado dificultad para dar cabida a archivos interactivos dentro de una estructura de entrada - salida proyectada para archivos secuenciales ordinarios.

ARCHIVOS DE ACCESO DIRECTO. Permiten tener acceso a cualquier componente individual al azar, están organizados como un conjunto no ordenado de componentes, con el valor de su índice asociado a cada componente. Inicialmente el archivo está vacío. A una operación de escritura se le da un componente para que lo copie al archivo, así como el valor del índice que deberá asociarse a ese componente, ésta crea un nuevo componente en el dispositivo de almacenamiento externo y copia el valor designado en él. El valor del índice se asocia casi siempre con la localidad del componente.

ARCHIVO SECUENCIAL INDIZADO. Busca tener acceso a los componentes en orden a partir de un componente seleccionado al azar. Establece un compromiso entre las organizaciones secuenciales y de acceso directo puro. Un archivo secuencial indizado requiere un índice de valores de los índices o llaves, tal como ocurre para un archivo de acceso directo, pero las entradas del índice deben estar ordenadas por los valores de las llaves. Cuando una operación de lectura o escritura selecciona un componente con un valor de llave en particular, entonces esa pareja del índice se convierte en el componente actual del archivo, es decir, el apuntador de posición de archivo está situado en ese componente. Para avanzar al próximo componente de archivo en la serie, se tiene acceso a la siguiente entrada de índice, y esa entrada se convierte en el componente actual. En esta

forma, el acceso secuencial a componentes es posible sin un cambio importante respecto a la organización de acceso directo.

CAPÍTULO 5:

“ENCAPSULACIÓN”.

5.1- TIPOS ABSTRACTOS DE DATOS.

La abstracción es un proceso mental con 2 aspectos complementarios, el de destacar los detalles relevantes del objeto en estudio y el de ignorar los detalles irrelevantes del objeto (en ese nivel de abstracción). Algunos ejemplos del uso de la abstracción como mecanismo para disminuir la complejidad son:

-Los Lenguajes de Alto Nivel permitieron a los programadores abstraerse del sin fin de detalles de los lenguajes ensambladores, y trabajar de un modo independiente de las máquinas concretas.

-Las ideas de Macro en los Lenguajes Ensambladores y la de Procedimiento con Parámetros, permiten dar un nombre a un conjunto complejo de instrucciones y activarlas con una sola instrucción.

-Las construcciones creadas para sincronizar procesos en muchos lenguajes de programación concurrente abstraen al programador de la necesidad de tener en cuenta si la máquina subyacente dispone de un solo procesador, de varios que comparten una memoria común, o si consiste en una red de computadoras.

-Las especificaciones formales mediante predicados formalizan el efecto de las instrucciones de un lenguaje sin tener que descender al detalle de cómo se ejecutan en una computadora.

La Abstracción Funcional es la idea de crear procedimientos y funciones e invocarlos mediante un nombre. Se destaca qué hace la función y se ignora cómo lo hace, es decir, el algoritmo concreto y las variables auxiliares necesarias para conseguir el efecto pretendido. La abstracción produce un ocultamiento de información. Los primeros Lenguajes de Alto Nivel (Fortran, Cobol) ya tenían este mecanismo y no soportan la abstracción de procedimientos. Primero, no incorporan las instrucciones de control IF-THEN-ELSE, WHILE-REPEAT y CASE. Segundo, aunque en ellos es posible definir argumentos para subrutinas, no es posible especificar sus tipos de datos, por lo que muchos errores de interfaz, que podrían ser detectados por el compilador, deben ser eliminados manualmente por el programador. El siguiente paso fue la creación de tipos definidos por el

programador, que contribuyen a elevar el nivel del lenguaje, pues posibilitan la definición de tipos de datos cercanos al problema que se pretende resolver.
El concepto de Tipo Abstracto de Datos (por J. Guttag en 1974), clarificó esta situación. Un Tipo Abstracto es una colección de valores y de operaciones que se definen mediante una especificación que es independiente de cualquier representación. El calificativo abstracto expresa precisamente esta cualidad de independencia de la representación. Para definir un nuevo tipo, el programador debería comenzar por decidir qué operaciones le parecen relevantes y útiles para operar con las variables pertenecientes al mismo. Es decir, debería comenzar por establecer la interfaz que van a tener los usuarios con dicho tipo. Características:

-ABSTRACCIÓN. Se basa en lo que hace y no como lo hace. Define las funciones que definen nuestro objeto y da un acceso significativo.

-ENCAPSULAMIENTO. Ocultamiento o privacidad de la información, sirve para separar la interfaz de una abstracción y su implementación. Es un concepto complementario al de abstracción y da lugar a que las clases se dividan en:
.Especificación: comprende el número de componente, tipo de cada componente, nombre a ser usado para la selección de cada componente; numero máximo de componentes, organización de los componentes.
.Implementación: comprende la representación de la abstracción, así como los mecanismos que conducen al comportamiento deseado. Operaciones.

-OCULTACIÓN DE INFORMACIÓN. Ocultamos los detalles, así no se conoce como se implementa. El encapsulado evita la corrupción de los datos de un objeto, protegiéndolos del uso arbitrario y no pretendido. Permite modificar los elementos internos del objeto sin afectar a los usuarios del objeto. Para implementar un TDA un lenguaje debe proveer: una forma de definir los tipos de datos y operaciones abstractas para estos objetos, encapsular los datos, es decir, que sólo puedan ser manipulados por operaciones abstractas. Otros motivos que hacen conveniente la especificación formal de un tipo abstracto son los siguientes:

-Unanimidad de interpretación por parte de los distintos usuarios del tipo.
-Posibilidad de verificar formalmente los programas usuarios del tipo.
-Deducción, desde la especificación de propiedades satisfechas por cualquier implementación válida de tipo.

5.2- ENCAPSULACIÓN MEDIANTE SUBPROGRAMAS.

Su importancia es el diseño del programa, donde un subprograma forma los bloques básicos sobre los cuáles son construidos muchos programas y representa una operación abstracta que el programador define opuestamente a las operaciones abstractas que son definidas en el lenguaje. Y el nivel de diseño del lenguaje, es la implementación de los mecanismos que hacen posible la definición e invocación de subprogramas. Características de los Subprogramas: permite crear abstracción de proceso encapsulando código, definiendo una interfaz de invocación para paso de parámetros y resultados. También ayuda a reutilizar código, ahorrando memoria y tiempo de codificación.

LA INTERFAZ DEL SUBPROGRAMA. Es el único medio de comunicación entre el usuario y el código encapsulado en el subprograma. Sus elementos de definición:
-Nombre: permite referenciar al subprograma como unidad e invocarlo.
-Parámetros (Opcional): define la comunicación de datos (nombre, orden y tipo de parámetros formales).
-Valor de retorno: Opcional para funciones (tipificado).
LOS ELEMENTOS ESENCIALES EN EL ESTUDIO DE LOS SUBPROGRAMAS:
-La Firma (signature), contrato entre el invocador y el subprograma que define la semántica de la interfaz.
-El Protocolo especifica cómo debe realizarse la comunicación de parámetros y resultados (tipo y orden de los parámetros y, opcionalmente, valor de retorno).

SUBPROGRAMAS COMO OPERACIONES ABSTRACTAS. Su definición cuenta con la Especificación y la Implementación, ambas las da el programador. La

Especificación de un subprograma deberá poder ser entendida sin la necesidad de entender como es éste implementado. La Implementación de un subprograma se realiza igual que la de una operación primitiva, incluyendo: el Nombre del subprograma, la Signatura (o prototipo) del subprograma dando el número de argumentos, su orden, el tipo de ellos, como el número de resultados, su orden y tipo de cada uno de ellos; y la Acción realizada por el subprograma (como la descripción de la función que computa).

Un subprograma representa una función matemática que mapea un determinado conjunto de argumentos en un particular conjunto de resultados. Cuando un subprograma regresa un solo objeto de datos como resultado se llama Función Subprograma o Función. Algunos lenguajes emplean el uso de “keywords” en la declaración, como son las palabras reservadas procedure o function, como en la siguiente declaración en Pascal: **function** FN(X: real; Y: integer): real

Un Procedimiento o Subrutina, es cuando un subprograma regresa más de un resultado o modifica sus argumentos para entregar resultados. Sintaxis en C es:

void Sub(float X, int Y, float *Z, int *W)

Un programa representa una función matemática y el problema es determinar la función que computa porque:

-Un subprograma puede tener implícitamente argumentos en forma de variables no locales que éste emplee.

-Un subprograma puede tener resultados implícitos entregados en forma de modificaciones no locales o bien en cambios de sus argumentos de entrada-salida.

-Un subprograma no puede modificar su ejecución ordinariamente dados los valores de los argumentos de entrada, sin embargo, sí puede transferir el control a manejadores de ejecuciones o bien terminar con la ejecución abruptamente.

-Un subprograma deberá ser sensitivo a la historia de sus ejecuciones, es decir, deberá almacenar variables locales entre invocaciones, lo que le permitirá responder en base a las invocaciones anteriores que ha tenido.

IMPLEMENTACIÓN DE UN SUBPROGRAMA. Un subprograma representa una operación del nivel de computadora virtual construido por el programador, se implementa usando las operaciones y estructuras de datos que posee el Lenguaje de Programación. La implementación es el cuerpo del subprograma, éste consiste en declaraciones de datos locales que definen las estructuras de datos usadas por el subprograma, además de una serie de instrucciones que definen las acciones que tendrán efecto cuando el programa se ejecute. Estas declaraciones e instrucciones se encuentran encapsuladas de tal forma que ninguna de ellas podrá ser empleada por separado de la ejecución del subprograma; el usuario sólo podrá invocar al subprograma con un conjunto de argumentos determinado y por tanto recibir los resultados del cálculo. El cuerpo de un subprograma en C es típico:

```
float FN(float X, int Y)          - Signatura del programa
{
        float M(10); int N;       - Declaración de tipos de datos locales.
.       - Secuencia de instrucciones, acciones que realiza el subprograma
}
```

La invocación de subprogramas deberá hacerse mediante el paso de los argumentos apropiados, enumerados en la especificación del mismo. Por tanto los resultados que regresan podrán ser conocidos. La conversión y verificación de tipos se podrá llevar a cabo estáticamente en la traducción del programa; o se puede llevar a cabo de una forma dinámica durante la ejecución del programa. En este caso, los mecanismos de conversión entre tipos deberán ser proporcionados automáticamente por el lenguaje.

DEFINICIÓN E INVOCACIÓN (activación) DE UN SUBPROGRAMA. La definición de un subprograma es una propiedad estática de un programa. Durante la ejecución del programa, si un subprograma es llamado Invocado, una activación del subprograma es creada. Cuando se termina de ejecutar el programa, la activación del subprograma es destruida. Si se genera otra llamada, una nueva activación es creada. De una sola definición del subprograma se pueden generar múltiples activaciones, la definición sirve como una plantilla para la creación de

activaciones durante la ejecución. La diferencia entre definiciones y activaciones de subprogramas es importante. Una definición se encuentra almacenada dentro de un programa de una forma escrita, y es la única información que se tiene de un subprograma cuando es traducido. Por otra parte, las activaciones de los subprogramas sólo existen durante la ejecución del programa.
Un subprograma representa un bloque de almacenamiento cuando éste se ejecuta, en éste bloque se almacenan componentes y datos que son relevantes para la activación del subprograma. El ciclo de vida de una activación se lleva a cabo entre la llamada, momento en el que se la activación se crea y se ocupa la memoria, y el resultado o final del subprograma, tiempo en el que la memoria es liberada y la activación es destruida.

Los componentes que requiere en tiempo de ejecución la activación de un subprograma son dados por la definición:
-La signatura es información del almacenamiento requerido para parámetros y resultados de la función.
-La declaración es información sobre el almacenamiento requerido para variables locales.
-El almacenamiento de literales y variables constantes.
-El almacenamiento del código ejecutable generado por las instrucciones incluidas en el cuerpo del programa.

La definición del subprograma contiene la información necesaria para generar la organización de áreas de almacenamiento y código ejecutable durante la traducción. El resultado de la traducción será una plantilla que será empleada para construir cada activación durante el tiempo de ejecución. Para construir una activación a partir de la plantilla del subprograma sería necesario copiar toda la plantilla a una nueva localidad de memoria. Para evitar hacer una copia completa de la plantilla, la podemos dividir en dos partes:

-Parte Estática (segmento de código) compuesta de las constantes y el código ejecutable. Permanece invariante durante la ejecución del subprograma, y se emplea una sola copia de ésta plantilla por todas las activaciones.
-Parte Dinámica (registro de activación) compuesto por los parámetros, resultados de funciones, datos locales, y otros datos de carácter local como áreas de almacenamiento local, apuntadores y ligas a variables no locales. Esta parte es igual para cada activación, sin embargo los valores que contiene varían de una a otra. Cada activación debe poseer su propio registro de activación.

El tamaño y estructura del registro de activación requerido para cada subprograma puede ser determinado desde la traducción de éste. Un registro de activación es representado como un record data object. Cuando un subprograma es llamado en tiempo de ejecución, el único dato que requiere para crear la activación es el tamaño del registro de activación. No obstante, cuando un subprograma es llamado, se llevan a cabo acciones que no son vistas por el usuario como son, la creación del registro de activación, la transmisión de parámetros, la creación de ligas para variables no locales, además de otras actividades. Estas acciones se llevarán a cabo antes de ejecutar las instrucciones contenidas en el cuerpo del subprograma. La ejecución de estas tareas, se realiza ejecutando instrucciones insertadas a manera de “prologo” por el traductor, como un bloque de código al principio del subprograma. Una vez terminada la ejecución del subprograma una serie de tareas similares son llevadas a cabo para liberar el espacio empleado por el registro de activación. Estas acciones son incluidas por el traductor a manera de “epílogo” como un bloque de código al final del subprograma.

SUBPROGRAMAS GENÉRICOS. Un solo nombre y muchas definiciones. Pueden estar sobrecargados.

DEFINICIÓN DE SUBPROGRAMAS COMO OBJETOS. En lenguajes complicados (C,C++,Java) la definición del programa es independiente de su ejecución. Durante la ejecución del subprograma, la definición del mismo es invisible e

inaccesible. Los lenguajes que emplean intérpretes (LISP,Prolog,Perl) no hacen distinción entre estas dos fases.
La traducción es un proceso que toma la definición del subprograma en forma de una cadena de caracteres y produce un run-time data object que representa la definición. La ejecución es una operación que toma el run-time data object y realiza la activación. En la interpretación, la activación del subprograma se considera como una metaoperación que tiene lugar antes de la ejecución de todo el programa. En Prolog y LISP el llamado de un subprograma se realiza en tiempo de ejecución a partir de código fuente que es convertido en un ejecutable del cuerpo del subprograma. En ambos lenguajes es posible comenzar con la ejecución de un programa sin tener un subprograma en particular. Durante el tiempo de ejecución el subprograma será leído y convertido a un código ejecutable.

COMPILACIÓN SEPARADA E INDEPENDIENTE. Son formas para hacer la compilación de los subprogramas e integrarlos a un sistema.
-Compilación Separada: unidades de programas pueden compilarse en diferentes tiempos, pero se consideran dependencias (comprobación: interfaces, variables) de acuerdo a lo que exporta e importa (ADA).
-Compilación Independiente: Se compilan unidades de programa sin información de otras (C, C++ y Fortran).
Es más eficiente la compilación separada, porque permite la anexión de código o el cambio de los subprogramas en un tiempo posterior a la compilación.

5.3- DEFINICIÓN DE TIPOS.

Es la definición de un nuevo tipo de dato abstracto, requiere mecanismos para la definición de una clase de un objeto dato. Una Definición de Tipo no define completamente un tipo de dato abstracto, porque este no incluye la definición de las operaciones sobre los datos de ese tipo. En una Definición de Tipo, el nombre de tipo es dado junto con una declaración que describe la estructura de una clase

del objeto dato, y cuando un objeto dato en particular de esa estructura es necesitado, sólo es necesario tener el nombre del tipo y no repetir la descripción completa de la estructura de datos. La Definición de Tipos simplifica la estructura de los programas, permite que la modificación sea hecha sólo en la definición del tipo y no en la definición de cada variable. También, si el objeto dato es pasado como un argumento a un subprograma, sólo necesitamos usar el nombre del tipo y no incluir la descripción del argumento en la definición del subprograma. La Definición de Tipos es usada como un molde para construir objetos dato durante la ejecución de un programa. Un objeto dato de un tipo puede ser creado en la entrada a un subprograma (si está en la declaración de variables locales del subprograma), o puede ser creada dinámicamente usando una operación de creación como la operación malloc. Es una nueva forma de encapsulamiento y ocultamiento de información. Un subprograma puede declarar una variable de un tipo sólo utilizando el nombre del tipo.

La definición del tipo oculta la estructura interna del objeto dato de ese tipo. Si un subprograma solo crea objetos de ese tipo, usando el nombre del tipo, pero no accede a los componentes internos de los objetos dato, entonces el subprograma es independiente a la estructura particular declarada en la definición de tipos. La definición puede ser modificada sin cambiar el subprograma. Si el diseño del lenguaje tiene la restricción de que sólo unos pocos subprogramas designados pueden acceder a los componentes internos de los objetos datos, entonces la definición de tipo tiene efectivamente encapsulada la estructura de los objetos dato de ese tipo.

La información contenida en la declaración de una variable es usada durante la traducción para determinar la representación de almacenamiento para el objeto dato, para el manejo del almacenamiento y la verificación de tipos. Estas declaraciones no están presentes durante el tiempo de corrida, son utilizados solamente en la actualización de los objetos dato apropiados en tiempo de corrida. Una definición de tipo, de forma similar, es usada solamente durante la traducción.

El traductor del lenguaje introduce la información de una definición de tipo a una tabla durante la traducción y, en donde el nombre del tipo es referenciado en una declaración subsecuente, usa la información de la tabla para producir el código ejecutable apropiado para el establecimiento y manipulación del objeto dato deseado durante la ejecución. La definición de tipos permite algunos aspectos de traducción, como es determinar la representación del almacenamiento, hacerlo sólo una vez para una simple definición de tipo y no muchas veces para diferentes declaraciones.

EQUIVALENCIA DE TIPOS. La verificación de tipos, siendo estática o dinámica, involucra una comparación entre el tipo de dato del argumento actual dado a una operación y el tipo de dato del argumento que es esperado por la operación. Si los tipos son iguales, entonces el argumento es aceptado y la operación procede; si es diferente, entonces es considerado un error o una conversión es usada para convertir el tipo de argumento e igualarlo al que era esperado.

-Equivalencia de Nombres: dos tipos de datos son equivalentes sólo si tienen el mismo nombre. De esta manera, los tipos Vect1 y Vect2 son de diferente tipo, aunque los objetos dato definidos tienen la misma estructura. Una asignación del tipo X:=Z es válida pero X:=Y no lo es. La equivalencia de nombres de tipos es el método utilizado en Ada, C++ y por parámetros de subprogramas en Pascal. La equivalencia de nombres tiene desventajas porque cada objeto usado en una asignación debe tener el mismo nombre de tipo y no pueden tener tipos anónimos. Una definición de tipo puede servir en todo o en gran parte de un programa, una vez que el tipo de un objeto dato es pasado como un argumento a una serie de subprogramas, no puede ser definido nuevamente en cada subprograma.

EQUIVALENCIA ESTRUCTURAL. Dos tipos de datos son equivalentes si definen objetos dato que tengan los mismos componentes internos, estos componentes significan que la misma representación de almacenamiento puede ser utilizada para ambas clases de objetos dato. La equivalencia estructural no tiene las desventajas de la equivalencia de nombres, pero tiene sus propios problemas:

-Cuando dos tipos son estructuralmente equivalentes.
-Dos variables son estructuralmente equivalentes, pero el programador las declara como tipos separados.
-Determinando si dos definiciones de tipo complejo son estructuralmente equivalentes, si frecuentemente se hace, puede ser una parte costosa de la traducción.
Los aspectos involucrados en la elección de una definición de equivalencia de tipos son importantes en el diseño de lenguajes, como en Ada y Pascal donde la definición de tipos juega un rol central. En lenguajes anteriores como FORTRAN, COBOL y PL/1, no tienen definición de tipos y sin embargo algunas formas de equivalencia estructural son usadas.

IGUALDAD DE OBJETOS DATO. Cuando el compilador determina que dos objetos son del mismo tipo.
-Igualdad de Stac (TopStack) se refiere a los objetos dato en Dato, hablamos del tope de la pila, la igualdad entre X y Y sería: X.TopStack = Y.TopStack Para toda I entre 0 y TopStack –1: X.Data[I]=Y.Data[Y] X y Y representan stacks equivalentes.
-Igualdad de set: NumberInSet se refiere al número de objetos dato en los sets A y B, entonces la igualdad entre A y B se definiría como: A.NumberInSet = B.NumberInSet
A.Data[0] ... A.Data[NumberInSet –1] es la permutación de B.Data[0] ...
B.Data[NumberInSet-1], una vez que el orden de inserción de los elementos no es relevante.
La forma en que planteamos las operaciones de stack se ve como operaciones push y pop y las operaciones de set se ven como insertar y borrar. No se cuenta con un mecanismo que fácilmente formalice la igualdad de objetos dato complejos. La forma usual para la construcción de datos definidos por el programador es incluir una operación separada para los “iguales”. Por eso, si se está construyendo un tipo de dato stack, además de las operaciones usuales de *pop, push, empty* y

top, es necesario incluir la operación *StackEquals* (Igualdad de Stack) si esta función es utilizada en el programa.

DEFINICIÓN DE TIPOS CON PARÁMETROS. Cuando el compilador traduce la declaración de una variable con una lista de parámetros después del nombre de tipo, el compilador primero pone el valor del parámetro en la definición de tipo para tener una definición completa de tipo sin parámetros. Los parámetros en la definición de tipos tienen unas pocas veces efecto en la organización de la implementación del lenguaje en tiempo de ejecución, sólo en donde un subprograma puede aceptar un objeto dato como un argumento de tipo parametrizado, y así preparase para la posibilidad de un argumento de diferente tamaño al llamado.

CAPÍTULO 6:

“HERENCIA”.

6.1- CONCEPTO, BENEFICIOS Y FACETA DE LA HERENCIA.

Busca hacer el software re-usable y extensible. Sirve a los diseñadores para crear nuevos módulos de software, evitando rediseñar y decodificar todo desde cero. Nuevas clases pueden heredar el comportamiento (operaciones, métodos) y la representación (instanciación de variables y atributos) de clases existentes.

Heredando el comportamiento permitimos compartir código (la esencia de la reusabilidad) sobre módulos de software. La representación de la herencia establece la compartición de estructuras sobre objetos de datos y también provee un mecanismo muy natural para organizar la información. Esto es establecer objetos "taxonómicos" dentro de una buena definida jerarquía de herencia.

La herencia tiene jerarquía y se ejemplifica a continuación:

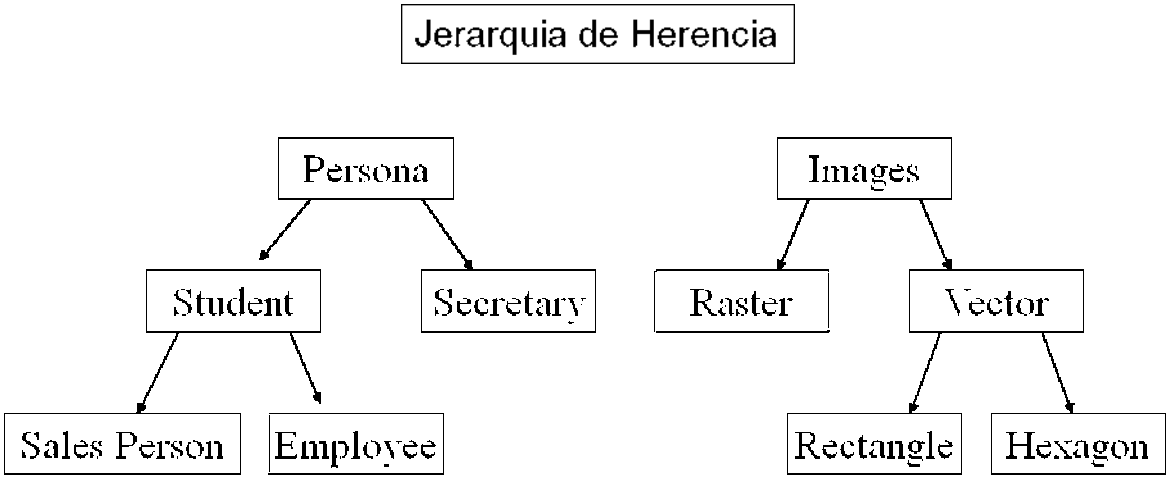


FIG 6: Jerarquía de Herencia.

BENEFICIOS DE LA HERENCIA. Característica de la programación orientada a objetos y herramienta poderosa para el programador, tiene múltiples ventajas:

-Reusabilidad de Software: el comportamiento es heredado de otra clase, el código que provee este comportamiento no tiene que ser re-escrito.

-Compartir código: puede ocurrir en muchos niveles usando las técnicas de orientación a objetos. En un nivel, muchos programadores separados o proyectos pueden usar la misma clase y esto se refiere a componentes de software.

También, cuando dos o más clases diferentes son desarrolladas por un programador como parte de un proyecto, heredado de una clase padre sencilla.

-Consistencia de la Interfase: Cuando se tiene múltiples herencias de una misma superclase, el comportamiento de la herencia es la misma en todas las clases.

-Componentes de Software: La herencia permite construir componentes de software re-usable.

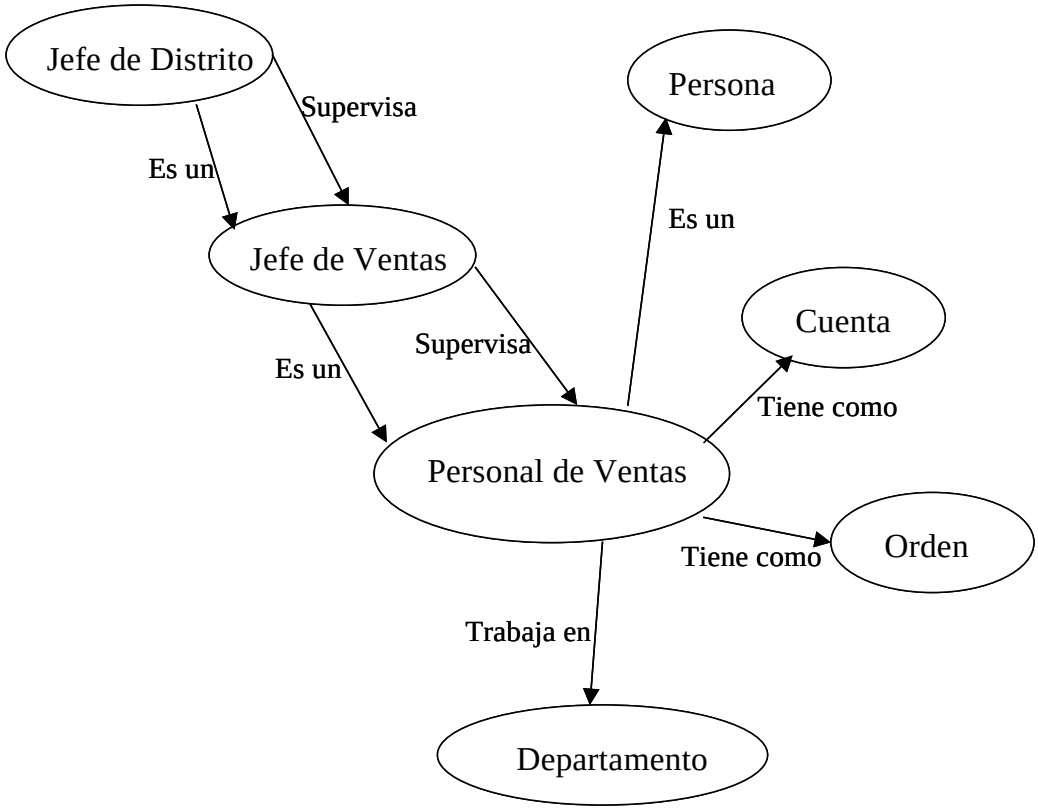
-Prototipos rápidos: Cuando tenemos un sistema de software pueden ser construidos grandemente por componentes de software reusables, el tiempo del desarrollador puede ser concentrado para entender la porción del sistema que es nuevo o inusual.

-Polimorfismos: La producción de software es una manera convencional es generalmente escrita en un diagrama bottom-up, pero con la herencia puede ser descrita con un diagrama top-down. Esto es que podemos escribir el sistema a partir de un nivel de abstracción muy grande hacia las rutinas de bajo nivel. El polimorfismo permite al programador generar componentes re-usables en alto nivel que puede ser usado para diversas aplicaciones con sólo cambiar algunas partes de bajo nivel.

-Ocultación de Información: el programador re-utiliza los componentes de software, ellos solo necesitan entender la naturaleza del componente y su interfase. No es necesario que el programador tenga la información detallada de la clase heredada.

HERENCIA COMO REPRESENTACIÓN DEL CONOCIMIENTO. La herencia es la raíz del “sentido común” del paradigma de la representación del conocimiento usado en la Inteligencia Artificial. Las redes semánticas consisten de nodos que representan conceptos (objetos) y ligas que representan relaciones. En la representación de redes semánticas, los nodos y las ligas tienen etiquetas. La más poderosa de las etiquetas representa las relaciones de herencia, esta es simbolizada como “es un”. La herencia es una clase de relación entre conceptos. Otro tipo de ligas de relaciones pueden representar atributos (instanciación de

variables) o conceptos (objetos). A continuación, se presenta un ejemplo sencillo y claro de herencia como representación del conocimiento:



Red Semántica para Personal de Ventas

FIG 7: Representación de sistema empresarial a través de la herencia.

DIFERENTES FACETAS DE LA HERENCIA. La herencia es la poderosa técnica que organiza bases de código complejo y ayuda a la construcción de nuevas clases. Es rica en la semántica de las relaciones sobre entidades, en un espacio de objetos que pueden ser expresados directa y naturalmente. Introduce complejidad y se puede integrar con más conceptos de orientación a objetos como la encapsulación, tipos, visibilidad y otros estados de los objetos.

-Herencia y Subtipos: usados en lenguajes orientados a objetos con diferencias.

-Visibilidad de Variables Heredadas y Métodos: En lenguajes orientados a objetos (Simula) permiten la manipulación directa de las variables instanciadas. Otros

lenguajes (C++) distinguen entre la manipulación pública y privada. Con la herencia hay la 3era alternativa, pide la visibilidad en la subclase.

-Herencia y Encapsulación: La visibilidad de las variables instanciadas violan el principio de ocultamiento de información, entonces tenemos un conflicto entre la herencia y la encapsulación, si las instancias son de las variables de la superclase son accesadas directamente.

-Especialización: En las clases existentes. Las clases pueden ser especializadas con extender su representación (instanciación de variables) o comportamiento (operaciones).

-Herencia de Objetos: casi todos los lenguajes orientados a objetos soportan la herencia de clases (la habilidad de una clase para tener una representación heredada y métodos de otra clase). Se tienen varios modelos de computación que incorpora operaciones con objetos y uso solamente con herencia de objetos para la organización de espacios de objetos. Esto es llamado Sistema Prototipo.

-Herencia Múltiple: cuando es necesaria la herencia de más de una clase. Tenemos una clase heredera de más de un padre y la posibilidad de conflictos: métodos o variables instancia da con un mismo nombre pero con semántica diferente o no relacionada que son heredadas de diferentes superclases.

6.2- MECANISMOS Y CLASIFICACIÓN DE LA HERENCIA.

MECANISMOS DE HERENCIA. Herencia es el mecanismo de compartir atributos entre un hijo y su padre, en herencia el hijo hereda algunos atributos del padre. En los lenguajes OO un mecanismo natural de herencia es extensión de atributos entre los objetos existentes.

-Embebido: hace una copia de los atributos del objeto donador, esto provee de una simple explicación de la semántica estándar de self como receptor de la invocación.

-Delegando: es una redirección del acceso del archivo e invocación del método de un objeto o prototipo a otro de tal manera que un objeto puede ser visto como una extensión de otro.
-Herencia dinámica: Es creada cuando la liga a los padres puede ser actualizada dinámicamente.

CLASIFICACIÓN DE LA HERENCIA. Se clasifica en herencia de clases (class) y objetos, de tipos (encapsulamiento, datos abstractos) y de métodos.
Tipo de datos abstracto es un nuevo tipo de datos definido por el programador que incluye: un tipo de datos definidos por el programador, un conjunto de operaciones abstractas sobre objetos de ese tipo, y encapsulamiento de objetos de ese tipo, de tal manera que el usuario del nuevo tipo no pueda manipular esos objetos excepto a través del uso de operaciones definidas.

TIPOS DE DATOS ABSTRACTOS GENÉRICOS. Los tipos primitivos de datos integrados en un lenguaje, permiten declarar el tipo básico de una clase nueva y luego especificar atributos de los objetos de datos. La definición genérica de tipo abstracto permite especificar por separado un atributo del tipo de esta clase. Una definición genérica de paquete representa una plantilla que se puede usar para crear tipos de datos abstractos particulares. Al proceso de crear la definición particular del tipo a partir de la definición genérica se llama Ejemplarización. La clasificación de la herencia en los lenguajes OO, puede ser variable, hay dos aspectos importantes:
-Herencia de estructura: son los tipos de datos definidos en la clase y propiedades de los mismos.
-Herencia de comportamiento: es la herencia de funciones y procedimientos que forman parte del objeto.

6.3- OBJETOS Y MENSAJES.

Smalltalk representa un enfoque hacia el desarrollo de objetos y métodos. Un programa en Smalltalk se compone de un conjunto de definiciones de clase integradas por objetos de datos y métodos, además de tres características primordiales:

-Definiciones de clase: son enunciados ejecutables que definen la estructura interna y los métodos.

-Ejemplarización de objetos: son objetos específicos para cada definición de clase invocando métodos de creación dentro de la definición de clase.

-Paso de mensaje: los métodos se pasan como mensajes a un objeto para llevar a cabo una acción.

Existen 3 tipos de mensajes en Smalltalk: Un Mensaje Unario es un método que no tiene parámetros, un Mensaje Binario se usa para operadores aritméticos y los Mensajes de Palabra Clave que son funciones o procedimientos que se aplican dependiendo del objeto.

HERENCIA DE CLASES. Los datos en Smalltalk se basan en una jerarquía de clases. Si un método que se pasa a un objeto no está definido dentro de esa clase, se pasa a la clase progenitora, y así sucesivamente. La clase OBJETO es la superclase progenitora de todas las clases. La herencia de métodos es una característica primitiva en Smalltalk.

CONCEPTOS DE ABSTRACCIÓN. La herencia proporciona un mecanismo para pasar información entre objetos y clases relacionadas. Si A => B significa que B es una clase relacionada con A. ¿Cuál es la relación entre objetos de A y objetos de B?

-Especialización: El objeto derivado B obtiene las propiedades más precisas presentes en el objeto A. Es la forma de herencia más común.

-Descomposición: Separa una abstracción en sus componentes. Mecanismo típico de encapsulamiento en lenguajes como Ada sin la herencia de métodos.

-Ejemplarización: permite crear concurrencias de una clase.
-Individualización: Objetos similares agrupados unos con otros (propósito común).

6.4- METACLASES Y HERENCIA MÚLTIPLE.

Una clase contiene la descripción de la estructura y el comportamiento de la instancia. En muchos lenguajes OO, las clases son fábricas que crean e inicializan instancias. EJEMPLO Smalltalk: una instancia de una clase C es creada a través del mensaje *new.* iC := C new. Aquí la clase es tratada como un objeto que puede crear instancias de otros objetos. En unos lenguajes OO hay 2 tipos de objetos:
-Objetos Clase: Objetos que pueden actuar como modelo y crear instancias de ellos mismos.
-Objetos Terminales: Objetos que solamente pueden ser instanciados pero no pueden crear instancias para otros objetos.

METACLASES. Son clases para dar origen a clases (no todos los lenguajes soportan el concepto de metaclase). Las clases no son la primera clase de los objetos ya que ello mismo no son clases de instancias, no son creadas por un mensaje y tampoco pueden recibir uno. Para que todas las entidades sean objetos, dos tipos de generadores existen y son los objetos clase y objetos terminales. Hay ventajas al tratar las clases como objetos:
-Las clases pueden ser usadas para almacenar un grupo de información, si la clase es tratada como un objeto, la información global de todas las variables instanciadas de la clase puede ser almacenada en una clase “class instance variable” (llamada class variable en Smalltalk). Métodos asociados con la clase (llamados class methods) pueden ser usados para extraer o actualizar los valores de la class variable.
-Los objetos clase se usan en la creación e inicialización de nuevas instancias de la clase. El mensaje new que es enviado al objeto clase para crear una nueva instancia, puede incorporar nuevos argumentos.

Existen lenguajes OO que persiguen en el soporte de metaclases lo siguiente: Soporte explícito para crear e instanciar metaclases y soporte implícito de metaclases. El modelo de Smalltalk-76 introdujo el concepto de metaclases. La herencia simple esta dada y la raíz de una gráfica de herencia es llamada Objeto. Lenguajes OO sin el concepto de metaclases: C++, C, Object pascal.

HERENCIA MÚLTIPLE. Mecanismo que permite que una clase se le herede más de un padre y permite combinar muchas clases existentes que utilizan. Extiende las estrategias en la unión de todos los padres inmediatos, más específicamente, para cada Ci es un predecesor inmediato de C. Los lenguajes OO la definen, pero los 1eros lenguajes no la tienen implementada. Los métodos de C son definidos como: métodos de C = métodos locales de C.
La herencia múltiple presenta el siguiente problema: Si se crean dos clases a partir de una misma superclase y se crea una cuarta clase a partir de estas dos últimas, entonces habrá un problema al referirse de los métodos heredados, ya que no se sabrá a que método se hará referencia pues se llaman igual. También existe este problema para cualquier clase con herencia múltiple cuyos miembros heredados se llamen igual.

Estrategias para resolver el problema, consisten en mecanismos implementados por el Lenguaje de Programación, no todas se pueden aplicar en los lenguajes, por esto en algunos de estos no se implementa la herencia múltiple. Ejemplos:
-Linearización: establece un orden jerárquico lineal en la herencia, el compilador establece una regla para saber a cual objeto se va a hacer referencia y si en esta superclase no existe este método o variable, entonces consultará con la siguiente superclase.
-Prohibir Conflictos: cuando el compilador detecta que hay 2 métodos o variables con el mismo nombre, este marca un error.
-Renombrar Variables Instancias y Métodos: el lenguaje le permite al usuario renombrar los diferentes métodos y variables de las clases heredadas.

-Conocer Operaciones para los Subtipos: documentar al usuario de los nombres de los métodos y variables de las clases que se van a heredar (independiente al lenguaje).
-Tipo de Dato Protegido (Protected): es el nivel intermedio de protección entre el acceso público y el acceso privado. Los miembros y “friend” de la clase base, y los miembros y “friend” de las clases derivadas son los únicos que pueden acceder a los miembros “protected” de una clase. Protected rompe con el encapsulamiento.
-Clase Amigo “Friend”: pueden acceder a elementos declarados en la parte privada “private” y protegida “protected” de la clase a la cual declaro como amigo. En C++ se declaran a las clases amigo con la palabra reservada “friend”. También con esta palabra se pueden declarar a funciones como amigo. Esto se utiliza principalmente para la sobrecarga de operadores.

TIPOS DE INFORMACIÓN QUE EN C++ SE HEREDA. Cuando la herencia es de tipo Public los tipos quedan iguales. Cuando la herencia es de tipo protected, la parte publica y protegida de los elementos heredados quedan de tipo protected. Cuando la herencia es de tipo private, la parte pública y protegida de los elementos heredados quedan de tipo private.

CAPÍTULO 7:

“CONTROL DE SECUENCIA”.

Las estructuras de control proveen el marco en el que las operaciones y los datos se combinan en programas y conjuntos de programas. Controlan el orden de las operaciones, tanto primitivas como las definidas por el usuario.

7.1- SECUENCIA DE EXPRESIONES ARITMÉTICAS Y NO ARITMÉTICAS.

REPRESENTACIÓN DE EXPRESIONES EN TIEMPO DE EJECUCIÓN. La representación de árbol de una expresión permite al traductor realizar elecciones eficientes para la evaluación. La traducción de expresiones se realiza en dos etapas, la 1era establece la representación de árbol de la expresión y la 2da (opcional) elige el orden de evaluación eficiente para ésta. La traducción de expresiones en su representación de árbol presenta y el procedimiento básico de traducción es sencillo. La 2da etapa de traducción, en que el árbol es traducido en una secuencia ejecutable de operaciones primitivas, presenta varios problemas:

1-REGLAS DE EVALUACIÓN UNIFORME. Uno espera aplicar una regla de evaluación todo el tiempo para evaluar las expresiones.

-Evaluación Ávida o Glotona: para cada nodo de operación en el árbol, 1ero se evalúan los operandos (o se genera código para evaluarlos) y luego se aplica la operación (o se genera código para aplicarla) a los operandos evaluados. El orden exacto en que los operandos son evaluados no importa, así que el orden de evaluación de los operandos o de operaciones independientes puede elegirse a manera de optimizar el almacenamiento temporal o para optimizar otros aspectos. No siempre se puede aplicar esta regla de evaluación.

-Evaluación Apática o Tardía: nunca se evalúan los operadores antes de aplicar la operación, siempre se pasan los operandos sin evaluar y la operación decide qué evaluaciones son necesarias. Funciona en teoría, pero su implementación es impráctica en muchos casos. Se requiere simular con software el paso de operandos para lograrlo. Los lenguajes explicativos (LISP,Prolog) la usan mucho, pero para los lenguajes aritméticos (C,FORTRAN) su soporte es prohibitivo. Estas

dos reglas de evaluación uniforme corresponden a dos técnicas comunes de paso de parámetros a subprogramas: por valor y por referencia, respectivamente. Ninguna regla de evaluación uniforme es satisfactoria. Comúnmente las implementaciones de lenguajes usan una mezcla de ambas técnicas.

2- EFECTOS SECUNDARIOS. Una posición propone que no deben permitirse, ya sea prohibiendo sus funciones o dejando indefinido el valor de las expresiones en las que un efecto secundario puede crear ambigüedades. La otra posición propone que deben permitirse y que la definición del lenguaje debe especificar el orden de evaluación para que el programador pueda aprovecharse de los efectos secundarios. El problema es que hacen imposibles muchas optimizaciones y sus implementaciones proveen interpretaciones conflictivas, aunque ordinariamente los efectos secundarios son permitidos.

3- CONDICIONES DE ERROR. Pueden aparecer en operaciones primitivas (overflow, división por cero), su solución varía de un lenguaje a otro e incluso de una implementación a otra.

4- EXPRESIONES BOOLEANAS DE CORTO CIRCUITO. Muchos errores de programación se deben a la creencia de que el operando izquierdo de una operación booleana hará corto circuito al resto de la evaluación, si el valor de la expresión puede decidirse del operando izquierdo. Una solución en Ada es incluir dos operaciones booleanas especiales, and then y or else, las cuales hacen corto circuito explícitamente, además de las operaciones booleanas ordinarias and y or, que no hacen corto circuito.

7.2- CONTROL DE SECUENCIA ENTRE ENUNCIADOS.

-Control de Secuencia entre Instrucciones: cada instrucción (statement) es una unidad que representa un paso de una computación. Las instrucciones normalmente contienen expresiones.

Asignaciones a Objetos de Datos: al asignar valores nuevos a objetos de datos, cambia el estado de nuestro programa. Este paso fundamental tiene variantes:
-Instrucción de Asignación Explícita: definida para cualquier tipo de dato primitivo y asigna el valor del objeto de data a su localidad en memoria.
-Instrucciones Input: leen información del teclado y de archivos de datos, llevan una o más asignaciones.
-Otras Instrucciones de Asignación: en SNOBOL4 cualquier referencia al variable INPUT causa la asignación de un nuevo valor a este variable. En Prolog el proceso de resolución implica asignaciones implícitas a variables.

TIPOS DE CONTROL DE SECUENCIA AL NIVEL DE INSTRUCCIÓN. Las formas de control de secuencia al nivel de instrucciones son:
-Composición: las instrucciones son ejecutadas en el orden que aparecen.
-Alternación: seleccionar una de varias secuencias alternativas de instrucciones.
-Iteración: ejecutar una secuencia de instrucciones 0 o más veces.

CONTROL EXPLÍCITA DE SECUENCIA. La forma más antigua es usar la instrucción GOTO. Hay 2 formas principales de GOTO y se implementan fácil con una o pocas instrucciones al nivel de código de máquina.
-GOTO Incondicional: GOTO label; donde label (etiqueta) indica a que instrucción se va a transferir la control de ejecución.
-GOTO Condicional: IF (expresión booleana) THEN GOTO label; donde label (etiqueta) indica a que instrucción se va a transferir el control de ejecución, si la expresión booleana es verdadera.
Otras instrucciones implementadas en lenguajes son BREAK y CONTINUE.
-BREAK: fuerza un salto en el control de ejecución hacia el final de la estructura de control actual, se usa para salir de una instrucción de FOR, WHILE o SWITCH.
-CONTINUE: el control salta a la próxima iteración de un loop de WHILE o FOR.

DISEÑO DE PROGRAMACIÓN ESTRUCTURADA. Diseña programas y enfatizan:
-Diseños jerárquicos de estructuras del programa usando solamente composición, alteración e iteración.
-Representación directo del diseño jerárquico en el texto de la programa.
-La secuencia de ejecución que corresponda a la secuencia de las instrucciones en el texto del programa.
-Cada grupo de instrucciones en el programa tiene solamente una función.
La ventaja de la programación estructurada es que los programas son más fáciles de entender, de depurar y de mantener.

El Programa Primo (por Maddux) describe una teoría consistente de las estructuras de control, es una generalización de programación estructurada para definir la descomposición jerárquica única de un diagrama de flujo.
Los Diagramas de Flujo tienen 3 clases de nodos, los Nodos de Función que son computaciones realizadas por un programa y se representan como cajas con un solo arco de entrada y un solo arco de salida. También representan una instrucción de asignación, que implica un cambio de estado en la máquina virtual después de su ejecución. Los Nodos de Decisión son representados como cajas en forma de rombo con un solo arco de entrada y dos arcos de salida etiquetados como falso y verdadero. Las etiquetas representan predicados y controlan el flujo de salida de una decisión booleana. Un Nodo de Unión se representa como un punto donde dos arcos llegan juntos para formar un solo arco de salida.

El Programa Propio es nuestro modelo formal de una estructura de control, tiene un solo arco de salida y tiene una ruta desde el arco de entrada a cada nodo y desde cada nodo al arco de salida.
El Programa Primo es un programa propio que no puede subdividirse en programas propios más pequeños. Ejemplo de su forma de representación:

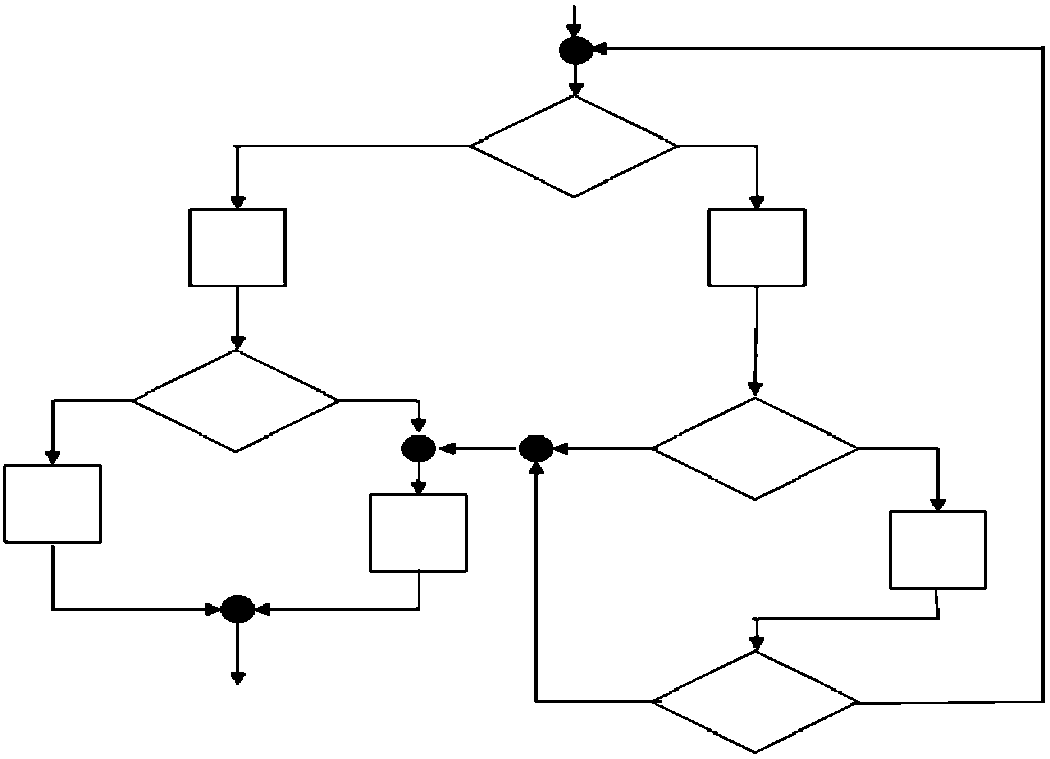


FIG 7 : Diagrama de un programa propio.

El Programa Compuesto es un programa propio que no es primo, se puede reemplazar cada componente primo por un nodo de función y repetir este proceso hasta obtener una descomposición única de cualquier programa compuesto. Todos los primos se pueden enumerar, empezando por los programas de un solo nodo, luego los de dos nodos, etc. La mayoría de estos programas no hacen nada (no cambian el estado de la máquina virtual) o representan las estructuras de control ya definidas. Los primos que no hacen nada son inefectivos, sólo dos de ellos no se ciclan. Ejemplo de su forma de representación:

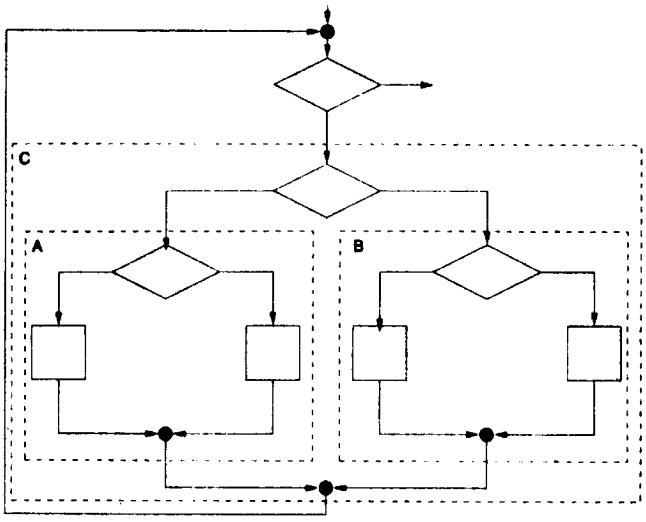


FIG 8: Diagrama de programa compuesto.

Los lenguajes de programación están definidos con las estructuras de control descritas. Estas estructuras de control representan primos con pocos nodos. El único primo ignorado por muchos lenguajes es el do-while-do.

RECONOCIMIENTO DE PATRONES. Parte importante de varios lenguajes de programación. PERL, ML y SNOBOL4 tienen herramientas muy poderosas de reconocimiento de patrones (pattern matching), y por eso trabajan con gran eficacia y facilidad con cadenas de texto. En PROLOG, el reconocimiento de patrones es esencial para la sustitución que en su vez es la técnica fundamental de la unificación.

UNIFICACIÓN. Es la operación básica para la resolución de programas en programas directivos. La substitución es una parte integral para la unificación, se trata de buscar la sustitución que puede unificar (hacer iguales) dos términos (si es posible). Una manera de tratar de unificar dos términos es dibujarlos como árboles, y buscar los pares de desacuerdo uno por uno.

CAPÍTULO 8:

“CONTROL EN SUBPROGRAMAS O RUTINAS”.

Los programas se componen de Rutinas o Subprogramas que son unidades pequeñas. La rutina es todo aquello que cumpla con los principios que son comunes a todos los Lenguajes de Programación, se puede ver de dos formas, procedimientos (procedures) y funciones (functions). Las funciones regresan un valor al término de su ejecución y los procedimientos son una función que regresa el valor vacío (null en muchos lenguajes). Las rutinas deben tener un nombre, alcance (scope), tipo de valor de regreso, y un par de valores. El nombre de la rutina debe introducirse mediante una Declaración de Rutina. El alcance del nombre es desde el punto de la declaración hasta el final de un bloque que en ocasiones es determinado estáticamente y en otras dinámicamente, dependiendo del lenguaje. Las rutinas se activan mediante una Invocación de Rutina (o Llamado de Rutina) que además debe pasar a esta los parámetros necesarios para su ejecución y debe de estar dentro del alcance de ésta. El alcance de una rutina se divide en el alcance local que incluye las declaraciones hechas dentro de ésta, que sólo pueden ser accesadas dentro de la misma rutina y el alcance no-local que incluye los datos y rutinas que están a su alcance, pero que, sin embargo, han sido declaradas independientemente de esta. Las variables y rutinas que son accesibles para todas las rutinas de un programa son Elementos Globales. El encabezado de una rutina define que tipo de rutina es y contiene su nombre, los parámetros de entrada y el valor de salida (si hay). El tipo de una rutina puede ser definido precisamente por el concepto de firma. Un Llamado de Rutina es de tipo correcto si es hecho conforme al tipo de rutina. Algunos lenguajes son capaces de tener apuntadores hacia rutinas, es decir, tratar a las rutinas como si fueran variables, en cierto modo. Otros como Ada y Pascal hacen una distinción entre la declaración y la definición de una rutina. Esto se hace principalmente para poder hacer llamados recursivos entre dichas rutinas.

La representación de una rutina durante la ejecución es llamada Instancia de Rutina y está compuesta por un bloque de código y un registro de activación. El Registro de Activación contiene toda la información necesaria para ejecutar la rutina como los objetos asociados con las variables locales, el apuntador de

regreso, que es el lugar al que deberá regresar la ejecución del programa una vez que se termine de ejecutar la rutina.

La Referencia Ambiental de una Instancia de Rutina consiste de todas las variables (locales y no locales) que están relacionadas con objetos que se encuentran dentro del registro de activación. Y se puede ver como ambiente local y no local. La modificación de variables no locales a través de una rutina se llama Efecto Lateral. Si las rutinas se pueden activar recursivamente, es decir, una unidad se activa ya estando activa, se necesita un registro de activación para cada llamado que se haga a la rutina, pues el código que se ejecuta es siempre el mismo y podrían producirse errores debido a ambigüedades o valores incorrectos. Además, la “liga” que se hace entre el registro de activación y el Cuerpo de la Rutina se debe hacer dinámicamente. Es obvio que se debe hacer una distinción entre los Parámetros Formales y Parámetros Reales de una Rutina, los primeros aparecen en la definición de la rutina y los segundos aparecen en el llamado de la rutina. La mayoría de los Lenguajes de Programación ubican a los Parámetros Reales por la posición en la que aparecen, sin embargo, no es siempre necesario que aparezcan todos los parámetros.

RUTINAS GENÉRICAS. Al crear rutinas se busca “factorizar” una porción de código que es común a varias partes de un programa en la que sólo varían los valores de las variables relacionadas con el proceso a realizar. Por lo que se le da un nombre a ese código y se pone en un lugar “especial”. Sin embargo, no siempre se puede “factorizar” este código a pesar de que la tarea sea en esencia la misma, ya que pueden diferir en el tipo de parámetros. Algunos Lenguajes de Programación ofrecen una solución a este problema por medio de las Rutinas Genéricas que son como machotes desde los cuales se pueden generar varias rutinas específicas.

Alias y Sobrecarga: a veces se necesita usar dos rutinas que hacen lo mismo pero difieren en los parámetros o tipo de parámetros, es natural querer que estas dos

rutinas lleven el mismo nombre, pero con la herramienta hasta ahora revisada eso no es posible. Aún cuando no haya ambigüedad (para nosotros) que rutina estaríamos invocando. Por eso se creó la Sobrecarga, es cuando un nombre se dice que está Sobrecargado si se refiere a más de una entidad en ciertos puntos de un programa, pero la ocurrencia específica del nombre provee suficiente información para permitir que la liga entre el código y la activación esté únicamente determinada.

ESTRUCTURA DE TIEMPO DE EJECUCIÓN.

-Lenguajes Estáticos (1eras versiones de FORTRAN y COBOL), se aseguran de determinar los requerimientos de memoria de un programa antes de ser ejecutados. No pueden computar funciones recursivas porque con estas no es posible determinar la memoria necesaria para la ejecución de un programa.

-Lenguajes basados en Pilas (ALGOL 60), son más demandantes en términos de requerimientos de memoria, pues no puede ser determinada en tiempos de compilación. Aunque es posible saber como se va a usar la memoria con una estrategia LIFO (last in, first out). Es decir, el último registro de activación que fue almacenado es el próximo que será borrado.

-En los Lenguajes Dinámicos no hay forma de saber como se va a usar la memoria, los datos se almacenan conforme se van necesitando en tiempo de ejecución. El problema es administrar la memoria eficientemente, por eso se introduce el término “heap” que hace referenci a a la memoria de datos.

-Los Lenguajes con un solo Enunciado Simple tienen tipos de valores e instrucciones solamente y no tiene funciones. Sólo maneja datos que pueden conocerse estáticamente. Los programas consisten de una rutina principal (main()), que contiene la declaración de los datos que se usarán y las instrucciones que manipularán esos datos. Solo existe un registro de activación

para la rutina principal, en él, hay espacios para cada una de las variables que aparecen en el programa.

-Los Lenguajes con Rutinas Simples sirven para agregar una nueva característica que permita definir rutinas en un programa, y a las rutinas les permiten definir sus datos locales. Un programa en este tipo de lenguajes consistirá de lo siguiente: Un conjunto de declaraciones (de datos globales) posiblemente vacío, un conjunto de declaraciones o definiciones de rutinas posiblemente vacío y una rutina principal (main()), que contiene declaraciones de datos locales e instrucciones que se activarán automáticamente cuando el programa se ejecuta. La rutina principal no puede ser llamada por las otras rutinas.

Las rutinas pueden acceder sus datos locales y los globales no redefinidos dentro de la rutina. Las rutinas no pueden estar anidadas, es decir no se pueden llamar a sí mismas, no tienen parámetros y no regresa ningún tipo de valor. El tamaño de los registros de activación puede conocerse en tiempo de compilación y todos los registros de activación se les pueden asignar memoria antes de ejecutar el programa. Lo mismo puede hacerse con las variables. Este tipo de manejo de memoria tiene la ventaja de que nunca pueden producirse insuficiencias de memoria en tiempo de ejecución, sin embargo, puede desperdiciar memoria pues se reserva espacio en memoria para las subrutinas incluso si estas no se ejecutan.

FUNCIONES RECURSIVAS. La recursión tiene que ver con la habilidad que tienen los subprogramas para llamarse a sí mismos (recursión directa) o llamar a otro subprograma de manera recursiva (recursión indirecta) y la capacidad de regresar valores, es decir, que se comporten como funciones.
Para cada invocación de una rutina se crea un nuevo registro de activación en el tiempo de ejecución, aunque el tamaño del registro de activación se conoce, no se sabe el número de veces que tal rutina será invocada y cuantas instancias de su registro de activación se necesiten. Todas las diferentes instancias del registro de activación de un subprograma tienen el mismo código, así que el segmento de

código no cambia de una activación a otra, pero se necesitan distintos registros de activación para colocar los diferentes valores del ambiente local.

Un registro de activación es creado por cada función invocadora en cada invocación, y cada creación establece un nuevo enlace con el segmento de código correspondiente para crear una nueva activación de la función invocada. Para enlazar las variables con su dirección en el segmento de datos, se usa un espacio en el segmento de datos del programa que contiene la dirección base del registro de activación, que actualmente se está ejecutando (valor CURRENT). Cuando la instancia del subprograma actual termina, su registro de activación no se necesita más, por lo que se debe liberar el espacio que ocupaba este registro de activación dejando así espacio disponible para las futuras invocaciones. Una vez descargado un registro de activación es necesario recuperar el registro anterior, es decir, el de la rutina invocadora. Como el registro de activación que se libera es el más recientemente creado, los registros de activación se pueden colocar en una estructura con una política de que el último en entrar es el primero en salir, es decir, en una pila o stack. Para hacer posible el regreso de una invocación es necesario conocer el punto de regreso, es decir, la siguiente instrucción a ejecutar en el segmento de código de la función invocadora y la dirección base del registro de activación de la misma. Para ello en el registro de activación se reserva un espacio para el punto de regreso y otro para la dirección base del registro de activación de la rutina invocadora, este último es llamado la Liga Dinámica (Dynamic Link). La cadena de ligas dinámicas originada en el registro de activación actual es llamada la Cadena Dinámica (Dynamic Chain), que representa la secuencia de registro de activación en tiempo de ejecución.

Para colocar un nuevo registro de activación es necesario conocer la dirección de la primera celda libre en el segmento de datos, para ello se puede reservar un espacio en el segmento de datos del programa o bien, usar el registro (SP. Es necesario proveer un espacio de memoria para el valor de regreso del subprograma, si es que este se requiere. Un registro de activación es destruido

después del regreso, así que el valor del regreso se tiene que guardar en el registro de activación de la rutina invocadora. Cuando una función es invocada, el registro de activación de la rutina invocadora es extendido para guardar el valor de regreso y la rutina invocada escribe el valor en tal espacio (usando un desplazamiento negativo).

La Estructura de Bloques es para controlar el alcance de las variables, definir su tiempo de vida y dividir un programa en unidades más pequeñas. Dos bloques en un programa pueden ser ajenos (sin porción en común) o anidados (uno contiene completamente al otro). Existen dos tipos, en el primero se permite que aparezcan declaraciones locales en un enunciado compuesto, en el segundo se permite anidar definiciones de rutinas, conjuntamente a estos dos tipos se les llama Estructura de Bloque.

Anidamiento por Enunciados Compuestos: Define el alcance de sus variables localmente declaradas, éstas son visibles en el enunciado compuesto, incluyendo los enunciados compuestos inmersos en él, con el mismo nombre si no han sido redeclaradas. Una declaración interna enmascara una declaración externa con el mismo nombre. Un enunciado compuesto también define el tiempo de vida de un dato localmente declarado. Un espacio en memoria es enlazado a una variable cuando el bloque en el que está declarada entra en ejecución, el enlace es removido cuando el bloque sale de ejecución.

Una estructura de bloque puede ser descrita por un Árbol Estático de Anidamiento (SNT), el cual muestra cómo están anidados los bloques. Cada nodo del SNT describe un bloque, los descendientes de un nodo representan los bloques inmediatamente anidados en él. Para implementar la Estructura de Bloques en el Registro de Activación se asigna espacio suficiente para todas las variables, tomando en cuenta que dos bloques ajenos no pueden estar activos al mismo tiempo, así que es posible utilizar el mismo espacio en memoria para variables de

bloques ajenos, a esto se le llama Sobreponer (Overlay) y se puede hacer en tiempo de compilación.

Anidamiento por Rutinas Localmente Declaradas: Una rutina puede ser declarada dentro de otra rutina, de manera parecida a las variables, las rutinas localmente declaradas en el cuerpo de una rutina son visibles (y por ello pueden ser ejecutadas) dentro de su cuerpo y en los cuerpos de las rutinas inmersas en él, una declaración interna enmascara a una externa con la misma firma. En una rutina también es posible acceder a las variables locales y a las no locales declaradas en las rutinas que la encierran. Al igual que para los enunciados compuestos, también se puede definir el árbol de anidamiento estático para subrutinas. Cada bloque es un enunciado compuesto o bien, el cuerpo de una rutina.

En el caso del manejo de funciones recursivas cada registro de activación las variables locales son enlazadas en tiempo de ejecución al registro de activación actual y las variables globales se enlazan fácilmente. En el caso de rutinas anidadas no todas las variables son locales, así que es necesario enlazar las variables no locales al registro de activación de la rutina envolvente. La liga dinámica no sirve para este propósito ya que es posible que una rutina interna llame a una externa, de esta manera no se cumplirían las reglas de alcance basadas en un anidamiento estático (que son las que se quieren).

Una forma de acceder a las variables no locales posibles es que para cada registro de activación se tenga un apuntador al registro de activación de la unidad que estáticamente la encierre en el texto del programa, a este apuntador se le llama Liga Estática (Static Link). La secuencia de ligas estáticas a partir del registro de activación actual es llamada Cadena Estática **(**Static Chain). Las referencias a variables no locales se pueden realizar por medio de una búsqueda a través de la cadena estática, la cual, es seguida hasta encontrar el enlace correcto. En este caso el enlace a variables locales se puede tratar de la misma

manera que cualquier otro ambiente no local o bien, como un caso especial, utilizando las direcciones absolutas.

Para Invocar un Subprograma se siguen los siguientes pasos: asignar espacio en el Stack para el valor de regreso, colocar el valor de regreso, colocar la liga dinámica, colocar la liga estática, colocar CURRENT a la dirección base del actual registro de activación, colocar el registro SP o FREE a CURRENT más el tamaño del registro de activación y saltar al código del subprograma invocado.

COMPORTAMIENTOS MÁS DINÁMICOS.

Registros de Activación cuyo Tamaño se conoce en la Unidad de Activación: se supone que el tamaño de todas las variables es conocido en tiempo de compilación. Como los Arreglos Dinámicos, cuyas cotas se conocen en tiempo de ejecución, cuando la unidad (subprograma o enunciado compuesto) en el cual el arreglo está declarado es activada. En tiempo de compilación, se puede reservar un espacio en memoria para los descriptores de los arreglos dinámicos. El descriptor incluye una celda para cada una de las cotas superiores e inferiores de cada dimensión, suponiendo que la dimensión del arreglo es conocida estáticamente, de esta manera el tamaño de descriptor se conoce en tiempo de compilación. Todos los accesos al arreglo dinámico son trasladados a referencias indirectas a través de un apuntador en el descriptor, cuyo desplazamiento es determinado estáticamente. En tiempo de ejecución el registro de activación sigue varios pasos para su posicionamiento en memoria. Primero se colocan los datos cuyo tamaño es conocido estáticamente y los descriptores de los arreglos dinámicos. Segundo, cuando se encuentra la declaración de un arreglo dinámico, se colocan las cotas de las dimensiones, el tamaño del arreglo es evaluado y el registro de activación se extiende para incluir espacio para la variable, esto es posible ya que el registro de activación actual se encuentra en el tope del stack. Tercero, el apuntador en el descriptor se coloca una referencia al área que acaba

de ser reservada. Los últimos dos pasos son implementados por código que es generado para cada declaración de un Arreglo Dinámico.

Alojamiento completamente Dinámico de Memoria: se asume que todas las variables son colocadas en memoria automáticamente cuando el alcance, en el cual ellas son declaradas, es activado en tiempo de ejecución, y se remueven de memoria cuando este termina. Otra variación en la cual los datos se pueden colocar en la memoria de manera explícita a través de instrucciones explícitas para su posicionamiento en memoria es la definición de Apuntadores y la existencia de Enunciados de Asignación que colocan tales datos de manera completamente dinámica. De acuerdo a este esquema, los datos son posicionados explícitamente cuando se necesitan. Estos datos no se pueden colocar en el stack, como se hacía con el posicionamiento automático, ya que la semántica de este tipo de datos dice que su tiempo de vida no depende de la unidad en la cual el enunciado de asignación aparece, sino que dura tanto como pueda ser accesible. Una implementación de este concepto consiste en colocar los datos dinámicos en el segmento de datos empezando por la dirección más alta, o bien tener otra área adicional al segmento de datos. Esta área es llamada heap. Los nuevos datos son colocados en el heap cuando las instrucciones de asignación se ejecutan, estas instrucciones enlazan la dirección donde se colocaron estos datos a una variable que se encuentra en el registro de activación de la unidad donde se llamó a la instrucción.

LENGUAJES DINÁMICOS. Se refieren a los lenguajes que adoptan reglas dinámicas en lugar de estáticas.

Tipos Dinámicos: El tipo de variable y los métodos de acceso, y de operaciones permitidas no pueden ser determinados en un tiempo de compilación. Para variables de tipo dinámico, se almacena en el registro de activación el tipo de la variable. Si el tipo de la variable puede cambiar entonces pueden hacerlo también el tamaño y los contenidos de su descriptor. Ya que no sólo el tipo de la variable

cambia, sino también el tamaño de su descriptor, los descriptores deben ser guardados en el heap. Cada variable necesita un apuntador en el registro de activación que apunte hacia el descriptor de la variable en el heap, el cual a su vez puede contener un apuntador hacia el objeto mismo.

Alcance Dinámico: El alcance de un nombre es dependiente de la cadena de llamadas en tiempo de ejecución (cadena dinámica) en lugar de serlo de la estructura estática del programa. El problema se refiere a las variables no locales. La regla del alcance dinámico define el alcance dinámico de cada asociación en términos del flujo dinámico de la ejecución del programa. Ejemplos de lenguajes que utilizan alcance dinámico son APL, Snobol, dialectos iniciales de LISP. Dos modos en los que las referencias no locales pueden ser implementadas en los lenguajes con alcance dinámico son:

-El Acceso Profundo, la referencia puede ser resuelta buscando a través de las declaraciones en otros subprogramas, cuya actividad está en curso, empezando con el que más recientemente ha sido activado (la cadena dinámica es seguida). No hay manera de determinar durante compilación la longitud a recorrer en la cadena dinámica. Cada instancia de registro de activación debe ser recorrida hasta se encuentre la primera instancia de la variable. Debido a esto, los lenguajes de alcance dinámico tienen por lo general una velocidad baja de ejecución. Los registros de activación deben almacenar los nombres de las variables para el proceso de búsqueda, mientras que en las implementaciones de lenguajes con alcance estático sólo los valores son requeridos (los nombres no se requieren para el alcance estático porque todas las variables están representadas por los pares <dirección base, desplazamiento local>).

-El Acceso Superficial es un método alternativo de implementación, no una semántica alternativa. Las variables en este método, declaradas en subprogramas no son almacenadas en los registros de activación de aquellos subprogramas, porque con los enfoques dinámicos sólo hay una versión visible de la variable en

un momento dado. Una forma de implementar el acceso superficial es tener un stack separado para cada nombre de variable en todo el programa. A cada momento que una nueva variable es creada por una declaración en la activación del subprograma, se da a la variable una celda en su respectivo stack de acuerdo a su nombre. Cada referencia al nombre es para la variable en el tope del stack, porque es la más recientemente creada. Cuando el subprograma termina, el tiempo de vida de la variable local termina, así que es retirada del stack. Este método permite muy rápidas referencias a las variables, pero el mantenimiento del stack en las entradas y en las salidas de los subprogramas es costoso.

Otra Forma: Una llamada de subprograma requiere que todas sus variables locales sean colocadas lógicamente en la Tabla Central (tabla central de ambiente de referencia). En conclusión, esta tabla central común a todos los subprogramas contiene en todo momento de ejecución del programa, todas las asociaciones a identificación activas en curso sin importar si son locales o no locales. Si la serie de identificadores referenciados en cualquier subprograma puede ser determinada en compilación, entonces la tabla central es inicializada para contener una entrada para cada identificador sin importar el número de diferentes subprogramas en los que aparezca el identificador. Cada entrada en la tabla contiene también una Bandera de Activación que indica si un identificador en particular tiene una asociación activa, así como el espacio para un apuntador al objeto de la asociación.

En el alcance dinámico, los atributos correctos para variables no locales visibles para una declaración en el programa no pueden ser determinados estáticamente. Tales variables no son siempre las mismas. Una declaración en el subprograma que contiene una referencia a una variable no local puede referirse a diferentes variables cada vez que la declaración es ejecutada. Esto lleva a los siguientes problemas:

-Durante la ejecución del subprograma, sus variables locales son visibles para cualquier otro subprograma en ejecución, sin importar su proximidad textual. No hay modo de proteger las variables locales de su accesibilidad. Los subprogramas son siempre ejecutados en el ambiente inmediato al invocador, por lo que el alcance dinámico resulta en programas menos confiables que con un alcance estático.

-El alcance dinámico no puede verificar estáticamente el tipo de las variables no locales, como consecuencia de la incapacidad de determinar estáticamente la declaración para una variable referenciada como no local. El alcance dinámico hace también que los programas sean mucho más difíciles de leer, porque la secuencia de llamada de los subprogramas debe ser conocida para determinar el significado de las referencias para las variables no locales.

Una ventaja es que con el alcance dinámico los programas heredan los contextos de los invocadores, esto es, cualquier variable declarada en el invocador que no es nuevamente declarada en el subprograma es visible en el subprograma invocado. Este puede ser el método conveniente de comunicación entre las unidades del programa, aunque es menos segura que otros métodos, tal como la transmisión de parámetros.

Paso de Parámetros: hay 2 formas en las que un subprograma puede tener acceso a los datos que serán procesados. Son mediante acceso directo de variables no locales (declaradas en otro lugar, pero visibles en el subprograma) y mediante transmisión de parámetros. Los datos transmitidos mediante parámetros son ingresados mediante nombres que son locales para el subprograma.

Datos como Parámetros: Los parámetros formales están caracterizados por uno de los siguientes modelos semánticos. Pueden recibir datos del parámetro real correspondiente: in-mode*;* Pueden transmitir datos hacia el parámetro real: out-mode ó pueden hacer ambos: inout-mode*.* Estos modelos describen el

comportamiento del parámetro. Existen dos modelos conceptuales de cómo se efectúan las transferencias de datos en las transmisiones: ya sea que un valor real sea desplazado físicamente (hacia el invocado, hacia el invocador, o ambos), o con una transferencia de la transmisión de acceso (usualmente un apuntador).

Paso de Parámetros por Copia: se puede dividir a su vez en tres modos de acuerdo a como la variables locales correspondientes a variables locales son inicializadas y la forma en que estas afectan a los parámetros reales.

Paso por Valor: El valor del parámetro real es usado para inicializar el parámetro formal correspondiente, el que actúa entonces como una variable local en el subprograma, proveyendo así una semántica in-mode. Normalmente es implementado mediante una transmisión real de datos, porque los accesos son en general más eficientes con este método. Puede ser implementado como la transmisión de un camino de acceso hacia el valor del parámetro real en el invocador, pero puede requerirse que el valor exista en una celda de escritura protegida, lo cual no es simple de hacer. Si el desplazamiento físico es hecho, es porque el almacenamiento adicional es requerido por el parámetro formal, ya sea en el subprograma invocado o bien en un área fuera del subprograma invocador y del invocado. De manera adicional, el parámetro real debe ser desplazado físicamente hacia el área de almacenamiento del parámetro formal correspondiente. Las operaciones de almacenamiento y de desplazamiento pueden ser costosas si la talla del parámetro es grande, como en el caso de un arreglo largo.

Paso por Resultado: Es un modelo de implementación para parámetros out-mode. Cuando el parámetro es transmitido por resultado, ningún valor es transmitido hacia el subprograma invocado. El parámetro formal correspondiente actúa como una variable local, pero justo antes de que el control sea transferido de regreso hacia el invocador su valor es transmitido de vuelta hacia el parámetro real del invocador, el cuál es una variable. Cuando los valores son devueltos, también se

requiere de un almacenamiento suplementario y de operaciones de copia. El problema con la transmisión de un camino de acceso con este método es que debemos asegurar que el valor inicial del parámetro real no sea usado en un subprograma invocado.

Paso por Valor - Resultado: Es un modelo de implementación para parámetros inout en el que los valores son desplazados. El valor del parámetro real es usado para inicializar el parámetro formal correspondiente, que actúa entonces como una variable local. Los parámetros formales deben tener almacenamiento local asociado con el subprograma invocado. Al término del subprograma, el valor del parámetro formal es transmitido de vuelta hacia el parámetro real. Se requiere de almacenamientos múltiples para parámetros y de tiempo para copiar valores. El orden de asignación de los parámetros puede ser una causa de problemas.

Paso por Referencia: Es otro modelo de implementación para parámetros de modo inout. Transmite un camino de acceso, usualmente sólo una dirección al parámetro real. Con ello, el parámetro real es compartido con el subprograma invocador. Es eficiente en términos de espacio y tiempo, ya que no se requiere duplicar el espacio ni efectúa copia alguna. El acceso a los parámetros formales será más lento pues se requiere de nivel más de direccionamiento indirecto que cuando los valores de los datos son transmitidos. Pueden además ocurrir cambios involuntarios en los parámetros reales. Por último, pueden crearse alias, ya que los caminos de acceso a los subprogramas abiertos están abiertos. Solo las direcciones deben ser colocadas en la pila. Si se trata de constantes, la dirección de la constante es transmitida. En caso de que sea una expresión, el compilador construye el código para evaluar la expresión antes de que la transmisión se realice y entonces coloca la dirección del resultado en la pila, en algunos lenguajes solo se permite el paso por referencia cuando los parámetros reales son variables.

Paso por Nombre: Es un método de transmisión de parámetros de modo inout que no corresponde a un modelo de implementación sencillo. El parámetro formal es substituido textualmente por el parámetro real correspondiente en toda su ocurrencia en el subprograma. La conexión real con un valor o una dirección es retrasada hasta que el parámetro formal es asignado o referenciado, con lo que se logra mayor flexibilidad. La forma del parámetro real dicta el modelo de implementación de los parámetros de la Llamada por nombre, lo que constituye una gran diferencia. Si el parámetro real es una variable escalar, entonces este método es equivalente a la Llamada por referencia; si es una constante entonces es equivalente a la Llamada por valor. Si es un elemento en arreglo o una expresión, entonces no es ninguno de ellos porque pueden cambiar en el tiempo de ejecución (el valor de la expresión del índice puede cambiar durante la ejecución entre los tiempos de varias referencias.) Si el parámetro real es una expresión que contiene referencias a una variable, la Llamada por nombre es de nuevo diferente a cualquier otra: la expresión es evaluada para cada referencia al parámetro formal al momento de que la referencia es buscada. Si cualquiera de las variables en la expresión es accesible y es cambiada por el subprograma, el valor de la expresión puede cambiar con cada referencia al parámetro formal. Este método es sumamente flexible, pero el proceso es lento. El concepto de enlace tardío en el que está basada la Llamada por nombre es usado en polimorfismo dinámico.

La Evaluación Perezosa es otro mecanismo útil que es una forma de enlace retardado: El proceso de evaluación de las partes del código funcional sólo cuando resulta cierto cuando la evaluación es necesaria. Normalmente es implementado por medio de un proceso sin parámetros o segmento de código, llamado thunk. Un thunk debe ser invocado por cada referencia hacia un parámetro pasado por nombre en el subprograma invocado. El thunk evaluará la referencia en el ambiente de referencia adecuado, el cual es el que transmite el parámetro real. Los thunks están ligados con sus medios de referencia en el momento de la invocación que transmitió el parámetro pasado por nombre. El thunk devuelve la

dirección del parámetro real. Si el la referencia al parámetro está en una expresión, el código de referencia debe incluir la dereferencia necesaria para obtener el valor desde la celda cuya dirección fue devuelta por el thunk.

RUTINAS COMO PARÁMETROS. En los lenguajes que soportan variables de tipo rutina se dice que tratan a las rutinas como objetos de primera clase. En éstos, las rutinas pueden ser pasadas como parámetros. La descripción de los parámetros del subprograma debe ser enviada junto con el nombre del subprograma, asumiendo que los tipos de los parámetros reales serán checados en el tiempo de ejecución contra el tipo de los parámetros formales. Para establecer el ambiente de referencia correcto para ejecutar subprogramas pasados como parámetros tenemos tres opciones:

-El ambiente del subprograma que invoca al subprograma pasado.

-El ambiente del bloque o subprograma, en el que el subprograma pasado está declarado.

-El ambiente del bloque o subprograma, que incluye la instrucción de llamada que trasmite al subprograma como un parámetro real.

La primera elección es llamada Enlace Superficial (shallow binding*)* y la siguiente elección es llamada Enlace Profundo (deep binding*)*. La última elección nunca ha sido usada porque puede ser asumido como un ambiente en el que el subprograma aparece como un parámetro que no tiene conexión natural con el subprograma de transmisión. El enlace profundo se utiliza en los lenguajes estáticos estructurados por bloques, está implementado para crear una representación explícita de un ambiente referenciado (generalmente aquel en el que la subrutina se ejecutaría si esta fuera invocada directamente, es decir, no como un parámetro).

CAPÍTULO 9:

“ADMINISTRACIÓN DE ALMACENAMIENTO”.

Manejar la memoria es interesante e importante para la programación. Cada diseño de lenguaje tiene características o restricciones que permiten el uso de ciertas técnicas de manejo de memoria. Los detalles del mecanismo, su representación en hardware y software son tareas del implementador. Muchos programas y sus elementos requieren almacenamiento durante la ejecución del programa.

PRINCIPALES ELEMENTOS QUE REQUIEREN ESPACIO EN TIEMPO DE EJECUCIÓN.

-Segmentos de Código para los Programas de Usuario: muchos bloques de almacenamiento son reservados para guardar los segmentos de código que representan los programas de usuario.

-Programas del Sistema en Tiempo de Ejecución: otra parte de los bloques de almacenamiento durante la ejecución es reservada para los programas de sistema que soporten la ejecución de programas del usuario. Estas pueden ser desde simples rutinas de librerías.

-Estructuras de Datos y Constantes definidas por el Usuario: debe de existir espacio disponible para las estructuras que fueron creadas o declaradas por el usuario, incluyendo las constantes.

-Puntos de Regreso de Subprogramas: como tienen la propiedad de ser invocados desde cualquier parte del programa, debe reservarse memoria para el control de secuencia generada internamente, como los puntos de retornos de retorno de subprogramas.

-Entornos de Referencia: Asociaciones de identificadores que pueden requerir de una gran cantidad de espacio, por ejemplo una lista A en LISP.

-Temporales en Evaluación de Expresiones: se requiere un espacio de almacenamiento temporal para los resultados intermedios de una evaluación.

-Temporales en el Paso de Parámetros: al llamar un subprograma, una lista de los parámetros en turno debe de ser evaluada y los resultados almacenados temporalmente hasta que sea completada la evaluación de la lista. Cuando la evaluación de algún parámetro de la lista requiera llamadas a funciones recursivas, existe la posibilidad de que se emplee un número ilimitado de almacenamientos temporales sucesivos.

-Buffers de Entrada y Salida: sirven como áreas de almacenamiento temporal donde los datos son almacenados durante el tiempo de transferencia física hacia el almacenamiento externo. Casi siempre son reservadas para los buffers cientos de localidades.

-Datos Misceláneos del Sistema: almacenamiento para diversos datos propios del sistema, como tablas, estados de información para entrada y salida, para la recolección de basura y contadores de referencia.

PRINCIPALES OPERACIONES QUE REQUIEREN ALMACENAMIENTO EN MEMORIA.

-Llamadas a Subprogramas y Retorno de Operaciones: ambientes de referencia, llamadas a subprogramas y datos usados en subprogramas son operaciones que requieren almacenamiento. La ejecución del retorno desde un subprograma requiere la liberación del espacio de almacenamiento utilizado durante la ejecución.

-Creación y Destrucción de Operaciones de Estructuras de Datos: Si el lenguaje posee operaciones que permiten la creación de nuevas estructuras de datos en puntos arbitrarios durante la ejecución de un programa (no solamente cuando

inicia algún subprograma), entonces estas operaciones normalmente requieren disponer de espacio aparte del utilizado cuando inicia el subprograma.

-Inserción y Borrado de Operaciones de Componentes: Si el lenguaje proporciona operaciones de inserción y borrado de estructura de datos, para que estas operaciones puedan ser implementadas, se podría requerir de operaciones de almacenamiento y liberación del espacio.

ALMACENAMIENTO CONTROLADO POR EL PROGRAMADOR Y POR EL SISTEMA.

Fases del Manejo de Almacenamiento. Hay 3 aspectos básicos del almacenamiento: la Asignación Inicial donde al principio de la ejecución cada pieza de almacenamiento debe ser reservada para usarse o estar disponible. Si está libre inicialmente, debe de poder ser reservada dinámicamente durante la ejecución del programa. Cualquier sistema de almacenamiento requiere de alguna técnica para mantener un registro del espacio libre, así como mecanismos para su uso, dependiendo de las necesidades durante la ejecución; la Recuperación donde el almacenamiento que ha sido asignado y usado subsecuentemente debería de ser recuperado por el administrador de almacenamiento para su nuevo uso. El proceso de recuperación puede ser simple, como el reposicionamiento del apuntador de stack, o complejo, como la recuperación de Basura; y la Compactación y Nuevo Uso donde el espacio de almacenamiento recuperado podría estar inmediatamente listo para su nuevo uso. Otras veces podría ser necesaria la compactación, que es la construcción de grandes bloques libres a partir de piezas pequeñas de almacenamiento. El nuevo uso normalmente involucra las mismas técnicas que la asignación inicial.

Manejo de Almacenamiento Estático: forma más sencilla de asignación. El espacio reservado durante la traducción del programa permanece fijo durante la ejecución. Normalmente el almacenamiento para los segmentos de código del usuario y del

sistema es reservado estáticamente. También los buffers de I/O y datos misceláneos del sistema. La reservación estática no requiere de administración de almacenamiento en tiempo de ejecución, y por supuesto, tampoco recuperación y nuevo uso.

Manejo de Almacenamiento basado en Stacks: forma más simple de administración de almacenamiento en tiempo de ejecución. El espacio libre al inicio de la ejecución consiste de un bloque secuencial en la memoria. Conforme nuevo espacio es solicitado, este es tomado de locaciones sucesivas de este bloque, comenzando por algún extremo. El espacio debe de ser liberado en orden inverso, para que el bloque de espacio liberado quede siempre en la parte alta del snack. En este caso, todo lo que se necesita para el control del almacenamiento es un simple apuntador de Stack (stack pointer). Este siempre apunta a la parte alta del stack, donde se localiza la siguiente palabra libre de almacenamiento. Todo el espacio usado yace debajo de la dirección señalada por el stack pointer.

Administración de Almacenamiento por Heap - Elementos de Tamaño Fijo. Un Heap es un bloque de almacenamiento con piezas que son asignadas y liberadas de una manera relativamente estructurada. Los problemas de reservación de espacio, recuperación, compactación y nuevo uso son difíciles de resolver. La necesidad de utilizar un Heap surge cuando un lenguaje permite almacenamiento para asignación y liberación en puntos arbitrarios durante la ejecución de un programa.

Recuperación - Contadores de Referencia y Recolección de Basura. La forma más sencilla de recuperación es la del Retorno Explícito. Cuando un elemento que estaba en uso está disponible para nuevo uso, este debe ser explícitamente identificado como Free y retornado a la Lista de Espacio Libre. Cuando los elementos son utilizados por el sistema, cada rutina del sistema es responsable de liberar el espacio. El retorno explícito es la técnica natural para administrar el almacenamiento en el heap, pero no siempre es posible emplearla. Hay dos

problemas que la inutilizan: si una estructura se destruye antes de que todos los accesos a la estructura hayan sido destruidos, cualquier trayectoria restante se convierte en una Referencia Dangling (Dangling References). Y si el último acceso a la estructura es destruido sin que la estructura misma sea destruida, entonces la estructura se convierte en Garbage.

Existen otras alternativas para la solución de estos problemas en el retorno explicito, una es la de los reference counts y otra la de los garbage colectors.

Reference Counts: El uso de un contador de referencia genera un requerimiento mayor de espacio. El contador de referencia indica cuantos apuntadores a un elemento existen. Cuando un elemento es colocado en el espacio libre, su contador de referencia se inicializa en 1. Cuando un se crea un nuevo apuntador al elemento, el contador se incrementa en uno. Cuando éste apuntador es destruido, el contador de referencia se decrementa en uno. Cuando el contador de referencia de un elemento llega a cero, el elemento es libre y regresa a la lista de espacio libre. Los contadores de referencia permiten evitar las referencias dangling y a generación de garbage en la mayoría de los casos.

Recolección de Basura: Cuando el espacio libre es ocupado totalmente y es necesario más espacio para almacenar, el cómputo se suspende temporalmente y un proceso de recolección de basura tiene lugar. Este identifica los elementos inservibles en la estructura y los regresa al espacio libre. Una vez liberado el espacio, el cómputo se reanuda y el proceso de recolección de basura no se vuelve a activar hasta que el espacio libre se haya agotado. El proceso de recolección de basura se lleva a cabo en dos fases: Marcado que es el proceso donde cada elemento activo dentro de la estructura es marcado. Esta marca pone el bit del colector de basura en apagado; y Barrido donde mediante un proceso secuencial se examinan todos los elementos de la estructura, siendo mandados al espacio libre todos aquellos que posean encendido el bit de recolección de

basura, al mismo tiempo que se lleva a cabo el barrido todos los elementos, sus bits de barrido son reestablecidos.

Administración de Almacenamiento por Heap - Elementos de tamaño variable. El manejo de estructuras de almacenamiento es complejo cuando los elementos que contienen están conformados por estructuras de datos creadas por el usuario o registros de activación. Este tipo de datos impiden que la memoria se maneje de una manera ordinaria y se deben aplicar técnicas para recuperar el espacio libre dentro de las páginas de memoria para reutilizar este espacio.

Almacenamiento Inicial y Nuevo Uso: Cuando los elementos guardados en la estructura de almacenamiento tienen un tamaño fijo, el uso del espacio libre y el espacio ocupado se realiza con simples sumas a partir del apuntador inicial y el tamaño de la página. En el caso de elementos de tamaño variable la localización del espacio libre mediante ésta técnica no es posible. El objetivo es formar bloques libres lo más grandes que sea posible. Para recuperar el espacio libre en los casos, se pueden emplear 2 métodos de nuevo uso de espacio:

-Uso de Espacio Libre, buscando en la lista de localidades un bloque de tamaño adecuado y devolviendo a la lista cualquier espacio sobrante después de la reservación.

-Compactando el Espacio Libre moviendo todos los elementos activos al principio de la estructura de almacenamiento y dejando el espacio libre al final del stack.

Nuevo Uso Directo mediante la Lista de Espacio Libre. En un método normal se alojarían las N palabras y el resto sería enviado al espacio libre, sin embargo cuando se hace uso de una lista de espacio libre, hay 2 formas para llevarla a cabo:

-Método del Primero Encontrado: cuando un espacio de N palabras es necesitado, la lista de espacio libre es revisada hasta que un bloque de N palabras o más palabras es encontrado, en ese momento se aloja la información y el resto del espacio es regresado a la lista de espacio libre.

-Método del Mejor Encontrado: cuando un espacio de N palabras es requerido, la lista de espacio libre es revisada hasta encontrar un bloque del menor tamaño que sea mayor o igual a N palabras. Este bloque es dividido en un bloque de N palabras y el resto del espacio es regresado a la lista de espacio libre.

Este último método aunque optimiza el espacio de almacenamiento tiene una desventaja, la cuál se refleja en el costo de recorrer toda la lista de espacio libre cada vez que se desea realizar un almacenamiento.

Recobro de Espacio con Bloques de Tamaño Variable: La solución que se presenta es el uso de un indicador entero que marque el tamaño del bloque, el cuál se deberá localizar después del bit de barrido, y así la recolección de basura se podría llevar acabo mediante la lectura de esta información. Durante esta fase de verificación se puede ejecutar una rutina de compactación que permita mantener la mayor cantidad de bloques vacíos juntos. Si combinamos ésta técnica de recolección de basura con una compactación completa podremos eliminar el uso de las listas de espacio vacío, sustituyéndolas por el uso de un solo apuntador al principio del espacio libre.

Compactación y el Problema de la Fragmentación de Memoria: Cuando se comienza el trabajo, hay un gran bloque de espacio libre, pero al continuar, la memoria se fragmenta en pequeñas piezas generadas por el almacenamiento y borrado de elementos. Al final de este proceso se da el caso de que no se puede alojar bloques de N palabras debido a que no tenemos suficiente espacio libre en forma consecutiva. Sin embargo, puede ocurrir que al contar el espacio libre que se tiene en la lista de espacio libre, se puede comprobar que hay mas espacio

libre del requerido. Para solucionar este problema se emplea la compactación de los bloques de memoria libres y hay 2 tipos:

-Compactación Parcial: cuando los bloques activos no pueden ser movidos, solo los bloques adyacentes contenidos en la lista de espacio libre puede ser compactada.

-Compactación Total: si los bloques activos en la estructura pueden ser movidos, todos los bloques activos son recorridos al inicio del stack de almacenamiento, dejando el espacio libre en el bloque final. La compactación total requiere que cuando un bloque activo es movido, todos los apuntadores a ese bloque apunten ahora a la nueva localidad.

CONCLUSIONES

Este texto, da de forma completa y sencilla, la información enfocada a las personas que programan en algún lenguaje, muestra una visión integral de los paradigmas a los que se sujetan los Lenguajes de Programación. También, menciona la importancia de los principios de los lenguajes, así como un esquema de su estudio para poder compararlos, conocer sus beneficios y elementos adversos.

Hoy en día, es más fácil tener a nuestra disposición programas con la máxima calidad y eficiencia, creados de una forma muy rápida que evoluciona porque tiene motivos importantes. Uno, es el crecimiento del conocimiento acerca de cómo hacer las cosas, su organización en técnicas y metodologías bien definidas y estructuradas, planteando alternativas como guías a seguir. Otro, es que se dispone de Lenguajes de Programación con mejores medios (herramientas como gráficos, etc), para que se desarrolle software aplicando la creatividad, ingenio y experiencia de los desarrolladores.

Desde hace años, existe una dependencia creciente de componentes de Software Re-usable por parte de programadores y equipos de desarrollo. Un ejemplo, es la Programación Orientada a Objetos y su incorporación en los Lenguajes de Programación. Es probable puedan existir proveedores de objetos componentes de software, ofrecidos a los programadores, como en la actualidad se ofrecen componentes hardware.

Por lo tanto, el enfoque del trabajo de los desarrolladores de software cambiará. Estarán los que se encargan de la integración de los objetos cuando se trata de desarrollar programas específicos, más que de desarrollar a detalle cada componente desde el principio. En tanto que los surtidores de tales componentes se especializan en desarrollar los objetos que les son demandados, que podrán ser componentes estándar o a la medida (con especificaciones únicas). Otra gran influencia en el desarrollo de programas, es resultado de la expansión en el uso de Internet en lo personal y en lo comercial.

Puede ser que se generalice el uso de programas ejecutados en Internet y los desarrolladores produzcan programas que envíen y reciban mensajes y archivos, además de que usen los inmensos recursos mundiales disponibles. Esto requerirá de Lenguajes de Programación más adaptados a este tipo de empresas.

Por eso, es necesario tener conocimiento, del Lenguaje de Programación que se va a adaptar más a nuestras necesidades de trabajo, así se va a poder obtener una máxima eficiencia en nuestro rendimiento y un resultado con alta calidad. También, se reducirán el esfuerzo y el tiempo de trabajo.

BIBLIOGRAFÍA

GENERAL

-Lenguajes de Programación, Diseño e Implementación. Terrence W. Pratt, Prentice Hall. 2ª. Edición.

-Programming languages: design and implementation. Pratt, Terrence W, Prentice Hall. 4ª. Edición.

INTRODUCCIÓN A LOS LENGUAJES DE PROGRAMACIÓN

-Programming Languages: Structures and Models. Herbert L. Dershem, Michael J. Jipping.

ASPECTOS DE DISEÑO DE LOS LENGUAJES DE PROGRAMACIÓN

-Aplique C++. Bruce Eckel. Osborne McGraw-Hill, México c1991.

ASPECTOS DE TRADUCCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

-Programming language Concepts. Carlo Ghezzi, Mehdi Jazayeri, Ed. Wiley.

-Introduction to the theory of Porgramming Languages. Bertrand Mayer. Prentice Hall International. Series in Computer Science.

TIPOS Y OBJETOS

-Peter Aitken y Bradley Jones. Aprendiendo C en 21 dias, edición Bestseller. Prentice Hall.
-Patrick Naughton y Herbert Schildt. Java. Manual de referencia. McGraw Hill.

HERENCIA

-Setra Khoshafian, Ramzmik Abnous. Object Orientation, Cencepts, Lenguages, Databases, User Interfaces. John Wiley.
-Timothy Budd. An intruduction to Object-Oriented Programing. Addison- Wesley

CONTROL DE SECUENCIA

-From Logic Programing to Prolog (Cap II). Krzyztof R. Apt. Prentice Hall.

ADMINISTRACIÓN DE ALMACENAMIENTO

-Data structures and algorithms. Aho, Hopcroft, Ullman.

REFERENCIAS

LENGUAJES http://www.inf.utfsm.cl/~rmonge/lenguaje/apuntes.html

BASIC http://www.jegsworks.com/Lessons-sp/lesson9/lesson9-2.htm

C++ http://www.jegsworks.com/Lessons-sp/lesson9/lesson9-2.htm

COBOL http://www.jegsworks.com/Lessons-sp/lesson9/lesson9-2.htm

FORTRAN http://www.geocities.com/SiliconValley/Horizon/7536/ejemplos.htm

SMALLTALK http://www.uam.edu.ni/uam99/ingieneria/prolog/Smalltalk

XML www.programacion.net/html/xml/htmdsssl/capitulo3/capitulo3.htm

PERL http://132.248.71.81/lacertus/tutorial_perl.html

LENGUAJE C http://habitantes.elsitio.com/cafcocar/Lista_programas.html

APL
http://www.engin.umd.umich.edu/CIS/course.des/cis400/index.html

SNOBOL
http://www.engin.umd.umich.edu/CIS/course.des/cis400/snobol/word.html

TIPOS DE DATOS ABSTRACTOS
http://confucius.gnacademy.org:8001/text/cc/Tutorial/Spanish/node4.html

ABSTRACT DATA TYPES
http://www.engin.umd.umich.edu/CIS/course.des/cis400/maxim/lectures/chp10.htm

LOS TIPOS ABSTRACTOS DE DATOS, SU CONCEPTO, TERMINOLOGÍA Y UNOS EJEMPLOS. http://www.infor.uva.es/~jmrr/tad2001/abstractos.htm