INDICE

CAPITULO 1

INTRODUCCIÓN A LOS LENGUAJES DE PROGRAMACIÓN

PORQUE ESTUDIAR LOS LENGUAJES DE PROGRAMACIÓN

- ***Mejorar la habilidad para desarrollar algoritmos eficaces.-*** Muchos lenguajes incluyen características que cuando se usan de forma apropiada , benefician al programador pero cuando se usan incorrectamente pueden desperdiciar grandes cantidades de tiempo de computo o conducir al programador a errores lógicos que consuman mucho tiempo.

- ***Mejorar el uso del lenguaje de programación disponible.-*** Atravez del entendimiento de cómo se implementan las características del lenguaje que uno usa, se mejora generalmente la habilidad para escribir programas mas eficaces. Por ejemplo, cuando se entiende como crea y manipula el lenguaje datos como arreglos, cadenas, listas o registros, se conocen los detalles de implementación de la recurción o se comprenden como se construyen clases de objetos que permiten construir programas mas sencillos integrados con tales componentes.

- ***Acrecentar el propio vocabulario con construcciones útiles sobre la programación.-*** El lenguaje sirve a la vez como una ayuda y como una restricción para el pensamiento. Las personas usan el lenguaje para expresar pensamientos, pero tambien para estructurar la manera como uno piensa, en la medida en que es directa en palabras. La familiaridad con un único lenguaje de programación tiende a tener un efecto similar de restricción.

- ***Hacer posible una mejor elección de un lenguaje de programación.-*** Cuando se presenta la situación, un conocimiento de diversos lenguajes puede permitir la elección de un lenguaje que sea precisamente el adecuado para el proyecto en particular, con lo cual se reduce el esfuerzo

- ***Facilitar el aprendizaje de un nuevo lenguaje***- Un lingüista, atravez de una comprención profunda de la estructura subyacente de los lenguajes naturales, suelen poder aprender un nuevo idioma con mas rapidez y facilidad que el inovado esforzado que entiende poco que incluso su lengua nativa.

- ***Facilitar el diseño de un lenguaje de programación.-*** Pocos programadores piensan alguna vez de si mismos como diseñadores de lenguajes ; sin embargo, todo programa tiene una interfaz de usuario que es, de hecho una forma de lenguaje de programación. La interfaz de usuario se componen de los comandos y formatos dados que se suministran para que el usuario se comunique con el programa, donde esta tarea es reducida por el, diseñador.

BREVE HISTORIA DE LOS LENGUAJES DE PROGRAMACION

Influencias que contribuyeron al desarrollo y evolución de los lenguajes de programación

- Capacidad de la computadora.
- Aplicaciones.
- Métodos de programación.
- Métodos de implementación.
- Estudios Teóricos.
- Estandarización.

1961-1965	HW.- Familias de arquitectura compatibles, almacenamiento en discos magneticos. METODOS.- Sistemas operativos de multiprogramación, compiladores dirigidos por sintaxis. LENGUAJES.- Cobol 61, Algol 60 (revisado), Snobol, Jovial, Notacion APL.
1966-1970	HW.-Tamaño y7velocidad crecientes y costos decrecientes, minicomputados, microprogramacion, circuitos integrados. METODOS.- Sistemas de compartición de tiempo e interactivos, compiladores de optimización, sistemas de escritura de traductores. LENGUAJES.- APL, Fortran 66, Cobol 65, Algol 68, Snobol 4,Basic, PL/I, Simula 67, Algol-w.
1971-1975	HW.- Microcomputadoras, edad de minicomputadoras, Sistemas pequeños de almacenamiento en masa, decadencia de las memorias de nucleo y acceso de las memorias de semiconductores. METODOS.- Verificación de programas, programación estructurada, Crecimiento inicial de la ingenieria de sw como diciplina de estudio. LENGUAJES.- Pascal, cobol 74, PL/I(estándar),C,Scheme, Prolog.
1976-1980	HW.- Microcomputadoras de calidad comercial, Sistemas grandes de almacenamiento en masa, computacion distribuida. METODOS.-Abstraccion de datos, semantica formal, tecnica de programacion concurrente. LENGUAJES.-Smalltalk , Ada , Fortran 77, ML.
1986-1990	HW.- Edad de las microcomputadoras, asenso de la estación de trabajo de ingenieria, arquitectura RISC, redes globales, Internet. METODOS.- Computación Cliente/Servidor LENGUAJES.-Fortran 90, c++, SML (estándar ml).
1991-1995	HW.-Estaciones de trabajo y microcomputadoras muy rapidas y de bajo costo, arquitectura masivamente paralelas, voz, video, fax, multimedia. METODOS.-Sistemas abiertos, macros de entorno, national information infraestructure (super carretera de la información) LENGUAJES.-Ada 95, Lenguajes de proceso (TCL, PERL)

CLASIFICACIÓN

***Areas De Aplicación*:**

- **Científica.-** (PASCAL, FORTRAN, ALGOL, SIMULA67).

- **Inteligencia Artificial.-(LISP**, PROLOG).

- **Programación de sistemas.-** (C, C++, MODULA-2, ENSAMBLADOR.)

- **Aplicaciones de negocios (procesamiento de datos). –** (COBOL, FOX, DBASE).

- **Propósito general.-** (C++, ADA, SIMULA67, JAVA)

- **Procesamiento de texto**

Por Nivel:

- **Alto nivel.-** (SNOBOL, PROLOG, Basic, FORTRAN, LISP, PASCAL, COBOL, C, BCL, etc.).

- **Medio nivel.-** (C, ENSAMBLADOR).

- **Bajo nivel.- (código** MAQUINA).

- **Declarativos.-**(4ta generación, fox, dbase,etc)

Paradigmas De Programación

Lenguajes Imperativos :

En estos Lenguajes las instrucciones que se utilizan están separadas por (;) este tipo de lenguajes están ligados a la propia arquitectura de Von Newman que consta en general de:

a) Una secuencia de celdas, llamada **memoria**, en la cual Se puede guardar en forma codificada, los mismos datos que instrucciones.

b) **Un procesador,** el cual es capaz de ejecutar en forma Secuencial una serie de operaciones, principalmente Aritméticas, boleanas, llamadas a comandos, etc.

Conceptos:

1. **– Datos:** números, caracteres, boleanas, guardados en celdas de Memoria.

2. **– Variables:** nombres de las celdas donde están los datos.

3. **– Instrucciones:** asignación, condicional, ciclo y entrada / Salida codificados como secuencia de comandos.

4. **– Programa:** una combinación de declaraciones, de variables y secuencia de instrucciones, la cual se puede guardar en memoira, para después de un proceso de compilación, sea ejecutada Por el procesador.

   Ejemplos: FORTRAN, ALGOL, PASCAL, C MODULA-2, ADA.

Lenguajes funcionales:

Los matemáticos desde hace buen tiempo están resolviendo problemas usando el concepto de función. Una función convierte ciertos datos en resultado.
Si se supiera como evaluar una función, usando la computadora, podría resolverse muchos problemas.
Así pensaron algunos matemáticos e inventaron los lenguajes de programación funcionales.
Además, aprovecharon las facilidades de manipulación de datos simbólicos y no solamente numéricos, que tiene las funciones, y la propiedad de composición para resolver problemas
complejos a partir de las soluciones a otros más sencillos. También se incluyó la posibilidad de definir funciones recursivamente.

Conceptos:

1.- Datos: expresiones simbólicas, agrupadas frecuentemente en listas.

2.- Funciones primitivas: conjunto de funciones predefinidas

3.- Definición de nuevas funciones: a partir de las ya existentes, utilizando la composición de funciones y la recursividad.

4.- Programa: conjunto de funciones y parámetros a evaluar.

5.- Evaluación: la ejecución de programas, consiste en la evaluación de una función aplicada a ciertos argumentos, para obtener resultados.

Ejemplos: LISP, SCHEMME, ML

Lenguajes Lógicos:

Otra forma de razonar para resolver problemas en matemáticas, se fundamenta en la lógica de primer orden. El conocimiento básico de las matemáticas se puede representar en la lógica en forma de axiomas, a los cuales se añaden reglas formales para deducir cosas verdaderas (teoremas) a partir de los axiomas.
A partir de esto se encontró la manera de automatizar computacionalmente el razonamiento lógico, que permitió que la lógica matemática diera origen a otro tipo de lenguajes de programación, conocidos como lenguajes lógicos o también como lenguajes declarativos porque el programador, para solucionar un problema, todo lo que tiene que hacer es describirlo vía axiomas y reglas de deducción.

Conceptos:

1.- Términos: que se expresan como variables, constantes y funciones.

2.- Cláusulas: predicados, denominados hechos y reglas, que expresan
Relaciones entre los términos.

3. –Meta: cláusula especifica que formula una pregunta a responder por él
Programa.

4. –Programa: conjunto de cláusulas que expresan el conocimiento del
Problema.

Ejemplos: PROLOG, SQL.

Lenguajes Orientados a Objetos:

A mediados de los 60´s, se empezó a vislumbrar el uso de las computadoras para la simulación del problema del mundo real está lleno de objetos, en la mayoría de los casos complejos, pero los cuales difícilmente se traducen a los tipos de datos primitivos de los lenguajes imperativos.

Conceptos:

1. **– Clase:** definición de un conjunto de variables y procedimientos,
Llamados métodos.

2. – Objetos: una instancia a partir de la definición de una clase, la cual tiene asociados datos a las variables y hereda los métodos Definidos en la clase.

3. – Herencia: jerarquía de clases que permiten reutilizar, en una subclase, Lo definido en las clases que le proceden en la jerarquía.

Ejemplos: C++, SIMULA 67, JAVA, etc.

ATRIBUTOS DE UN BUEN LENGUAJE

- ***Claridad, sencillez y unidad*.** Un programa debe proveer un conjunto claro, sencillo y unificado de conceptos que se puedan usar como primitivas en el desarrollo de algoritmos.
  Para esto es necesario contar con un número mínimo de conceptos distintos cuyas reglas de combinación sean tan sencillas y regulares como sea posible.
  La sintaxis de un lenguaje afecta la facilidad con la que un programa se puede escribir, poner a prueba, y más tarde entender y modificar. Una sintaxis que es particularmente tersa o hermética suele facilitar la escritura de un programa, pero dificulta su lectura cuando es necesario modificarlo más tarde.

- ***Otorgonalidad.*** Es el atributo de un lenguaje que le permite ser capaz de combinar varias características de éste en todas las combinaciones posibles, de manera que todas ellas tengan significado. Con esto es más fácil aprender el lenguaje y escribir los programas porque hay menos excepciones y cosas que recordar, aunque también este atributo tenga su lado negativo que es el hecho de que el programa se pueden compilar sin errores a pesar de tener características son lógicamente incoherentes o su ejecución es ineficiente.

- ***Naturalidad para la aplicación***. Un lenguaje necesita una sintaxis que, al usarse correctamente, permita que la estructura del programa refleje la estructura lógica subyacente del algoritmo. Los algoritmos secuenciales, algoritmos concurrentes, algoritmos lógicos, etc., todos ellos tienen estructuras naturales diferentes que están representadas por programas en esos lenguajes.

- ***Apoyo para la abstracción.*** Una parte considerable de la tarea del programador es proyectar las abstracciones adecuadas para la solución del problema y luego implantar esas abstracciones empleando las capacidades más primitivas que provee el lenguaje de programación mismo. Idealmente el lenguaje

debe permitir la definición y el mantenimiento de las estructuras de datos, de los tipos de datos y de las operaciones como abstracciones autosuficientes.

- ***Facilidades para verificar el programa.*** La confiabilidad de los programas escritos es siempre una preocupación medular. Por esto, se debe verificar que un programa ejecute correctamente la función requerida; existen varias técnicas para ello, por ejemplo a través de un método formal de verificación o por verificación de escritorio (leer y verificar visualmente). La sencillez de la estructura semántica y sintáctica es un aspecto primordial que tiende a simplificar la verificación de programas.

- ***Entorno de programación.*** La estructura técnica de un lenguaje de programación es uno de los aspectos que afecta su utilidad. La presencia de su entorno de programación adecuado puede facilitar el trabajo con un lenguaje técnicamente débil en comparación con un lenguaje más fuerte con poco apoyo externo. La disponibilidad de una implementación confiable, eficiente y bien documentada es uno de los factores que forman parte del entorno de programación.

- ***Portabilidad de programas.*** Es un criterio importante para la programación, y se refiere a que los lenguajes deben estar ampliamente disponibles y su definición deberá ser independiente de una máquina particular, es decir que los programas resultantes de la computadora en la cual se desarrollaron deben ser portables hacia otros sistemas de computadoras.

- ***Costo de uso.*** Este es un elemento importante en la evaluación de cualquier lenguaje de programación.

    - *Costo de la ejecución del programa.* Este tipo de costos es de importancia primordial para grandes programas de producción que se va a ejecutar con frecuencia. Con máquinas que trabajan a varios millones de instrucciones por segundo y que están ociosas gran parte del tiempo, se puede tolerar un incremento del 10% o 20% del tiempo de ejecución si ello significa un mejor diagnostico o un control más fácil por parte del usuario sobre el desarrollo y el mantenimiento del programa

    - *Costo de traducción de programas.* Cuando un lenguaje se utiliza en la enseñanza, es más importante la cuestión de una traducción (compilación) eficiente, que una ejecución eficiente, ya que los programas estudiantiles muchas veces se están depurando pero se ejecutan sólo unas

pocas veces. En casos así, es importante contar con un compilador rápido y eficiente en vez de uno que produzca un código ejecutable optimizado.

- *Costo de creación, prueba y uso de programas*. La preocupación por esta clase de costos de conjunto en el uso de un lenguaje se ha vuelto tan importante en muchos casos como la inquietud de la ejecución y compilación eficiente de los programas.

- *Costo de mantenimiento de los programas*. Muchos estudios han demostrado que el costo total del ciclo de vida, que incluye los costos de desarrollo y el de mantenimiento del programa, es el más grande. En mantenimiento incluye la reparación de errores, los que se descubren después de que se comienza a usar el programa, cambios que requiere el programa cuando se actualiza el hardware o el sistema operativo, etc.

LENGUAJES PARA DIVERSOS DOMINIOS DE APLICACIÓN

EPOCA	APLICACIÓN	LEN. PPALES	OTROS
Años sesentas	Negocios	Cobol	Ensamblador
	Cientifica	Fortran	Algol, basic Apl
	Sistemas	Ensamblador	Forth, Jovial
	I.A	Lisp	Snobol
Hoy	Negocios	Her. Microsoft	C, Visuales, Algol
	Cientifica	Fortran98,c, c++	Basic, Pascal
	Sistemas	C, C++	Pascal,Ada, Modula
	I.A	Scheme, Prolog	
	Edición	Text,Postcript	
	Proceso	Shell de unix, Tcl, Pert	Maquel
Nuevos	Nuevos paradigmas	ML, SMALLTALK	Eiffet

Algunos Autores De Los Lenguajes Mas Importantes

C++ .-Stroutop

C .-Dennis M. Ritchie

Basic.- Tomas Kurtz

Jovial, Ada, Forth.- Gogierno de las E.U

Snobol.- Griswold, Ivan Polonsky y David Forber

Cobol. Grace Hopper

Lisp.- Jonh McCarty

Fortran.- Jonh Bakus

EFECTOS SOBRE LOS ENTORNOS DE LOS LENGUAJES DE PROGRAMACIÓN

1.- ***Entorno de procesamiento por lotes.-***El primero y mas simple entorno operativo, se compone solo de archivos externos de datos. Un programa toma un cierto conjunto de archivos de datos como entrada, procesa los datos y procesa un conjunto de archivos de datos de salida; ejemplo, un programa de nomina. Este entorno operativo se designa como el procesamiento por lotes ya que los datos de entrada se reunen de archivos y son procesados en "lotes" del programa. Los lenguajes como Fortran, C y Pascal se proyectan inicialmente para este tipo de entorno.

- *Efectos sobre el diseño del lenguaje.-* La influencia de este entorno se aprecia en 4 áreas principales: características de entrada y salida, características de manejo de ecepciones y errores, recursos de regulación de tiempo y estructuras de programas.

2.-*Entornos Interactivos.-* Este es el mas común en la actualidad en computadoras personales y estaciones de trabajo, un programa interactua durante su ejecución directamente con un usuario en una consola de visualización, enviando alternativamente salidas hacia esta y recibiendo entradas desde el teclado o raton. Ejemplo; Los sistemas de procesamiento de texto, juegos de video, sistemas de gestión de bases de datos.

- *Efecto sobre el diseño del lenguaje.-* Las características de entrada salida interactivas son lo suficientemente diferente que las operaciones ordinarias con archivos, para casi todos los lenguajes proyectados para un entorno tengan dificultad para adaptarse en el otro. Por ejemplo el lenguaje C. Incluye funciones para tener acceso a líneas de texto desde un archivo y otras funciones que alimentan directamente cada carácter conforme lo digita el usuario en la terminal. En pascal la introducción directa de texto desde una terminal puede ser engorrosa.

3.-Entorno de sistemas incrustrados.- Un sistema de computadora que se usa para controlar parte de un sistema grande, como una planta industrial, un automóvil incluso una herramienta domestica se conoce como un sistema de computadora incrustable, el sistema de computadoras se ha vuelto parte integral del sistema mas grande y la falla del sistema computación el significa también comunmente la falla del sistema mayor, pero depende de aquel su importancia y la seguridad de su funcionamiento lenguajes como Ada, C++, favorecen este tipo de sistemas.

- *Efectos sobre el diseño del lenguaje.-* Los programas escritos para sistemas incrustados suelen operar sin un sistema operativo subyacente y sin los archivos de entorno y dispositivos de entrada y salida usuales. En vez de ello el programa debe interactuar con dispositivos distintos de los normales atravez de procedimientos especiales, estos se suelen facilitar atravez de características del lenguaje.

  Los sistemas incrustrados deben operar casi siempre en tiempo real, es decir, la operación del sistema mayor dentro del cual esta incrustrado la computadora requiere que esta sea capaz de responder a entradas y salidas dentro de intervalos de tiempo estrictamente restringidos.

4*.-Entornos de Programacion.-* Es el que resulta familiar para casi todos los programadores y es en el cual los programas se crea y pone a prueba, y que a su vez tiende a tener menos influencia sobre el diseño, del lenguaje. El entorno operativo en el cual se espera que los programas se ejecuten. Sin embargo se reconoce ampliamente que la producción de programas que operan de manera confiable y eficiente se simplifica mucho atrávez de un buen entorno de programación y de un lenguaje que permita el uso de buenas herramientas y practicas de programación. Cada herramienta de apoyo es otro programa que el programador puede utilizar como ayuda durante una o mas de las etapas de creación de un programa, estas incluyen editores, depuradores, verificadores, generadores de datos de prueba e impresoras.

- *Efectos sobre el diseño del lenguaje.-* Los entornos de programación han afectado el diseño de los lenguajes principalmente en dos áreas: las características que facilitan la compilación por separado y ensamblado de un programa apartir de componentes, así, como las características que ayudan a poner a prueba y depurar los programas.

Compilacion por separado

- Especificación de numero, orden y tipos de parametros que espera cualquier subprograma

- Declaración de tipos de datos.
- Definición de un tipo de dato.
- Compartición de datos.

Prueba y depuración

- Características para rastreo de ejecución.
- Puntos de interrupción.
- Asertos.

ESTRUCTURA Y OPERACIÓN DE UNA COMPUTADORA.

Una computadora es un conjunto integrado de algoritmos y estructura de datos capaz de almacenar y ejecutar programas.

***Computadora Real*.-** Cables, circuitos integrados, tarjetas de circuitos entre otras cosas.

Computadora Virtual.- Esta se constituye a travez de software por medio de programas que se ejecuten en otra computadora, también se les conoce como computadora simulada por software.

La computadora que ejecuta los programas traducidos puede ser una computadora de hardware o una computadora virtual compuesta en parte de hardware y en parte de software.

Una computadora consiste en 6 componentes fundamentalmente que corresponden estrechamente a los aspectos principales de un lenguaje de programación.

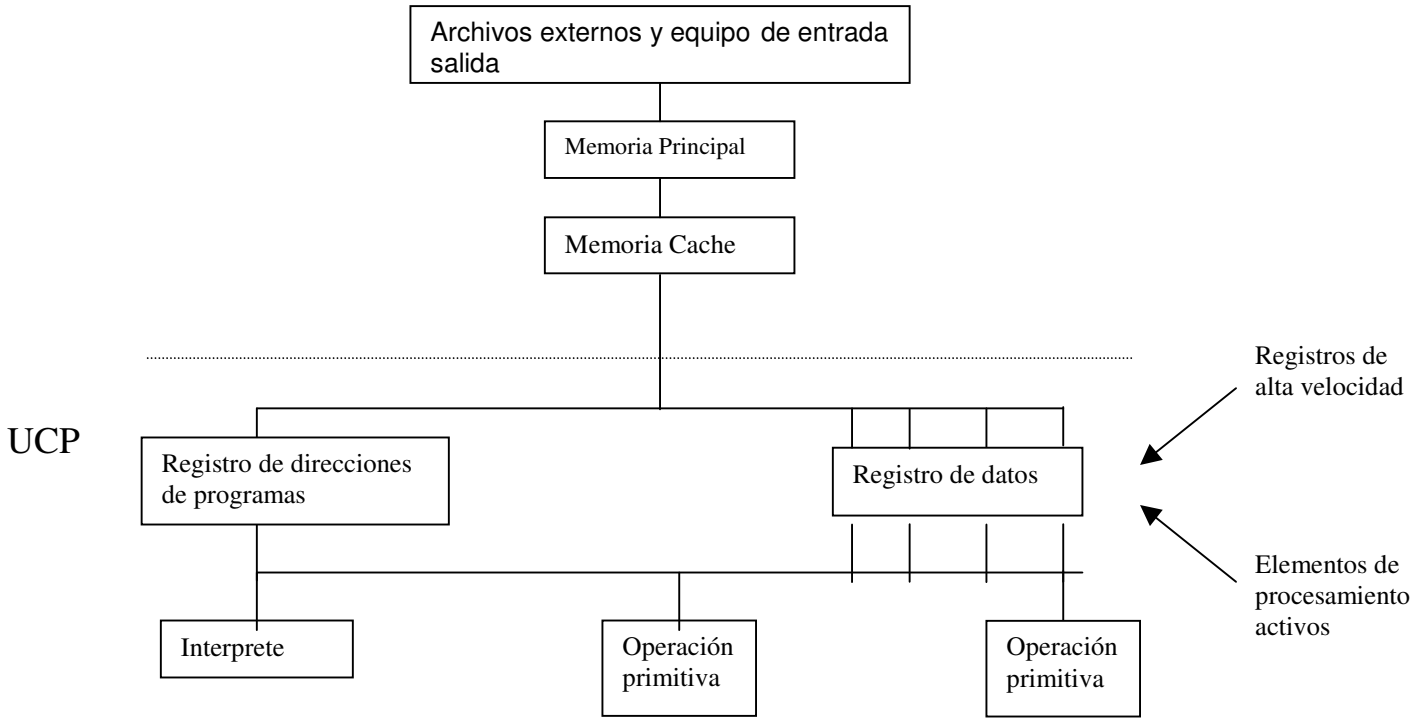
1.-*Datos.-* Una computadora debe suministrar diversas clases de datos elementales.
2.-*Operaciones primitivas.-* También debe proveer un conjunto de operaciones primitivas útiles para manipular los datos.
3.-*Control de secuencia.-* Una computadora debe aportar mecanismos para controlar el orden en que se van ha ejecutar las operaciones primitivas.

4.-*Acceso a datos.-* Una computadora debe proporcionar mecanismos para controlar los datos que se suministran en cada ejecución de una operación.

5*.-Gestion de almacenamiento***.-** Una computadora debe proveer mecanismos para controlar la asignación de almacenamiento para programas y datos.

6.-*Entorno de operaciones.-* También debe suministrar mecanismos para la comunicación con un entorno externo que contiene programas y datos que se van ha procesar.

Equipo físico de la computadora (Hardware).



Arquitecturas

CISC.- Computadora con conjunto complejo.

RISC.- Computadora de sistema de instrucciones reducido.

Capas de una computadora virtual para un programa en C

Computadora virtual desarrollada por el programador. (implemantado por el modelo de ejecución desarrollado en el programa en C para uso en la computadora virtual en C)
Computadora virtual de C. (Implementado por rutinas de biblioteca en tiempo de ejecución cargadas con el programa compilado.)
Computadora virtual del sistema operativo. (implementado por programas en lenguaje maquina en ejecución en la computadora virtual de firmware)
Computadora virtual de firmware. (instrucciones de lenguaje maquina implementadas por microcodigo ejecutado por la computadora real.)
Computadora de hardware. (Implementada por dispositivos físicos)

CAPITULO II

ESQUEMAS DE TRADUCCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN.

Traducción:

En teoría, puede ser posible construir una computadora de hardware o de firmware para ejecutar directamente el programa escrito en cualquier lenguaje, pero ordinariamente no resulta económico construir una máquina así. Las consideraciones prácticas tienden a favorecer las computadoras reales con lenguajes de máquina de nivel más bien bajo, sobre la base de velocidad, flexibilidad y costo. La programación desde luego se hace a más a menudo en un lenguaje de alto nivel muy alejado del lenguaje máquina mismo del hardware.

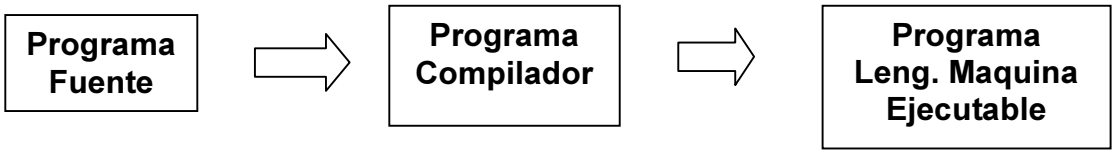
El problema que en efecto enfrenta el implementador del lenguaje es como hacer que se ejecuten los programas en el lenguaje de alto nivel en la computadora real.

TIPOS DE TRADUCCIÓN

- ***Interpretados*.** Este tipo de traducción lee y ejecuta instrucción por instrucción. LIPS, Prolog y Smalltalk, se suelen implementar a través del uso de un interprete de software.

  En esta clase de implementación de lenguajes, el traductor no procede código de maquina para la computadora que sé esta utilizando. En su lugar, el traductor no produce código de máquina para la computadora que sé esta utilizando. En su lugar, el traductor produce alguna forma intermedia del programa cuya ejecución es más fácil que la forma del programa original, pero que es distinta del código máquina.

- ***Compilados*.** Los lenguajes compilados pasan por el siguiente esquema:



Por lo común se piensa en C, Pascal, FORTRAN y ADA como en lenguajes que se compilan. Esto significa que los programas en estos lenguajes se traducen ordinariamente al lenguaje máquina de la computadora real que sé esta usando antes que comience la ejecución, y la simulación está confinada a un conjunto de rutinas de apoyo en tiempos de ejecución que simulan operaciones primitivas en el lenguaje fuente para las cuales no existe un análogo cercano en el lenguaje máquina.

Existen otros tipos de traducción-compilador

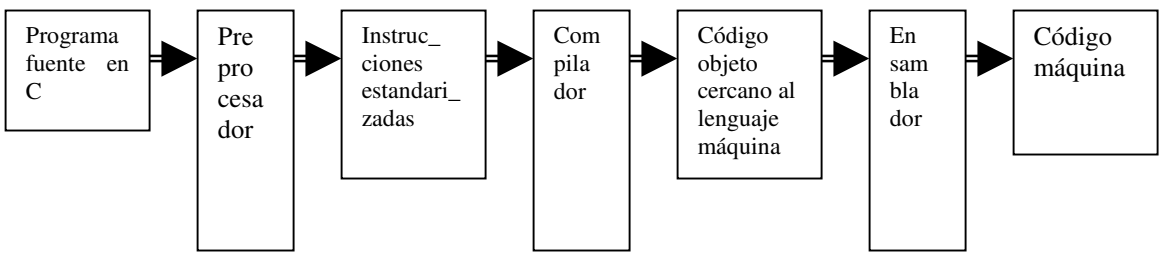
- ***Ensamblador:*** es un traductor cuyo lenguaje objeto es también alguna variedad de lenguaje máquina para una computadora real pero cuyo lenguaje fuente, un lenguaje ensamblador constituye en gran medida una representación simbólica del código de máquina objeto. Casi todas las instrucciones en el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

- ***Cargador o Editor de vínculos****:* es un traductor cuyo lenguaje objeto es un código de máquina real y cuyo lenguaje fuente es casi idéntico; y esta compuesto por lo general de programas en lenguaje máquina en forma reubicable junto con tablas de datos que especifican puntos donde el código reubicable se debe codificar para volverlo automáticamente ejecutable.

- ***Preprocesador o Macroprocesador****:* es un traductor cuyo lenguaje fuente es una forma ampliada de un lenguaje de alto nivel cuyo lenguaje objeto es la forma estandar del mismo lenguaje.

- ***Compilador****:* es un 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 ensamblador o algún otro lenguaje de máquina. C , por ejemplo se compila comunmente a un lenguaje ensamblador el cuál es convertido luego en lenguaje máquina por un ensamblador.

Traducción en C

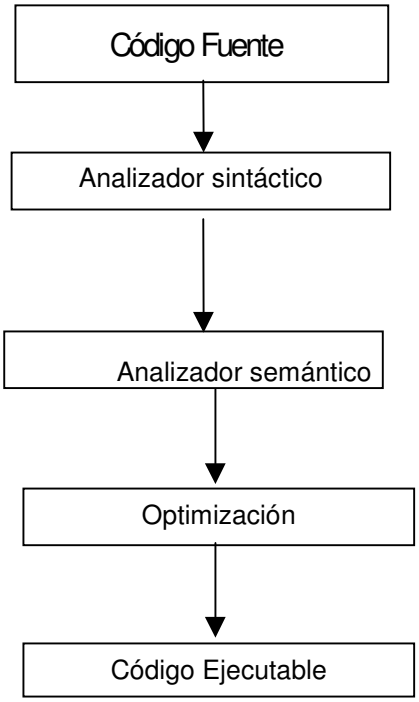


Estructura de un Compilador

En C la compilación se hace a muchos niveles por lo que es mucho más costoso.

Relación principal
- A mayor costo de traducción menor costo de eficiencia
- A menor costo de traducción mayor costo de eficiencia

Esquema De Traducción:



Implementación del Software o Simulación del Software

LISP, ML, Prolog, Smalltalk y BASIC se suelen implementar através del uso de un intérprete de software. En esta clase de implementación de lenguajes, el traductor no produce código de máquina para la computadora que se esta utilizando. En su lugar, el traductor produce alguna forma intermedia del programa cuya ejecución es más fácil que la de la forma del programa original, pero que es distinta del código máquina. El procedimiento de implementación para la ejecución de esta forma traducida se debe de representar por medio de software por que el intérprete de hardware no se puede usar directamente. Ordinariamente, el uso de un intérprete de software da por resultado una ejecución relativamente lenta del programa.

Sintaxis y Semántica

En los manuales de lenguaje de programación y en otras descripciones de lenguajes se acostumbra organizar la descripción del lenguaje entorno a las diversas construcciones sintácticas del mismo. Tipicamente, se da la sintaxis para una construcción de lenguaje como un tipo particular de enunciado o declaración, y luego se da también la semántica para esa construcción, la cuál describe el significado deseado. Por ejemplo: la computadora virtual de Pascal puede usar un vector "v" durante la ejeución de un programa, donde "v" tiene una estructura que esta dada directamente por la declaración anterior. Sin embargo, la computadora virtual de pascal puede tener otras estructuras de datos, como una pila central del subprograma "registros de activación", que no se ven directamente en absoluto en la sintaxis de los programas.

Sintaxis: La sintaxis de un lenguaje de programación es el aspecto que ofrece el programa. Proporcionar las reglas de sintaxis para un lenguaje de programación significa decir como se escriben los enunciados, declaraciones y otras instrucciones.

Elementos sintácticos de un lenguaje

1. ***Conjunto de caracteres:*** la elección del conjunto de caracteres es la primera que se hace al proyectar una sintaxis de lenguaje. Existen varios conjuntos de caracteres de uso amplio, como el conjunto ASCII, cada uno con un conjunto diferente de caracteres especiales además de las letras y dígitos básicos. La elección del conjunto de caracteres es importante en la determinación del tipo de equipo de entrada y salida que se puede usar al implementar el lenguaje.

2. ***Identificadores:*** la sintaxis básica para identificadores, una cadena de letras y dígitos que comienza con una letra, tiene amplia aceptación. Las variaciones entre lenguajes se da principalmente en la inclusión opcional de caracteres especiales (. _ -) para mejorar la legibilidad y en restricciones de longitud.

3. ***Símbolos de operadores:*** casi todos los lenguajes emplean los caracteres especiales "+" y "-" para representar las 2 operaciones aritméticas básicas, pero más allá de esto casi no existe uniformidad. Casi todos los lenguajes adoptan alguna combinación que utilizan caracteres especiales para ciertos operadores, identificadores para otros, así como algunas cadenas de caracteres.

4. ***Palabras clave y reservadas:*** una palabra clave es un identificador que se usa como una parte fija de la sintaxis de un enunciado ejemplo: if al principio del enunciado condicional en FORTRAN o Do al comenzar un enunciado de iteración. En FORTRAN la palabra clave es una palabra reservada si no se puede usar como un identificador elegido por el programador.

5. ***Palabras pregonadas:*** son palabras opcionales que se incertan en los enunciados para mejorar la legibilidad. COBOL ofreció muchas opciones de este tipo ejemplo: en el enunciado GO TO se escribe GO TO

rotulo, se requiere la palabra clave GO pero To es condicional; no repite información y solo se usa para mejorar la legibilidad.

6. ***Comentarios:*** la inclusión de estos en un programa es una parte importante de su documentación. Un lenguaje puede permitir comentarios de varias maneras:
   a) Renglones de comentarios por separado en el programa como FORTRAN
   b) Delimitados por marcadores especiales como los /* y */ de C sin que importe los límites del renglón.
   c) Comenzando en cualquier punto de un renglón pero ya concluidos al final del mismo como el guión en ADA, // en C++ o la admiración en FORTRAN90.

7. ***Espacios en blanco:*** las reglas con el uso de espacios en blanco varian ampliamente entre los lenguajes. En FORTRAN por ejemplo, los espacios en blanco no son significativos en cualquier parte excepto en datos literales de cadena de carácter. Otros lenguajes usan espacios en blanco como separadores de modo que desempeñan un papel sintáctico importante.

8. ***Delimitadores y corchetes:*** un delimitador es un elemento sintáctico que se usa simplemente, para señalar el principio o el final de alguna unidad sintáctica, como un enunciado o expresión. ej: () y ;. Los corchetes son delimitadores apareados ej: ( ) o parejas de begin-end, {}, “”. Los delimitadores se pueden usar simplemente para mejorar la legibilidad o simplificar el análisis sintáctico, pero con más frecuencia tienen el importante propósito de eliminar ambigüedades.

9. ***Formatos de campo libre y fijo:*** una sintaxis es de campo libre si los enunciados del programa se pueden escribir en cualquier parte de un renglón de entrada sin que importe la opción sobre el renglón o las interrupciones entre renglones. Una sintaxis de campo fijo utiliza la posición sobre un renglón de entrada para transferir información. La sintaxis de campo fijo estricta, donde cada elemento de un enunciado debe aparecer dentro de un renglón de entrada, se observa más a menudo en lenguajes ensambladores. Esta sintaxis es cada vez más rara. En la actualidad y la norma es en campo libre.

10. ***Expresiones:*** son funciones que acceden a objetos de datos en un programa y devuelven algún valor. También son los bloques sintácticos básicos de construcción a partir de los cuales se construyen enunciados (y a veces programas).

11. ***Enunciados:*** los enunciados constituyen el componente sintáctico más destacado en los lenguajes imperativos. Que es la clase dominante de lenguajes que se usan en la actualidad. Su sintaxis tiene un efecto decisivo sobre la regularidad, legibilidad y facilidades de escritura generales del lenguaje.
    Ciertos lenguajes adoptan un formato único de enunciado básico, mientras que otros emplean diferentes sintaxis para cada tipo distinto de enunciado. El primer enfoque hace énfasis en la regularidad, en tanto que el segundo resalta la legibilidad. Ejemplo: Snobol 4 tiene solo una sintaxis básica de enunciados, que es el enunciado de sustitución por concordancia de patrones, a partir del cual se pueden derivar otros tipos de enunciados. Casi todos los lenguajes se inclinan hacia el otro extremo de suministrar diferentes estructuras sintácticas para cada tipo de enunciado. Ejemplo: Cobol.

Semántica: La semántica de un lenguaje de programación es el significado que se da a las diversas construcciones sintácticas. Por ejemplo para proporcionar la sintaxis que se usa en Pascal para declarar un vector de 10 elementos de enteros se daría la declaración siguiente:

```
Var
```

Vect: array [1...10] of integer;

Por otra parte en C se expresa así:

Int vect [10];

Enlaces y Tiempo de Enlaces

Enlace: Se podría decir que el enlace de un elemento de programa a una característica o propiedad particular como simplemente la elección de la propiedad de entre un conjunto de propiedades posibles el momento durante la formulación o procesamiento del programa en el que se hace esta elección se conoce como tiempo de enlaces de esa propiedad para ese elemento. Existen muchas variedades distintas de enlaces en los lenguajes de programación, así como una diversidad de tiempos de enlaces. Lo existen también por definición de lenguaje o por su implementación. Ejemplo.- la asignación de una variable.

Clases de tiempo de enlaces.

Si bien no existe una clasificación simple de los diferentes tipos de enlaces, es posible distinguir unos cuantos tiempos de enlaces si se recuerda el supuesto básico de que el procesamiento de un programa, independientemente del lenguaje, siempre intervienen un paso de traducción seguido de la ejecución del programa traducido.

1.- ***Tiempo ejecución:*** muchos enlaces se llevan a cabo durante la ejecución del programa. Esto incluye enlaces de variables a sus valores, así como (en muchos lenguajes) el enlace de variables a localidades particulares de almacenamiento.
Es posible distinguir 2 subcategorías importantes :

a) *Al entrar a un subprograma o bloque.* En casi todos los lenguajes las clases importantes de enlaces solo pueden ocurrir en el momento de entrar a un subprograma o bloque durante la ejecución. Por ejemplo: en C y en Pascal el enlace de parámetros formales a reales y el enlace de parámetros formales a localidades particulares de almacenamiento solo pueden ocurrir al entrar a un subprograma.

b) *En puntos arbitrarios durante la ejecución.* Ciertos enlaces pueden ocurrir en cualquier punto durante la ejecución de un programa. El ejemplo más importante en este caso es el enlace básico de variables a valores a través de asignación, en tanto que si estos lenguajes como LISP y ML también

permiten que ocurra el enlace de nombres a localidades de almacenamiento en puntos arbitrarios del programa.

2.-***Tiempo de traducción (o compilación):*** Se pueden distinguir 3 clases distintas de enlaces en tiempo de traducción:

a) *Enlaces elegidos por el programador:* al escribir un programa, el programador toma concientemente muchas decisiones respecto a opciones de nombres de variables, tipos para las variables, estructuras de enunciados de programas, etc, que representan enlaces durante la traducción. El traductor de lenguaje utiliza estos enlaces para determinar la forma final del programa objeto.

b) *Enlaces elegidos por el traductor:* ciertos enlaces son elegidos por el traductor del lenguaje sin que el programador los especifique directamente. Ejemplo: la localidad relativa de un objeto de datos en el almacenamiento asignado para un procedimiento se maneja en general sin el conocimiento del programa.

c) *Enlaces elegidos por el cargador:* un programa se compone por lo común de varios subprogramas que se deben fusionar en un programa ejecutable único. El traductor enlaza típicamente variables a direcciones dentro del almacenamiento designado para cada subprograma. Sin embargo, este almacenamiento debe tener asignadas direcciones reales dentro de la computadora física que va a ejecutar el programa. Este enlace también es llamado tiempo de vinculación.

3.- ***Tiempo de implementación:*** ciertos aspectos de una definición de lenguaje pueden ser iguales para todos los programas que se ejecutan usando una implementación particular de un lenguaje, pero pueden presentar variaciones entre diferentes implementaciones. Ejemplo: los detalles asociados con la representación de números y operaciones suelen estar determinados por la manera como se efectua la aritmética en una computadora de hardware subyacente.

4.- ***Tiempo de definición de lenguaje:*** casi toda la estructura de un lenguaje de programación se fija cuando el lenguaje se define, en el sentido de la especificación de las alternativas disponibles para un programador cuando escribe su programa. Ejemplo: todas las posibles formas opcionales de enunciados, tipos de estructura de datos, estructuras del programa, etc., suelen fijarse en tiempo de definición de lenguaje.

Para ilustrar la diversidad de enlaces y tiempo de enlaces considerece este sencillo enunciado de asignación: *x:=x+10;* Supóngase que este enunciado apareció dentro de algún programa escrito en un lenguaje "x". Se podría investigar acerca de los enlaces y tiempos de enlace de al menos los elementos siguientes de este enunciado:

a) *Conjunto de posibles tipos para la variable x*. En conjunto de tipos permisibles para x suele fijarse durante la definición del lenguaje.

b) *Tipo de variable x*. El tipo particular de datos asociados a la variable x se suele fijar durante la traducción, a través de una declaración en el programa. En otros lenguajes como Smalltalk y Prolog, el tipo de datos de x puede enlasarce solo durante la ejecución a través de la asignación de un valor de tipo particular.

c) *Conjunto de variables posibles para la variable x*. Si x tiene tipo de datos real, entonces su valor en cualquier punto durante la ejecución es uno de un conjunto de series de bits que representan números reales. Así pues, el conjunto de valores posibles para x se puede determinar durante la

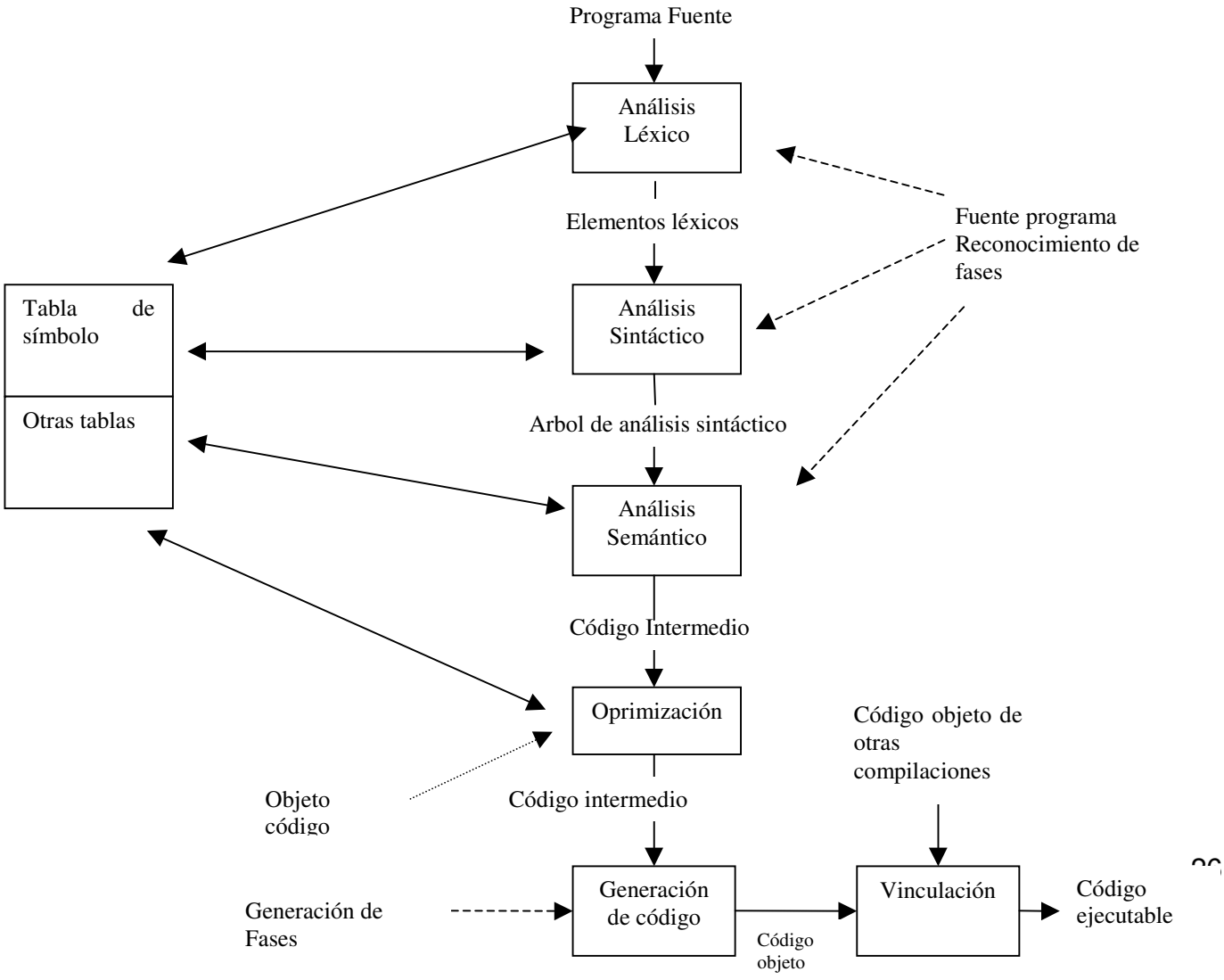
implementación del lenguaje; distintas implementaciones de lenguaje puede permitir diferentes intervalos de valores posibles para x.

d) *Valor de la variable x*. En cualquier punto durante la ejecución del programa, un valor particular esta enlazado a la variable x.

e) *Representación de la constante 10*. El entero 10 tiene ala vez una representación como constante en los programas usando la cadena 10, y una representación durante la ejecución casi siempre como una serie de bits. La elección de la representación decimal en el programa (Ejemplo: 10 para 10) se hace por lo común en el tiempo de definición de lenguaje.

f) *Propiedades del operador "+".* La elección del símbolo "+" para representar la operación de adición se hace en el tiempo de definición del lenguaje. Sin embargo, es común permitir que el mismo símbolo "+" represente homonimia, adición real, adición entera, adición compleja, etc., según el contexto. En un lenguaje compilado es frecuente hacer la determinación de cual operación se representa con "+".

ETAPAS DE LA TRADUCCION



Una traducción puede ser bastante sencilla, como en el caso de programas en Prolog o LISP pero, con frecuencia el proceso puede ser bastante complejo. Casi todos los lenguajes se podrían implementar con solo una traducción trivial si uno estuviera dispuesto a escribir un intérprete se software y aceptar velocidades lentas de ejecución. Sin embargo, la ejecución eficiente es un objeto deseable al traducir programas y llegar a códigos de máquina interpretables por el hardware.

- ***Análisis léxico:*** la fase fundamental de cualquier traducción es agrupar una serie de caracteres (programas) es sus constituyentes elementales: identificadores, delimitadores, símbolos de operadores, etc. Las unidades básicas resultantes se le llaman elementos o componentes léxicos. Típicamente el analizador léxico es la rutina de entrada del traductor y suele requerir mayor proporción de tiempo en la traducción, ya que se hace el análisis carácter por carácter.

- ***Análisis sintáctico:*** es la segunda etapa, también se le conoce como PARSING. En esta se identifican las estructuras del programa más grandes como son los enunciados, declaraciones, expresiones, etc. Usando los elementos léxicos producidos por la etapa anterior. El análisis sintáctico se alterna ordinariamente con el análisis semántico. Primero el análisis sintáctico identifica una serie de eventos léxicos que forman una unidad sintáctica como una expresión, llamada a subprogramas, enunciado, etc. Se llama entonces a un analizador semántico para que procese esta unidad. Esto se hace a través de una pila donde el analizador sintáctico introduce los elementos de la unidad a esta y el analizador semántico lo recupera.

- ***Análisis Semántico:*** es tal vez la fase medular de la traducción. Aquí se procesan las estructuras sintácticas reconocidas por el analizador sintáctico y la estructura del código objeto ejecutable comienza a tener forma. Y el análisis semántico es por lo tanto el puente entre las partes de análisis y síntesis de la traducción.
  Ocurren también en esta etapa otras funciones importantes como lo son el mantenimiento de tabla de símbolos de detección de errores, expansión de macros y ejecución de enunciados de tiempo de compilación. La salida de esta etapa es en efecto código ejecutable en traducción sencilla pero es más común que la salida sea alguna forma intermedia de programa ejecutable final, la cual sirve de entrada para su etapa de optimización del traductor. El analizador semántico se olvide ordinariamente en un conjunto de analizadores semánticos más pequeños donde cada uno maneja un tipo en particular de construcción del programa , interactuando entre ellos a través de la tabla central de símbolos.

- ***Optimización:*** el analizador produce ordinariamente como salida el programa ejecutable traducido representado en algún código intermedio. Una representación interna como una cadena de operadores y operandos con un atabla de series de operador/operando. A partir de esta representación interna, los generadores de código pueden crear el código objeto de salida con el formato apropiado. Ejemplo:

  A=B+C+D

  Código intermedio: (a) Temp1=B+C

  (b)Temp2=Temp1

(c)A=Temp2

El cual puede generar el código sencillo aunque ineficiente.

Cargar registro con B (a partir de (a))
sumar C al registro
Guardar registro en Temp1
Cargar registro con temp1(a partir de (b))
Sumar D al registro
Guardar registro en Temp2
Cargar registro en Temp2 (a partir de (c))
Guardar registro en A

CRITERIOS GENERALES DE SINTAXIS

El propósito primordial de la sintaxis es preveer una notación para la comunicación entre el programador y el procesador de lenguajes de programación. Los detalles son sintaxis se eligen en gran medida con base a criterios secundarios, los cuales no tienen relación con el objeto primario de comunicar información al procesador de lenguajes. Existen muchos criterios secundarios, pero los generales están orientados a hacer que los programas sean fáciles de leer, escribir, verificar, traducir y no ambiguos.

1. ***– Legibilidad.*** Un programa es legible si la estructura subyacente del algoritmo los datos que el programa representa quedan de manifiesto al inspeccionar el texto del programa. La legibilidad se mejora a través de las características del lenguaje tales como formatos naturales de enunciados, enunciados estructurados, uso abundante de palabras clave y palabras pregonadas, recursos para comentarios incrustados, símbolos nemotécnicos de operadores, etc.

2. *– Facilidad de Escritura*. Las características sintácticas que hace que un programa sea fácil de escribir suelen hallarse en conflicto con la que facilitan su lectura. Este criterio se mejora a través del uso de estructuras sintácticas concisas y regulares, por ejemplo C tiene por desgracia este atributo de proveer recursos para programas muy concisos de difícil lectura.

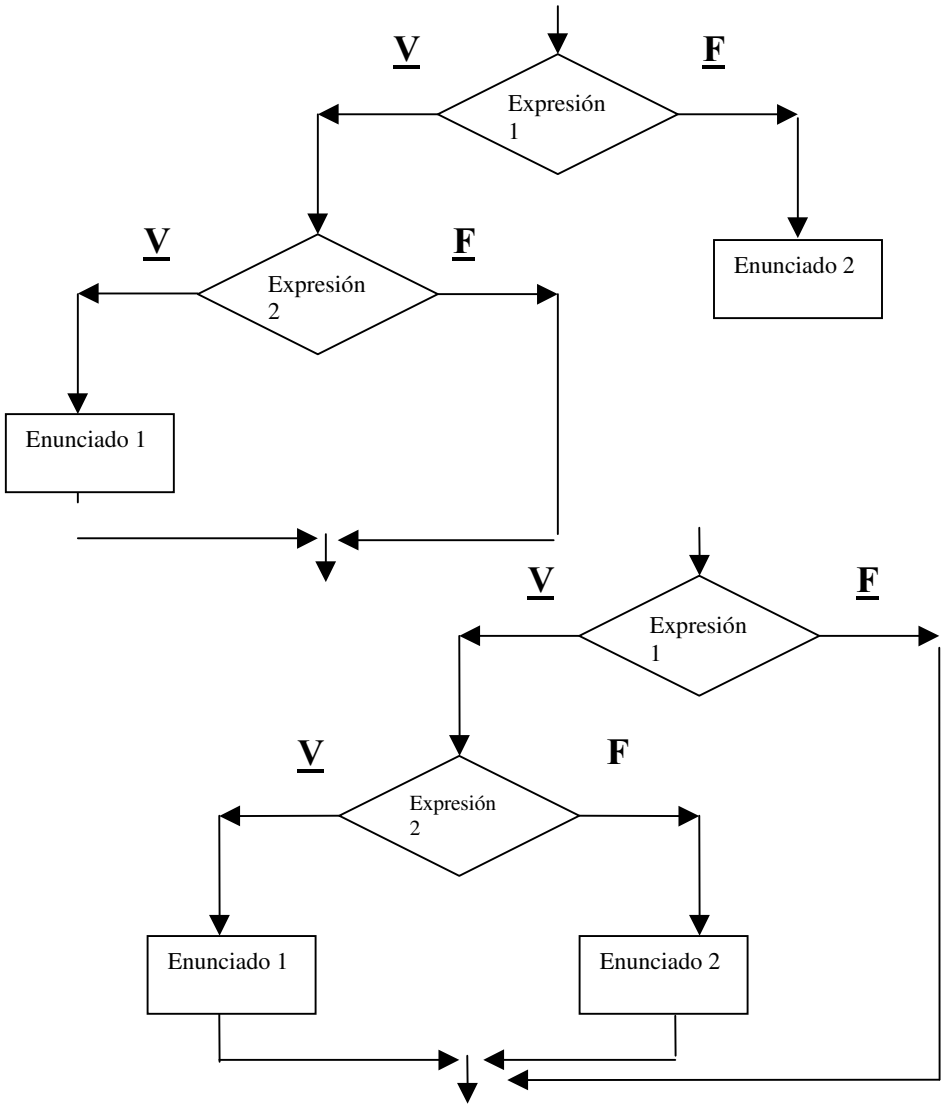
Una sintaxis es redundante si comunica el mismo elemento de información en mas de una forma. Cierta redundancia es útil porque facilita la lectura del programa; la desventaja es que la redundancia hace a los programas más elocuentes y por lo tanto dificulta su escritura, y traducción.

3. – Facilidad de Verificación. Tiene relación con la legibilidad y facilidad de escritura, el concepto de corrección del programa o verificación del programa. Ya que es difícil crear programas correctos, se necesitan técnicas que permitan probar que el programa es sintácticamente correcto.

4. - Facilidad de Traducción. Un tercer objetivo en conflicto es hacer que los programas sean fáciles de traducir a una forma ejecutable. La legibilidad y facilidad de escritura son criterios dirigidos a las necesidades del

programador humano. La facilidad de traducción se relaciona con las necesidades del traductor que procesa el programa escrito. La clave para una traducción fácil es la que se vea de la estructura.

5. – Carencia de Ambigüedad. La ambigüedad es un problema medular en todo diseño de lenguaje. Una definición de lenguaje proporciona idealmente un significado único para cada construcción sintáctica que el programador puede escribir. Una construcción ambigua permite dos o más interpretaciones distintas. El problema de ambigüedad surge por lo común no en la estructura de elementos individuales de programa, sino en la interacción entre diferentes estructuras.
Ambigüedad: 2 interpretaciones de un enunciado condicional



1.- if expresión boleana then begin if expresión booleana2 then enunciado1 end else enunciado2.
2.- if expresión boleana1 then begin if expresión booleana2 then enunciado1 else enunciado2 end.

MODELOS FORMALES DE TRADUCCION

Modelo Formal.

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

Gramática BNF

Esta notación especifica se conoce como BNF y fue desarrollado para la definición sintáctica de Algol por John Backus como una forma de expresar ideas para los lenguajes de programación. Peter Naur fue el presidente del comité que desarrollo Algol.

Sintaxis:

Una gramática BNF se compone de un conjunto finito de reglas de gramática. Un lenguaje de programación, es considerado desde el punto de vista sintáctico, por la forma y no por el significado. Se compone de un conjunto de programas sintácticamente correctos cada uno de los cuales es simplemente una serie de caracteres.

:= "se define como"

| "o"

:: "significa"

< Dígito> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Entero sin signo> ::= <dígito> | <entero sin signo> <dígito>

<Entero con signo>::= + <Entero> | - < Entero>

<Enunciado de asignación> ::= <Variable> = <Expresión aritmética>

<Expresión aritmética> ::= <Término>| <Expresión aritmética> + <Termino> / <Primario>

<Termino>::= <Primario> | <Termino>* <Primario> | <Término> / <Primario>

<Primario>::= <Variable> | <Número> | (<Expresión aritmética>)

<Variable>::= < Identificador> | <identificador>

<Identificador> ::= <Letra> { <Letra>|<digito>}

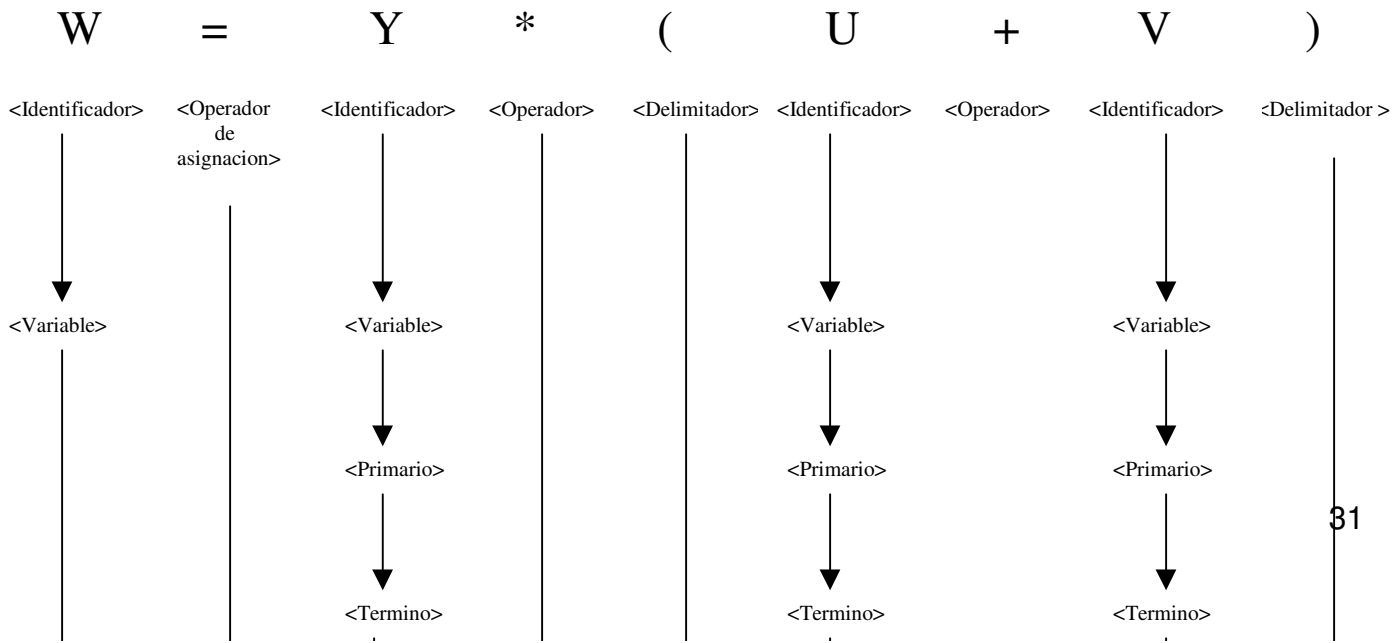
<Lista de subíndices>::= <Expresión aritmética>| <Lista de subíndices>, <Expresión aritmética>

Un lenguaje es cualquier conjunto de cadenas de caracteres (de longitud finita) con caracteres elegidos de algún alfabeto finito, fijo de símbolos.

Bajo esta definición, todo los lenguajes:

1. El conjunto de todos los enunciados de asignación en C.
2. El conjunto de todo los programas en C.
3. El conjunto de todos los átomos en LISP.
4. El conjunto compuesto de secuencias de letras A y B donde todas las A anteceden a todas las B (por ejemplo: AB, AAB, ABB, etc.)

Ejemplos:



CAPITULO 3

TIPOS DE DATOS ELEMENTALES, DE ESTRUCTURAS Y ABSTRACTOS

Dato:

Es un hecho susceptible de ser registrado y se obtiene mediante la observación, medición, análisis, etc. del mismo.

TIPOS DE DATOS ELEMENTALES DE ESTRUCTURAS Y ABSTRACTOS

ENLACE { **Objeto de Datos**
Valor de datos

OBJETO DE DATO

Objeto de Dato: Se refiere a un agrupamiento en tiempo de ejecución de uno a mas datos a través de computadora virtual. Algunos de los objetos de datos que existen durante la ejecución de programas son:

Definidos por el programador: Es decir son las variables, constantes, arreglos, archivos, etc., que el programador crea y manipula de manera explícita a través de declaraciones y enunciados en el programa.

Definidos por el sistema: Son objetos de datos que la computadora virtual construye para el "mantenimiento" durante la ejecución de programas y a los cuales el programador no tiene acceso directo, como pilas de almacenamiento en tiempo de ejecución, registros de activación de subprogramas, memorias intermedias de archivos y listas de espacio libre.

Un objeto de dato representa un recipiente para valores de datos, un lugar donde los valores de datos se pueden guardar y más tarde recuperar. Un objeto de dato se caracteriza por un conjunto de atributos, el más importante de los cuales son tipos de datos. Los atributos determinan el número y tipo de valores que el objeto de datos puede contener y también prescriben la organización lógica de esos valores.

Valor de Dato:

Puede ser un solo numero o carácter o posiblemente un apuntador a otro objeto de dato. Se representa por medio de un patrón de bits.

VARIABLE Y CONSTANTE

VARIABLE: Los objetos de datos que el programador define y nombra explícitamente en un programa, se conoce como una variable. Una variable simple en un objeto elemental de datos con nombre.

CONSTANTE: Los objetos de datos con nombre que está enlazado a un valor en forma permanente durante su tiempo de vida.

ENLACES Y ATRIBUTOS DE UN OBJETO DE DATO

1. **TIPO.-** Esta asociación se hace por lo común en el tiempo de traducción del programa asociado al objeto de dato en el conjunto del valor de dato que el objeto puede tomar.

2. **LOCALIDAD.-** El enlace de una localidad de almacenamiento en la memoria donde el objeto de dato esta representado no es ordinariamente modificable por el programador, sino que es establecido y puede ser codificado por las rutinas de gestión de almacenamiento en la computadora virtual.

3. **VALOR.-** Este enlace es por lo general el resultado de las operaciones de asignación.

4. **NOMBRE.-** El enlace a uno o mas nombres por medio de los cuales se puede hacer referencia al objeto durante la ejecución del programa. Se establece por lo común mediante declaración y se modifica a través de llamadas y devoluciones de subprogramas.

5. **COMPONENTES.-** El enlace de objeto de datos a uno o mas objetos de datos de los cuales es un componente. Se suele representar con un valor de apuntador y se puede modificar a través de un cambio en el apuntador.

TIPOS DE DATOS ELEMENTALES

Es una clase de objeto de dato ligada a un conjunto de operaciones para enlazarlos y manipularlos.

Todo lenguaje tiene un conjunto de tipos primitivos de datos que están integrados a este. Además un lenguaje puede proveer recursos que permitan al programador definir nuevos tipos de datos.

La diferencia principales entre lenguajes mas antiguos como FORTRAN y COBOL y los modernos ADA y C esta en el área de tipo de datos definidos por el programador. La tendencia mas reciente es dejar que los tipos mismos sean manipulados por el lenguaje de programación. Esta es una característica importante incorporado al ML y es una de las que esta presente en los modelos de programación orientado a objetos.

Los elementos básicos de una especie de tipo de dato son:

1. ATRIBUTOS.- que distingue objetos de datos de este tipo.
2. VALORES.- que los objetos de datos de este tipo de dato puede tomar.
3. OPERACIONES.- que definen las posibles manipulaciones de objeto de dato de este tipo.

EJEMPLO: Si se considera la especificación de un tipo de dato de arreglo, los atributos podrían incluir, el numero de dimensiones y el intervalo de sus índices para cada dimensión y el tipo de dato de los componentes; Los valores son los conjuntos de números que forman valores validos para componentes de arreglos y las operaciones podrían incluir la subindización para seleccionar componentes particulares del arreglo y otras operaciones para crear arreglos, y cambiar su forma y efectuar aritmética sobre parejas de arreglos.

Los elementos básicos de implementación de un tipo de dato son:

1. Representación de almacenamiento.
2. Manera en las operaciones definidas por el tipo de dato se representa en termino de algoritmo o procedimientos particulares que manipulan la representación

Declaraciones

Es un enunciado del programa que sirve para comunicar al traductor del lenguaje información acerca del nombre y tipo de los objeto de datos. que se necesitan durante la ejecución del programa. También puede servir para indicar los tiempos de vida deseados de los objetos de datos.
Tiempo de vida: Momento que aparece en memoria hasta que desaparece de ella.

Verificación de tipos

Significa comprobar que cada operación en un programa ejecuta , recibe el numero apropiado de argumentos del tipo correcto del dato. La verificación de tipos se puede hacer en la tiempo de ejecución (verificación dinámica de tipos) o en el tiempo de compilación o traducción (verificación estática de tipos).

La verificación dinámica de tipos por lo común se efectúa inmediatamente antes de la ejecución de una operación particular, y se implementa guardando una marca de tipo en cada objeto de dato.

Casi todos los lenguajes intentan eliminar o reducir al mínimo la verificación dinámica de tipos efectuando la verificación durante la compilación (verificación estática). Esta se efectúa durante la traducción de un programa, y la información necesaria se subministra en parte mediante las declaraciones que proporciona el programador.

TIPOS ELEMENTALES DE DATOS

1. **NUMERICO.-** En casi todos los lenguajes se encuentra alguna forma de datos numéricos. Los mas comunes enteros, reales, ya que suelen manejarse directamente en el hardware de la computadora.

   - **Enteros:** Un objeto de dato de tipo entero su único atributo regularmente es su tipo siendo su conjunto de valores un subconjunto ordenado dentro de ciertos limites finitos. (maxint 32567 a -32567). Las operaciones son: Asignación, relaciones, operaciones de bits y operaciones aritméticas.
   - **Subintervalo:** Un Subintervalo de un tipo entero de datos es un subtipo del tipo entero de dato y consiste en una serie de valores entero dentro de un cierto intervalo restringido. Ejemplo: los enteros en el intervalo del 1 al 10 o de –50 a 50. Se suele hacer una declaración de la forma
     En pascal a: 1..10;
     En ada a: integer range 1..10

   El conjunto de operaciones para un tipo de Subintervalo es el mismo que para un tipo entero ordinario.

   Aspectos importante sobre implementación:
   1.- Requerimiento de almacenamiento mas reducido.
   2.- Mejor verificación de tipo.
   - **Números reales de punto flotante:** Sus valores forman una serie ordenada de un rango negativo mínimo hasta un valor máximo determinado por el hardware (maquina convencional) pero los valores no están distribuidos de manera uniforme a través de este intervalo. De manera alternativa, en términos de número de dígitos que se usa en la representación decimal, puede ser especificada por el programador, (ejemplo c y ada). Las operaciones son: Aritméticas (a + b), asignación (a = b), las operaciones Booleanas a veces están restringidas debido a cuestiones de redondeo relacional (A == B), nivel bit (A << B).
   - **Números reales 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. (2.50 , 3.10).

2. **ENUMERACIONES.-** Con frecuencia se desea que una variable adopte solo uno de un numero reducido de valores simbólicos. Ejemplo
   (C) enumcolor =(rojo, verde, azul);
   (PASCAL) color(rojo, verde, azul);

   Una ENUMERACIÓN es lista ordenada de valores distintos.
   Las Operaciones Básicas son asignación, relacionales y las operaciones sucesor y predecesor.

3. **BOOLEANA:** también se conoce como TIPO de dato LÓGICO y se utiliza para representar cierto y falso. Operaciones Asignación, conjunción, disyunción inclusiva y negación o complemento y a veces equivalencia o exclusivo, implicación, entre otras.
   Su representación de almacenamiento es un bit, en la memoria su representación suele ser un byte por que los bits no pueden ser direccionables por separados.

4. **CARACTERES:** Casi todos los datos entran o salen en forma de caracteres. La cadena de caracteres se suele procesar como una unidad.

Ya sea directamente provistas a través de un tipo de dato de cadena de caracteres (PROLOG, ML, PASCAL) o a través de un tipo de dato de carácter donde una cadena de caracteres se considera como un arreglo lineal (C, ADA).

Un tipo de dato de carácter proporciona objetos de datos que tiene un solo carácter como su valor. El conjunto de posibles valores de carácter se toma ordinariamente como una enumeración definida por el lenguaje que corresponde a los conjuntos normales de caracteres que maneja el hardware y sistemas operativos subyacentes, el conjunto de carácter ASCII.

Las operaciones sobre datos de carácter incluyen relacionales, asignación y a veces operaciones para probar si un valor de carácter pertenece a las clases especiales "letra", "digito" o "carácter especial".

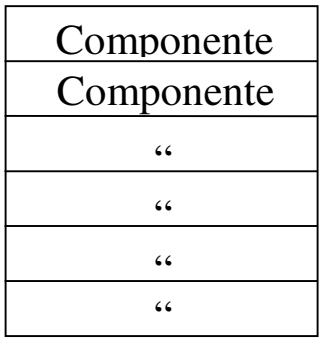
5. **TIPOS DE DATOS ESTRUCTURADOS:** Una estructura de datos es un objeto de datos que contiene otros objetos de datos como sus elementos o componentes. Los tipos mas importantes son arreglos, registros, cadenas, pila, listas, apuntadores y archivo. Un componente puede ser elemental o puede ser otra estructura de datos, Elementos: un componente de un arreglo puede ser un numero o un registro, una cadena de caracteres u otro arreglo.

REPRESENTACIONES DE ALMACENAMIENTO: Esto para una estructura de datos incluye:

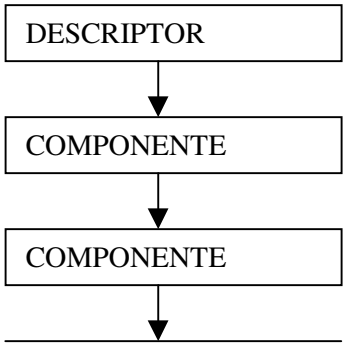
1. Almacenamiento para los componentes de la estructura.
2. Un descriptor optativo que guarda algunos (o todos) de los atributos de la estructura.

Las dos representaciones básicas son:

1. REPRESENTACION SECUENCIAL: En el cual la estructura de datos guarda en un solo bloque contiguo de almacenamiento que incluye tanto descriptor como componentes.
2. REPRESENTACION VINCULDA: En el cual le estructura de datos se guarda en varios bloques no contiguos de almacenamiento con los bloques vinculados entre si a través de apuntadores. Un apuntador del bloque A al bloque B llamado vinculo se representa guardando la dirección de la primera localidad del bloque B en una localidad reservada para ese fin en el bloque A.



SECUENCIAL



VINCULADO

1.- VECTORES Y ARREGLOS: Los vectores y arreglos son el tipo mas común de estructuras de datos. Un vector es una estructura de dato integrada por un numero fijo de componentes del mismo tipo organizados con una serie lineal simple. Un componente de un valor se selecciona dando su subíndice (tipo entero o numeración) que indica la posición del componente en la serie. Un vector también se designa con un arreglo unidimensional o arreglo bidimensional y a veces existen por mas de 3 dimensiones.

Cadena de caracteres: Es un objeto de datos compuesto por una serie de caracteres. Este tipo es muy importante en casi todos los lenguajes, debido en parte al uso de representación de datos de carácter para entrada y salida.

Operaciones:

1. CONCATENACIÓN: Es la operación de unir 2 o mas cadenas de caracteres.
2. RELACIONALES SOBRE CADENAS: Se considera que una cadena A es menor o mayor a una cadena B mediante un ordenamiento lógico grafico.
3. SELECCIÓN DE SUBCADENAS: Usando subíndices.
4. FORMATO DE E/S.

2.- OPERACIONES: Subindizacion que es la operación que selecciona un componente. Creación, destrucción, asignación y operaciones aritméticas.

3.- REGISTROS: Una estructura de datos compuesta de un numero fijo de componentes de distintos tipos: (heterogéneos) al igual que los vectores son formas de estructuras de datos lineales de longitud fija pero difieren de los aspectos.

1. Heterogeneidad de componentes.
2. Los componentes de registro se designan con nombres simbólicos en vez de indexarse.

4.- COMPONENTES DE REGISTRO: Se les suele llamar campos y los nombres de los componentes son entonces los nombres de los campos.

5.- LISTAS: Una estructura de datos compuesta de una serie ordenada se le conoce como lista. Estos son similares a los vectores y en cuanto se componen a una serie ordenada de objetos. Sin embargo las listas difieren de los vectores en varios aspectos:

1. Las listas son rara vez de longitud fija.
2. No suelen ser homogéneas.
3. Los lenguajes que usan listas declaran típicamente estos datos de manera implícita (LISP).
4. Selección de subcadena usando equipamiento de patrones.

Variaciones sobre listas: En ciertos lenguajes, se presentan variaciones sobre estructuras típicas de la lista: pilas y colas, árboles, grafos, etc.

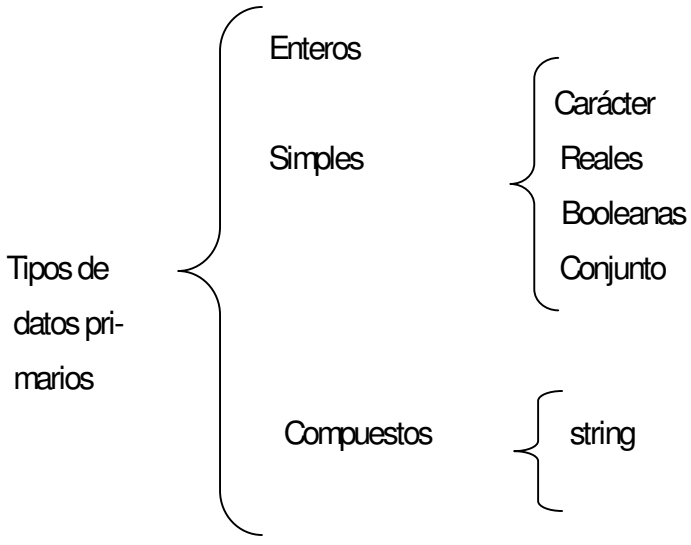
6.- APUNTADORES: Es un objeto de dato construido por el programador. Por lo común, en vez de incluir diversos tipos de objetos de datos vinculados de tamaño variable en un lenguaje de programación. Se proveen facilidades para permitir la construcción de cualquier estructura usando entre si los objetos de datos, componentes según se desee. Se necesitan varias características de lenguaje para hacer esto posible:

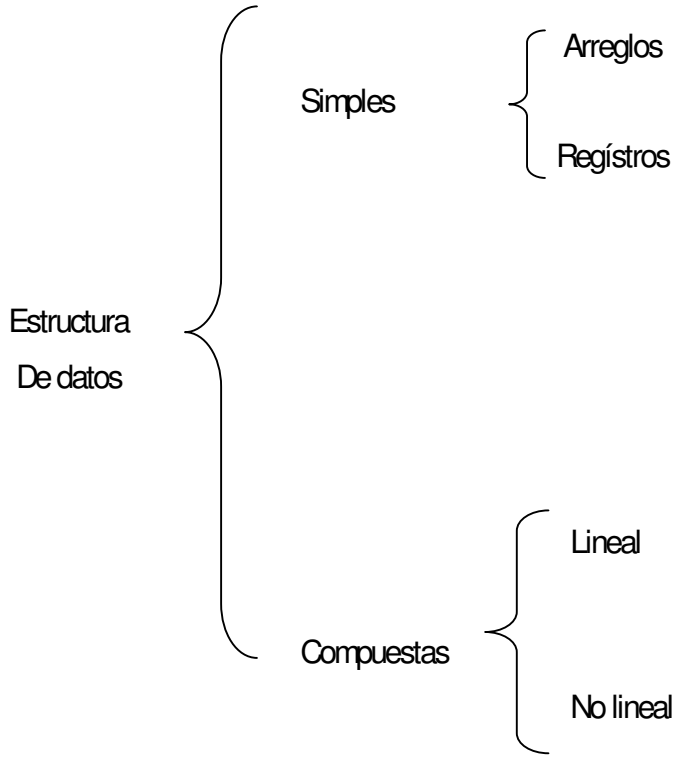
1. Un apuntador de tipo elemental de datos(referencia) que es un objeto de dato que contiene la localidad de otro objeto de datos o puede contener apuntadores nulos.
2. Una operación de creación.
3. Una operación desferencial valores de apuntador, la cual permite seguir a un apuntador hasta el objeto de datos hacia el cual indica.

7.- TIPOS DE DATOS ABSTRACTOS: Con el propósito de hacer extensivo este concepto a los datos definidos por el programador, definiremos un tipo de dato abstracto como:

1. Un conjunto de objetos de datos, ordinariamente usando una o mas definiciones de tipo.
2. Un Conjunto de operaciones abstracta sobre esos objetos de datos.
3. El encapsulamiento del todo en forma tal que el usuario de nuevo tipo excepto a través del uso de las operaciones definidas.

Especificaciones de datos elementales





CAPITULO 4

CONTROL DE SECUENCIA

Control de Secuencia Implicita y Explicita Entre Programas

Las estructuras de control de secuencias se pueden clasificar convenientemente en tres grupos:

- *Estructuras que se usan en expresiones,* y por lo tanto dentro de enunciados, puesto que las expresiones constituyen los bloques de construcción básicos para enunciados, como reglas de precedencia y parantesis.

- *Estructuras que se usan entre enunciados,* a tambien en grupos de enunciados, como en condicionales e interativos.

- *Estructuras que se usan entre subprogramas,* como llamadas de subprogramas, corrutinas y excepciones.

Esta división es por necesidad algo imprecisa. Por ejemplo, ciertos lenguajes, como LISP y APL, no tienen enunciados, solo expresiones, sin embargo aún se emplean versiones de los mecanismos usuales de control de secuencia de enunciados.

Las estructura de secuencia pueden ser implicitas y explicitas. Las *implicitas* (o por omision) son las que el lenguaje define que estan en operación, a menos que el programador las modifique a traves de alguna estructura explicita. Por ejemplo, casi todos los lenguajes definen que el orden fisico de los enunciados de un programa controla el orden en el cual los enunciados se ejecutan. Dentro de las expresiones también hay comúnmente una jerarquía de operaciones definidas por el lenguaje y que controla el orden de ejecución de las operaciones de la expresión cuando no hay paréntesis.

Las estructuras *explicitas* de control de secuencia son aquellas que el programador puede usar en forma optativa para modificar el orden implícito de las operaciones definido por el lenguaje, por ejemplo, usando paréntesis dentro de las expresiones, o enunciados GOTO y etiquetas de enunciados.

Mecanismos de control de secuencia en expresiones.

Implica 2 aspectos: el control del orden de ejecución de las operaciones , tanto primitivas como las definidas por el usuario.

Expresiones. Considérese la formula para computo de raíces de la ecuación cuadrática:

$$Raiz = -b \pm \frac{\sqrt{b2 - 4xaxc}}{2xa}$$

Esta formula en apariencia sencilla implica en realidad al menos 15 operaciones individuales. Codificada en lenguaje ensamblador o de maquina típica, requería al menos 15 instrucciones, mas aun, el programador

tendría que proporcionar almacenamiento para cada uno de ,los resultados del medio, y seguir pistas de los mismos, además de preocuparse por la optimización. Sin embargo, en un lenguaje de alto nivel como Fortran, la formula para una de las raíces se pueden modificar casi directamente como una sola expresión:

Root=(-b+sqrt(b**2-4*a*c))/(2*a)

La notación es compacta y natural, y el procesador del lenguajes, en vez del programador, se ocupa del almacenamiento temporal y la optimización.

Parece justo afirmar que la disponibilidad de expresiones en los lenguajes de alto nivel es una de las ventajas principales sobre los lenguajes maquina y ensamblador . Aunque se agrega un problema, ejemplo la expresión anterior en Fortran, ¿Es correcta?, ¿Como sabemos, por ejemplo, que la expresión indica que la substracción debe tener lugar después del computo de 4*a*c y no antes?

Los mecanismos de control de secuencia que operan para determinar el orden de las operaciones dentro de esta expresión son bastante complejos y sutiles.

CONTROL DE SECUENCIA ENTRE SUBPROGRAMA

Llamada-Regreso Simple De Subprogramación

Se acostumbra en programación a ver los programas como jerarquias. Un programa se compone de un solo programa principal; el cual, durante su ejecución, puede llamar varios subprogramas los que a su vez pueden llamar a otros subprogramas así sucesivamente a cualquier profundidad. Se espera que cada subprograma termine su ejecución en cierto punto y regrese el control al programa que lo llamó, durante la ejecución de su subprograma, la ejecución del programa que lo llamó se detiene temporalmente.

Cuando se completa la ejecución del subprograma, la ejecución del programa que lo llamó se reanuda en el punto que sigue inmediato a la llamada del subprograma. Esta estructura de control se suele explicar por la REGLA DE COPIA: el efecto del enunciado CALL (llamado del subprograma) es el mismo que se obtendría si el enumerado CALL se remplazará por una copia del cuerpo del subprograma con las sustituciones apropiadas de parámetros e identificadores en conflicto antes de la ejecución.

Vistas en esta forma, las llamadas de subprograma se pueden considerar como estructuras de control que simplemente hacen innecesario copiar grandes números de enunciados a casi idénticos que ocurren en más de un lugar en un programa.

Para la implementación de la estructura simple de llamada-regreso que se emplea para el punto de vista de regla de copia de subprograma, existen algunos supuestos implícitos en esta perspectiva y que se pueden flexibilizar para obtener estructuras más generales de control de subprogramas.

1. *Los subprogramas no pueden ser recursivos.* Un subprograma es directamente recursivo si contiene una llamada a sí mismo; es indirectamente recursivo, si llama otro subprograma que llama al subprograma original o que inicia una cadena adicional de llamadas del subprograma que conduce finalmente de regreso a una llamada al original.

   En el caso de llamadas simples no recursivas de subprograma, se puede aplicar la regla de copia durante la traducción para remplazar llamadas de subprograma por copia del cuerpo del subprograma y eliminar por completo la necesidad del subprograma independiente.

2. *Se requiere CALL explicita en la regla de copia,* que se va a aplicar cada punto de la llamada de un subprograma de indicarse en forma explícita en el programa que se va a traducir. Pero para un subprograma usado como un manejador de excepción ninguna llamada explícita debe estar presente.

3. *Los subprogramas deben ejecutarse por completo en cada llamada.*

   Se asume en forma implícita en la regla de copia, que cada subprograma se ejecuta desde su principio hasta su fin lógico cada vez que se llama. Si se llama por segunda vez el subprograma empieza la ejecución de nuevo y otra vez se ejecuta hasta su fin lógico, antes de regresar el control. Pero un subprograma usando como una corrutina continua la ejecución del punto desde su última terminación.

Implementacion de la llamada simple-regreso

La llamada simple a subprogramas presentan ordinariamente 2 formas; la llamada de función, para subprogramas que regresan valores directamente y la llamada de procedimientos o de subrutinas para subprogramas que operan atravez de efectos colaterales sobre datos compartidos.

En general las estructuras de control de secuencias entre subprogramas, se puede decir que son mecanismos para la transferencia de control de la ejecución entre programas y subprogramas.

Para entender la implementacion de la estructura de control simple, es importante construir un modelo completo para expresiones y serie de enunciados que se están ejecutando, se piensa en cada una como representada por un bloque de código ejecutable en tiempo de ejecución.

La ejecución de la expresión o serie de enunciados significa simplemente ejecución de código, usando un interprete de hardware o software. Para subprogramas se nececitan:

1.- *Existe una distinción entre una definición de subprograma y una activación de subprograma.* La definición es lo que se ve en el programa escrito. Una activación se crea cada vez que se llama un subprograma, usando la plantilla creada apartir de la definición.
2.-*Una activación se implementa como dos partes*, un segmento de código que contiene los códigos ejecutables y constantes, y un registro de activación que contiene los datos locales, parámetros y diversos elementos de datos.
3.-El segmento de código es inmodificable durante la ejecución. Es creado por el traductor y se guarda estáticamente en la memoria. Durante la ejecución se usa pero nunca se modifica. Toda activación de subprograma utiliza el mismo segmento de código.
4.- El registro de activación se crea de nuevo cada vez que se llama el sub programa, y se destruye cuando el programa regresa. En tanto el subprograma se esta ejecutando, el contenido del registro de activación esta cambiando constantemente conforme se hacen asignaciones a variables locales y otros objetos de datos.

Recursión: Es un programa que a su vez se llama asi mismo.

Subprogramas recursivos

Esta es una de las estructuras de control de secuencias mas importante en programacion. Muchos algoritmos se representan con mas naturalidad usando la recursión. En LISP, donde las estructuras de listas son la principal estructura de datos disponibles, la recurción es el mecanismo primario de control para series repetitivas de enunciados, y sustituyen la interacción de casi todos los demás lenguajes.

Si se permiten llamadas recursivas de subprogramas, un subprograma A puede llamar a cualquier otro subprograma, incluso asi mismo, un subprograma B que llama a A y asi sucecivamente, desde el punto de vista sintáctico, al escribir el programa es probable que nada cambie, pues una llamada recursiva se igual que otra. En cuanto a concepto tampoco hay dificultad siempre y cuando este clara la distinción entre una definición y una activación de subprograma. La única diferencia entre una llamada ordinaria y una recursiva esque la llamada recursiva crea una segunda activación de subprograma durante el tiempo de vida de una activación. Sila segunda activación conduce a otra llamada recursiva, entonces pueden existir 3 activaciones de manera simultanea y asi sucecivamente.

Aunque las maquinas se vuelven mas rápidas año con año el problema básico prevalece, la memoria de control y la unidad central de proceso son siempre alrededor de un orden de magnitud mas rápida que la memoria

principal mas grande. Por consiguiente es frecuente que una computadora consuma gran parte del tiempo simplemente esperando que el hardware introduzca datos de la memoria principal mayor en la memoria de control, que es mas rápida.

Se han intentadado dos enfoques para resolver este problema. Se han proyectado software que produce un iso mas eficiente del hardware. Esto hace posible que el hardware trabaje con mas eficiencia ejecutando otro programa cuando un programa esta bloqueando esperando datos (excepciones y corrutinas).

Otra alternativa es crear hardware mas eficaz. El uso de una memoria cache entre la memoria principal y la memoria de control proporciona una aceleración efectiva de la memoria de control.

También esta combinado la naturaleza del computo. Las estaciones de trabajo han sustituido a las computadoras independientes.

Excepciones

Durante la ejecución de un programa suelen ocurrir sucesos o condiciones que se podrían conciderar como "excepcionales" en vez de continuar por la ejecución normal del programa, se necesita llamar un subprograma para llevar a cabo cierto procesamiento especial. Esto incluye:

a)***Condiciones de error.-*** obligan a procesar un error, como exceder la capacidad de una operación aritmética o hacer referencia a un elemento de arreglo con un subindice fuera de limites.
b)***Condiciones inpredecibles.-*** Que surgen durante la ejecución normal del programa, como la producción de encabezamientos especiales de salida al final de una pagina de impresora a un identificador de final de archivo en un archivo de entrada.
c)***Rastreo y monitoreo.-*** Durante la prueba del programa, como la salida de rastreo de impresión durante la prueba de programas cuando el valor de la variable cambia.

Es mas sencillo relajar el requisito de que los subprogramas se deben de llamar por medio de llamadas explícitas y suminitrar una forma de invocar un subprograma cuando ocurre una condición o suceso particular. A una condición o suceso de esta clase se le suele llamar excepción o señal, y al subprograma que lleva a cabo el procesamiento especial se le conoce como manejador de excepciones. A la acción de advertir la excepción , interrumpir la ejecución del programa y transferir el control al manejador de excepciones se le describe como plantear la excepción. Ejemplo, en lenguaje Ada existe una clase de excepciones llamadas checks (Verificaciones) o condiciones que requieren que se ejecute código.

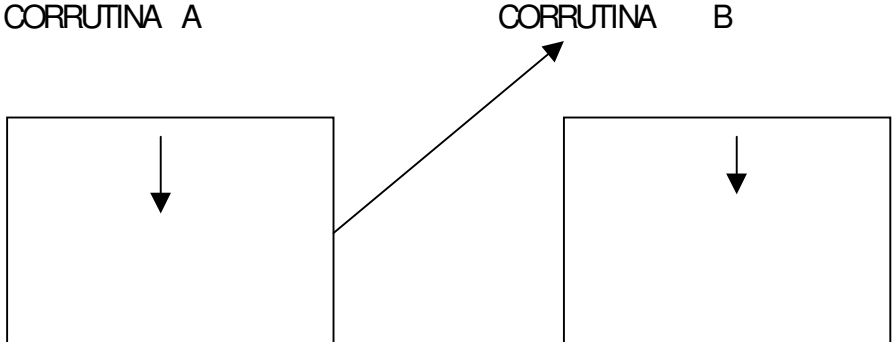
Estructura tipica de planteamiento de excepciones en lenguaje Ada

```
Procedure sub 15
        Valor_De_Dato_Malo: excepción;
              _otras declaraciones para sub
Begin
        - enunciados para procesamientos normal de sub excepción
        when Valor_De_Dato_Malo        ⇨
              -manejador para valores de datos malos
when constraint_error        ⇨
              -manejador para excepciones preferidas error_de _restricción
when others        ⇨
              -manejador para todas las demás
end;
```

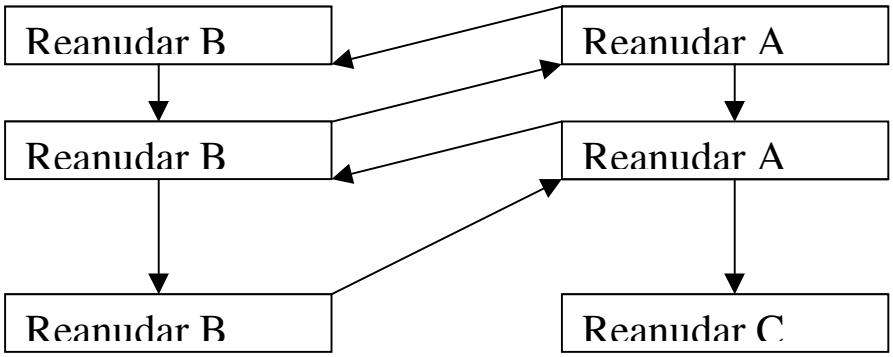
Los lenguajes que incluyan manejadores de excepciones: Ada, ML, C++.

Corrutinas

Son mecanismos para permitir que los subprogramas regresen a su programa de llamada antes de que se complete su ejecución. Estos subprogramas se llaman corrutinas. Cuando una corrutina recibe el control de otros subprogramas, se ejecuta parcialmente y luego se suspende cuando regresa el control de otros subprogramas. En un punto posterior el programa que llama puede reanudar la ejecución de la corrutina a partir del punto en el cual se susprendio previamente la ejecución.



Transferencias del control entre subrutinas



Las excepciones son llamadas implícitas a subprogramas.

Las corrutinas son llamadas explícitas a subprogramas.

Subprogramas planificados

El concepto de planificación de subprogramas es consecuencia del relajamiento del supuesto que la ejecución de un subprogramas siempre se debe iniciar de inmediato cuando se llama. Se puede pensar que un enunciado de llamadas de subprograma especifica el subprogramada llamado debe ponerse en ejecución de inmediato, sin completar la ejecución del programa de llamada. La finalización de la ejecución del programa de llamado se vuelve a planificar para que ocurra de inmediato al terminar el subprograma.

Las técnicas de reorganización mas comunes son:

1.- planifica los subprogramas para ejecutarse antes o después de otros. Ejemplo, call B after A.

2.- Planificar los programas para ejecutarse mediante una expresión booleana. Ejemplo, Call B when x=5.

3.- Planificar los subprogramas en base en una escala de tiempo simulada. Ejemplo, Call B at time = 25 or call B at time = hora actual + 10.

4.- Planificar los subprogramas de acuerdo con una designación de prioridad. Ejemplo, call B whit priority 7.

La planificación generalizada de subprogramas es una característica de los lenguajes de programación proyectados para simulación de sistemas, como GPSS, SIMCRIPT, SIMULA.

Programacion en paralelo y ejecución no secuencial

De manera mas general, varios subprogramas se podrían estar ejecutando simultaneamente. Cuando hay un solo orden de ejecución, el programa se escribe como un programa secuencial, porque la ejecución de sus subprogramas tienen lugar en un orden predefinido. De otra manera el programa se describe como un programa concurrente o en paralelo. Todo subprograma que se pueda ejecutar de manera simultanea con otros se conoce como tarea (o aveces proceso).

Los sistemas de computadoras capaces de ejecutar varios pogramas de forma simultanea son ahora bastante comunes. Un sistema de multiprocesadores tiene varias unidades centrales de procesamiento (CPU) que comparten una memoria común.

Un sistema de computadoras distribuidas o en paralelo tienen varias computadoras, cada una con su propia memoria y CPU conectadas en una red en la cual se puede comunicar unas con las otras. En estas clases de sistemas se pueden ejecutar muchas tareas de manera simultanea.

Incluso en una sola computadora, suele ser útil proyectar un programa de manera que se componga de muchas tareas independientes que se ejecutan en forma concurrente en la computadora virtual, no obstante que la computadora real solo se puede estar ejecutando una en un momento dado.

Los sistemas operativos que manejan multiprogramación y tiempo compartido proporcionan este tipo de ejecución concurrente para programas de usuarios individuales.

ENUNCIADOS BASICOS

Asignación a Objetos de Datos.

Los cambios al estado de computo de valores a objetos de datos es el mecanismo mas importante que afecta el estado de un computo que realiza un programa.

Una forma de esta clase de enunciados es la *asignación,* cuyo propósito primordial es atribuir al valor `n´ de un objeto de datos, (es decir, a su localidad de memoria) el valor `r´(el valor del objeto de dato) de cierta expresión. La asignación es una operación fundamental definida para todos los tipos elementales de datos. La sintaxis para una asignación especifica varía ampliamente:

PASCAL, ADDA, MODULA-2 a:=10; a;=b;

C → a=10; a=b; a++;a+1=2;

→ ←

→

APL a b

LISP (SETQ A B)

Otras Formas de Control de Secuencia a Nivel de Enunciados.

Se distinguen por lo común tres formas principales de control de secuencia a nivel de enunciados, estas son implícitas:

a) ***COMPOSICION.*** Los enunciados se pueden disponer en una serie textual, de modo que se ejecuten en orden, siempre que se ejecute la estructura mayor del programa que contiene la serie.

b) ***ALTERNANCIA.*** (condicionales) Dos series de enunciados pueden formar alternativas de modo que se ejecuten una u otra serie, pero no ambas, siempre que se ejecuta la estructura mayor del programa que contiene la serie.

c) ***ITERACIÓN.*** Una serie de enunciados se puede ejecutar en forma repetida 0 o más veces (0 significa que la ejecución se puede omitir del todo), siempre que se ejecute la estructura mayor del programa que contiene la serie.

Algunas formas explícitas de control de secuencia para enunciados son: BREAK, GOTO, EXIT():

MODELO DE IMPLEMENTACION DE SUBPROGRAMAS

Uno de los modelos más sencillos de implementación de subprogramas, es costoso en cuanto a precio y almacenamiento, con el fin de aumentar la rapidez de ejecución. Esta implementacion consiste en asignar almacenamiento para un solo registro de activación para cada subprograma de manera estatica, como una extensión del segmento de código en este modelo más simple (que usa en muchas implementaciones FORTRAN, COBOL) la ejecución del programa global, se inicia en un subprograma segmento de código y un registro de activación para cada subprograma y el programa principal, ya presente en la memoria la ejecución avanza sin asignación dinámica de almacenamiento. Cuando de llama un subprograma en su lugar, en un mismo registro de activación se usa en forma repetida.

REPRESENTACION DE LAS ESTRUCTURAS DE ARBOL

Hasta aquí, se ha considerado a las expresiones como entidades únicas y se ha pasado por alta la sintaxis y semántica reales necesarias para la evaluación de una expresión dada.

El mecanismo básico de control de secuencia en expresiones es la composición funcional: sé específico una operación y sus operandos; los operandos pueden ser constantes, objetos de datos u otras operaciones, cuyos operandos a su vez, pueden ser constantes, objetos de datos o aún otras operaciones, a cualquier profundidad. La composición contiene a una expresión, la estructura característica de un árbol representa la operación principal, los nodos entre la raíz y las hojas representan operaciones de nivel intermedio, y las hojas representan referencias de datos (o constantes).

NOTACIÓN

- **INFIJA.** Operadores entre los operandos (a +b)
- **PREFIJA.** Operadores antes de los operandos (+ab).
- **POSTFIJA.** Operadores después de los operandos.(ab+)

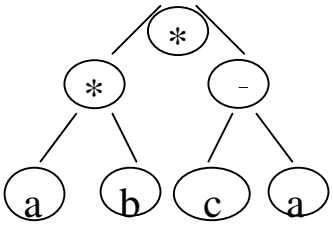
Las expresiones son un dispositivo poderoso y natural para representar series de operaciones, aunque plantes nuevos problemas. Si bien puede ser tedioso escribir largas series de instrucciones en lenguaje máquina, al menos el programador entiende con claridad el orden en el que las operaciones se van a ejecutar. Los mecanismos de control de secuencia que operan para determinar el orden de las operaciones dentro de esta expresión son, de hecho, bastante complejos sutiles.

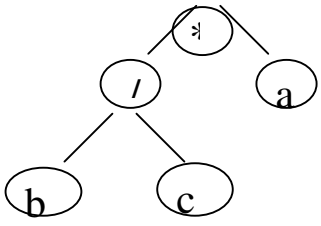
Ejemplo:

-b

Árbol de representación:

Ejemplos: **(a*b)*(c-a)** **a*(b/c)**





Semantica para expresiones

Cada una de las 3 notaciones tiene ciertos atributos que son útiles en el diseño de lenguajes de programación. Difieren principalmente en cuanto ala manera de calcular el valor para cada expresión.

1) **NOTACIÓN INFIJA.**

La notación infija es más adecuada para operaciones binarias (diádicas).

En esta notación, el símbolo del operador se escribe entre los dos operandos. Puesto que la notación infija para las operaciones aritméticas, relacionales y lógicas básicas es de uso tan común en las matemáticas ordinarias, la notación para estas operaciones ha sido adoptada ampliamente en lenguajes de programación.

Semántica Infija.- Aunque la relación infija es común, su uso en un lenguaje de programación, conduce a varios programas:

a) Puesto que la notación infija es adecuada solo para operadores binarios, un LP no puede usar solo notación infija, si no que combinar por necesidad notaciones infija y prefija (o postfija). La mezcla que hace la traducción sea más compleja.Los operadores binarios y las llamadas de función en argumentos múltiples deben ser excepcionales a la propiedad infija general.

b) Cuando aparece más de un operador infijo en una expresión, la notación es inherentemente ambigua al menos que se utilice paréntesis.

2) ***NOTACIÓN POSTFIJA***

La notación postfija es similar a la notación prefija, excepto que el símbolo de operación sigue a la lista de operandos. Ejemplo: (a+b)*(c-a), en postfija sería: ab+ca-*

Semántica postfija.- Puesto que en la notación postfija el operador sigue a sus operandos, cuando se examina a un operador sus operandos ya han sido evaluados. Por consiguiente, la evaluación de una expresión postfija P, usando una vez más una pila, ahora tiene lugar como sigue:

1.- Si el elemento sigue de p es un operando, colocarlo en la pila.

2.-Si el elemento siguiente de p es un operador n-ario, entonces sus n argumentos deben de ser los n elementos superiores en la pila.

La estructura de evaluación es directa y fácil de implementar. De hecho, es la base del código generador de expresiones en muchos traductores. Durante la traducción, la sintaxis de las expresiones a menudo se vuelven posfijas.

3) ***NOTACION PREFIJA***

Con la notación prefija, se puede evaluar cada expresión, en un solo examen de la misma. Es necesario conocer, el numero de argumentos para cada operación. Es por la razón que se necesitan símbolos especiales para la substracción didáctica, (-) y la menos unaria(-), para distinguir cuales la operación deseada.

Ademas del ahorro de paréntesis, la notación prefija tiene ciertas ventajas:

1.- La llamada de función usual ya esta escrita en la notación prefija.

2.-La notación prefija se puede usar para representar operaciones con cualquier numero de operandos. Solo es necesario aprender una regla sintáctica para escribir cualquier expresión. Ejemplo la notación polaca de cambridge de LISP.

3.-Es relativamente fácil decodificar, la notación; por esta razón se consigue sin dificultad la traducción de expresiones prefijas a secuencias de código simple mediante la implementación de pilas.

PROGRAMACION ESTRUCTURADA

Este termino se usa para diseño de programas que hacen énfasis en:

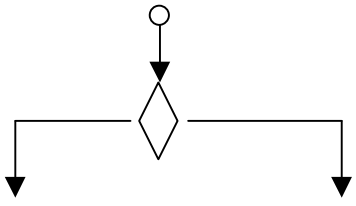
1.- El diseño jerárquico de estructuras de programas usando solo las formas simples de control (composición, alternancia, iteracion)

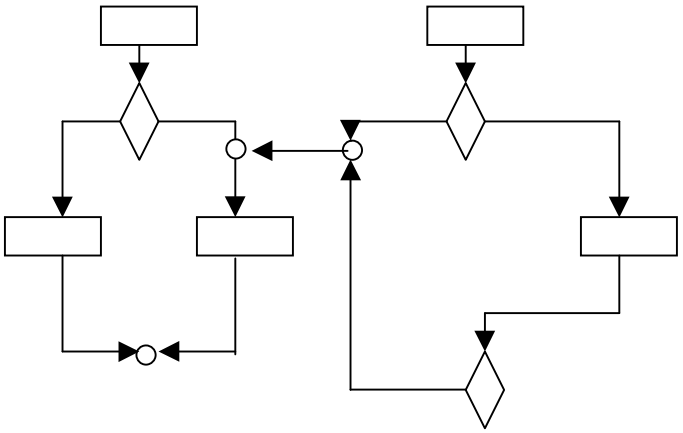
2.-La representación directa del diseño jerárquico en el texto del programa usando los enunciados de control estructurados, (if,case,for, etc).

3.- El texto del programa en el cual el orden textual de los enunciados corresponden al orden de ejecución.

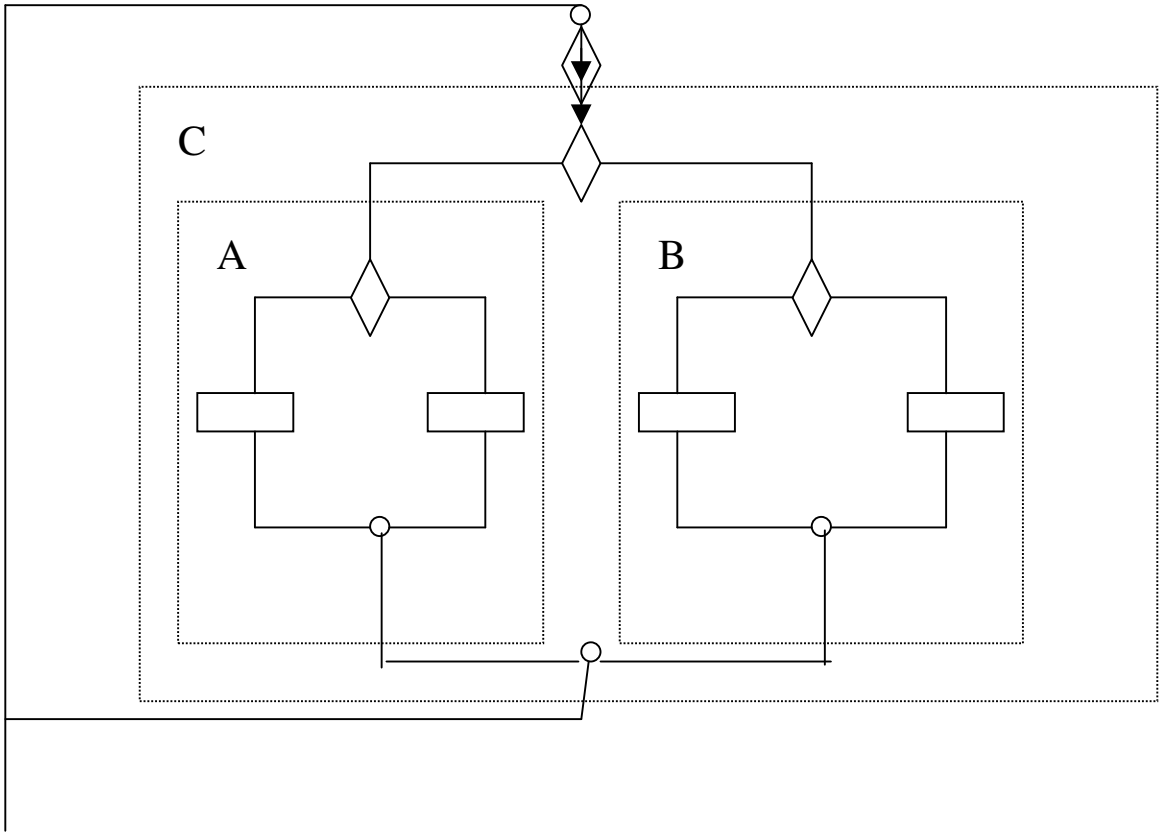
4.-El uso del grupo de enunciados de propósito único aun cuando sean necesario copiar los enunciados.

Cuando un programa se escribe siguiendo estos principios de programación estructurada, por lo común es mucho mas fácil de entender, depurar, codificar, que es correcto y mas adelante codificarlo y verificarlo otra vez.

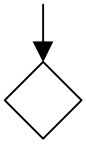


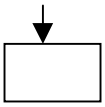


Programa Primo



Programa Compuesto





DECISIÓN ENUCIADO REUNION

Programa primo

Es un programa propio que no se puede subdividir en programas propios mas pequeños.

Programa propio

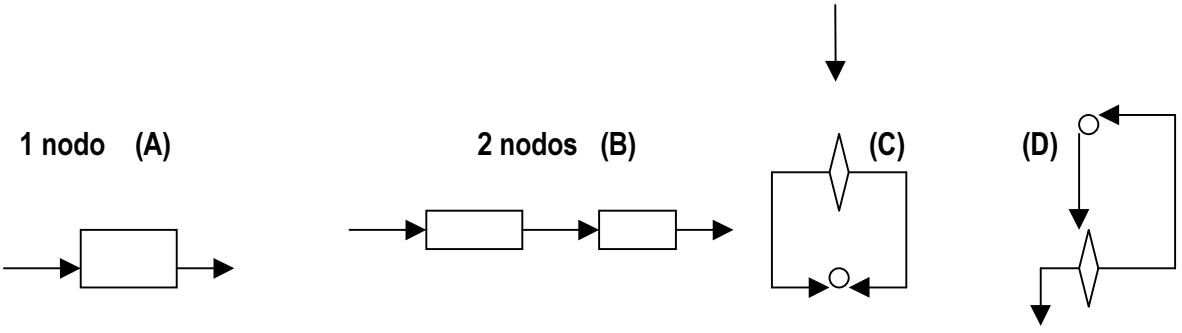
Todo diagrama de flujo consiste en tres componentes (nodo enunciado, nodo reunión y nodo decisión) cuya estructura de control tiene las siguientes características:

1) Un solo arco de entrada.
2) Un solo arco de salida.
3) Una ruta de arco de entrada de cada nodo y de cada nodo al arco de entrada.

Programa compuesto

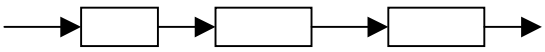
Se define como un programa propio que no es primo, al remplazar cada componente de un programa primo a un programa propio por un nodo de enunciado (ejemplo, retrasando los componentes primos A y B por nodos de enunciados se puede repetir el proceso hasta que se consigue una descomposición prima de cualquier programa propio.

Enumeración de programas primos:

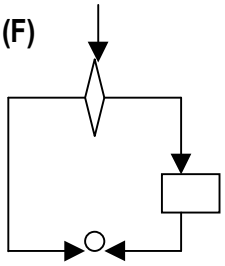


3 nodos

(E)

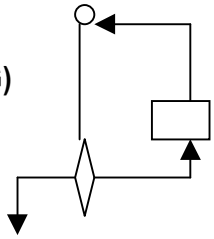


(F)



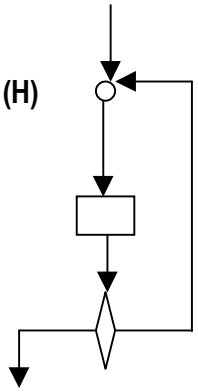
If-then

(G)



Do while

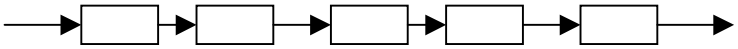
(H)



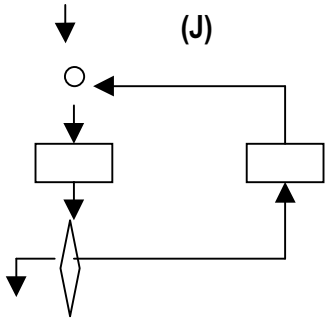
Repeat-until

4 nodos

(I)

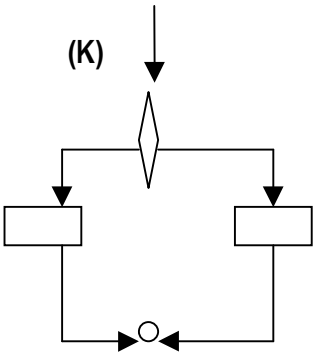


(J)



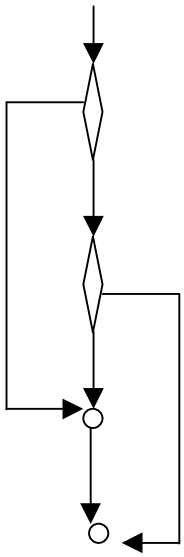
Do while do

(K)

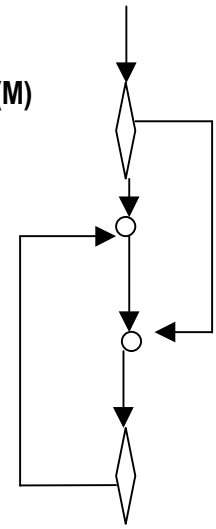


If then else

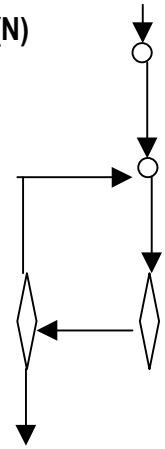
(L)



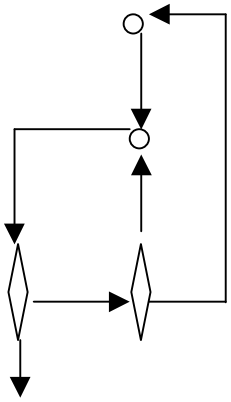
(M)



(N)



(O)



CAPITULO 5

CONTROL DE DATOS

La característica de control de datos de un LP son aquellas partes que tienen que ver con la accesibilidad de datos en diferentes puntos durante la ejecución del programa. Los mecanismos de control de secuencial del capitulo anterior, proporciona los medios para coordinar el orden en que las operaciones se invocan. Una vez que

se alcanza una operación durante la ejecución, se le debe suministrar los datos sobre los cuales va a operar. Las características de control de datos determinan como se puede suministrar datos a cada operación, y como se puede guardar el resultado de una operación y recuperarlo para uso posterior como operando por parte de una operación subsiguiente.

NOMBRE Y AMBIENTES DE REFERENCIA

Existen básicamente, solo dos maneras de hacer que un objeto de datos este disponible como operando para una operación. Estos son:

TRANSMISION DIRECTA. Un objeto de datos computado en un punto como resultado de una operación se puede transmitir directamente a otra operación operando, ejemplo:

X:=Y+2*Z.

El resultado de la multiplicación de 2*Z se transmite directamente a la operación de adición con y mediante el almacenamiento temporal sin que se les de un nombre a este objeto de datos.

REFERENCIA A TRAVES DE UN OBJETO DE DATOS DE UN NOMBRE. A un objeto de datos se le puede dar el nombre cuando se crea y el nombre se puede usar para asignarlo como un operando de una operación.

La transmisión directa se usa para control de datos dentro de expresiones (aritméticas), pero casi todo el control de datos fuera de expresiones implica el uso de nombres y la referencia de nombres.

AMBIENTE LOCAL DE REFERENCIA. El conjunto de asociaciones creadas al entrar a un subprograma y que representan parámetros formales, variables locales y subprogramas definidos solo dentro de ese subprograma conforma el ambiente local de referencia de esa activación del subprograma. El significado de una referencia aun nombre en el ambiente local se puede determinar sin salir de la activación del subprograma.

AMBIENTE NO LOCAL DE REFERENCIA. El conjunto se asociaciones para identificadores que se pueden usar dentro de un subprograma pero que no se crean al entrar a él se conoce como ambiente no local de referencia.

AMBIENTE GLOBAL DE REFERENCIA. Si las asociaciones creadas al inicio de la ejecución del programa principal están disponibles para usarse en un subprograma, entonces estas asociaciones forman el ambiente global de referencia de ese subprograma, el ambiente global es parte del ambiente no local.

AMBIENTE PREDEFINIDO DE REFERENCIA. Ciertos identificadores tiene una identificación predefinida, la cual se define directamente en la definición del lenguaje. Cualquier programa o subprograma puede usar estas asociaciones sin crearlas en forma explícita.

Todos los ambientes de referencia pasan por un cierto grado de *visibilidad*, se dice que una asociación para un identificador es visible dentro de un subprograma si es parte del ambiente de referencia para ese subprograma. Una asociación que existe pero que no forma parte del ambiente de referencia del subprograma actualmente en ejecución se dice que esta oculta de ese subprograma. Con frecuencia una asociación esta oculta cuando se entra a un subprograma que se define un identificador que ya está en uso en otra parte del programa.

AMBIENTES DE REFERENCIA

Concepto. Cada subprograma tiene un conjunto de asociaciones de identificador disponibles para su uso al hacer referencias durante su ejecución. Este conjunto de asociaciones de identificador se conoce como el ambiente de referencia de subprograma. El ambiente de referencia de un subprograma es ordinariamente invariable, durante su ejecución.

Se establece cuando se crea la activación del subprograma y permanece sin cambio durante el cambio de vida de la activación. Los valores contenidos en los diversos objetos de datos pueden cambiar, pero no así las asociaciones de nombres con objetos de datos y subprogramas.

Datos Locales y Ambientes Locales de Referencia

Ambiente local de un subprograma Q se compone de los diversos identificadores declarados en la cabeza del mismo. Los nombres de variables, nombres de parámetros normales, de subprogramas o constantes.
Para ambientes locales es fácil hacer congruentes la regla de alcance estático, el compilador mantiene simplemente una tabla de las declaraciones locales para identificadores que aparecen en la cabeza de Q, y mientras compila el cuerpo de Q recurre primero a esa tabla siempre que se requiere la declaración de un identificador.
Ej:

```
Procedure Sub(x:integer)
Var Y:real;
Z:array[1..3]of real
```

```
Procedure sub2
        Begin
        ……
        ……
        End(sub2)
Begin
  ……
end sub;
```

Tabla ambiente local

Nombre	Tipo	Contenido valor L
X	Entero	Parámetro de valor
Y	Real	Valor real
Z	Real	Arreglo local/descriptar[1..3]
Sub2	procedimental	Apuntador a segmento de código

La retención y la eliminación son los enfoques distintos de la semántica de ambientes locales. C, Pascal, Ada, Lisp, APL y Snobol4, Utilizan el enfoque de eliminación.

Las variables locales no conservan sus valores antiguos entre llamadas sucesivas de un subprograma. Cobol y muchas versiones de FORTRAN emplean el enfoque de retención las variables conservan sus valores antiguos entre llamdas. PL/I, algol proporcionan ambas opciones, cada variable individual se puede tratar de manera diferente.

ELEMENTOS DE PROGRAMA QUE PUEDEN TENER NOMBRE

EJECUCIÓN

1. Nombre de variables.
2. Nombre de parámetros.
3. Nombre de subprogramas

1. Nombre para tipos definidos. (usuarios)
2. Nombre para constantes definidas

TRADUCCIÓN 3. Nombre etiquetas de enunciados

4. Nombre de excepciones

5. Nombre para operaciones primitivas.

6. Nombre para constantes de literales

ASOCIACIONES Y AMBIENTES DE REFERENCIA

El control de datos se ocupa en gran parte del enlace de identificadores (nombres simples) a objeto de datos y subprogramas particulares, esta clase de enlaces se conocen como una asociación y se pueden representar como una pareja compuesta de identificador y su objeto de datos o subprograma asociado.

Alcance Dinámico: cada asociación tiene un alcance dinámico, el cuál es la par de la ejecución del programa durante la que esa asociación existe como parte de un ambiente de referencia. Así, el alcance dinámico de una asociación se compone del conjunto de activación de subprograma dentro de las cuales es visible.

Operación de Referencia*:* es una operación de la siguiente signatura:

Ref_op:id_x ambiente-de-referencia-→ objeto de datos o subprograma

Donde ref_op, dado un identificador y un ambiente de referencia, encuentra la asociación apropiada para ese identificador en el ambiente y regresa el objeto de datos o la definición de subprograma asociado.

Referencias locales, no locales y globales: una referencia a un identificador es una referencia local si la operación de referencia encuentra la asociación en el ambiente local; es una referencia no local o global si la asociación se encuentra en el ambiente no local o global, respectivamente.

Pseudonimo para objetos de dato: durante su tiempo de vida un objeto de dato puede tener más de un nombre, ejemplo: puede haber varias asociaciones en diferentes ambientes de referencia, donde cada una proporciona un nombre distinto para el objeto de datos ejemplo: cuando un objeto de datos, se transmite por referencia a un subprograma como parámetro, se puede hacer referencia a él a través de un nombre de parametro formal en el subprograma, y también conserva su nombre original en el programa que llama.

Alternativamente, un objeto de datos puede convertirse en componente, de varios objetos de datos a través de vínculos de operador, y tener así varios nombres compuestos a través de los cuales se puede tener acceso a el. Los nombres múltiples para el mismo objeto de da datos son posibles de diversas maneras en casi todos los lenguajes de programación.

Cuando un objeto de datos es visible a través de mas de un nombre (simple o compuesto) en un solo ambiente de referencia cada uno de los nombres se conoce como pseudonimo del objeto de datos. Cuando un objeto de datos tiene múltiples nombres, pero un único nombre en cada ambiente de referencia en el cual aparece, no surgen problemas, sin embargo, la capacidad para hacer referencia al mismo objeto de datos usando nombres distintos dentro del mismo ambiente de referencia plantea serios problemas tanto para el usuario como para el inplementador del lenguaje.

ALCANCE ESTATICO Y DINAMICO

El alcance dinámico de una asociación para un identificador, es el conjunto de activaciones de subprogramas en las cuales la asociación es visible durante la ejecución. El alcance dinámico de una asociación siempre incluye la activación de subprograma, en la cuál esa asociación se crea como parte del ambiente local. Aunque también puede ser visible como una asociación no local en otras activaciones de subprogramas.
Una regla de alcance dinámico define que el alcance dinámico de una asociación en términos del curso dinámico de ejecución del subprograma. Ejemplo: una regla afirma que el alcance de una asociación creada durante una activación del subprograma "P" incluye no solo esa activación, si no también cualquier activación de un subprograma llamado por "P" o por un subprograma llamado por "P" y así sucesivamente, a menos que esa ultima activación se subprograma defina una nueva asociación local para el identificador que oculte la asociación original.

Alcance Estático

Cuando se examina la forma escrita de un programa, se advierte que la asociación entre referencias a identificadores y declaraciones particulares o definiciones del significado de esos identificadores también es un problema.
Cada declaración u potra definición de un identificador dentro del texto del programa tiene un cierto alcance llamado alcance estático.

Una declaración crea una asociación de texto del programa entre un identificador y cierta información acerca del objeto de datos o subprograma que va a ser nombrado por ese identificador durante la ejecución del programa.
El alcance estático de una declaración es la parte del texto del programa donde un buzo del identificador es una referencia a esa declaración particular del identificador.
Una regla de alcance estático en pascal se usa para especificar de una referencia a una variable "x" en un subprograma "p" se refiere a la declaración "x" al inicio del subprograma "u" cuya declaración, contiene la declaración de "p" y así sucesivamente. Las reglas de alcance estático relacionan referencias con declaraciones de nombres en el texto del programa; las reglas de alcance dinámica relacionan referencias con asociaciones para nombres durante la ejecución del programa. Es claro que las reglas de alcance estático y dinámico deben de ser congruentes.

ESTRUCTURA DE BLOQUES

Tal como se encuentra en lenguaje estructurado en bloques, como Pascal, PL/I, Ada, tienen una estructura de programa característica y un conjunto asociado de reglas de alcance estático. En un lenguaje estructurado de bloques, cada programa o subprograma esta organizado como un conjunto de bloques anidados.
La característica principal de un bloque es que introduce un nuevo ambiente local de referencia. Un bloque se inicio con un conjunto de declaraciones para nombres (declaraciones de variables, definiciones de tipo, de constantes) enseguida de un conjunto de enunciados n los cuales se puede hacer referencia, esos nombres se consideran un bloque como equivalente a una declaración de subprograma, aunque la definición es alta el bloque varia de un lenguaje a otro. Las declaraciones que hay en un bloque definen su ambiente local de referencia que es invariable en la ejecución.
En lenguaje C hay una estructura de bloques, pero existe únicamente dentro de un subprograma. Esto proporciona la capacidad de suministrar nombres no locales sin los gastos de estructura para la activación de subprograma.

REGLAS DE ALCANCE ESTATICO EN PROGRAMACION ESTRUCTURADA DE BLOQUES

1. Declaraciones que están en la cabeza de cada bloque definen el ambiente local de referencia para el bloque. Cualquier referencia seguido a un identificador dentro del cuerpo del bloque(sin incluir sus bloques anidados). Se considera una referencia ala declaración local para el identificador si existe uno.
2. Se hace referencia a un identificador dentro del cuerpo del bloque y no existe una declaración local, entonces la referencia se considera una referencia a una declaración dentro del bloque que encierra inmediatamente al primer bloque. Si no existe una declaración ahí, entonces se refiere a una declaración en el bloque que encierra inmediatamente a ese bloque y así sucesivamente. Si se alcanza el bloque más exterior antes de lograr una declaración entonces la referencia es a la declaración dentro de ese bloque más externo. Por ultimo, si no se encuentra una declaración ahí, se usa la declaración del ambiente predefinido del lenguaje en su caso, o la referencia se toma como un error.
3. Si un bloque contiene otra definición de bloque entonces cualesquier declaraciones locales dentro del bloque interior o de los bloques que el mismo contenga están ocultas por completo del bloque exterior, y no se pueden hacer referencias a las mismas desde el.
4. Un bloque puede tener nombre (por lo común cuando representa un subprograma con nombre). El nombre del bloque se vuelve parte del ambiente local de referencia del bloque influyente.

DATOS COMPARTIDOS

Compartición de datos

- * **Paso de parametros**
  - * Llamadas valor-constante
  - * Llamadas referencia
  - * Llamadas valor
  - * Llamadas resultado
  - * Llamadas nombre
  - * Llamadasvalor- resultado
- * **Ambientes comunes**
  - package (Ada)
  - extern (C)
- ***Exposicion estatica y dinamica**
  - (alcance)

Tipos de parametros

- a) formales
- b) Reales

Datos compartidos en Subprogramas

Un objeto de datos que es estrictamente local suelen compartirse entre varios subprogramas para que las operaciones de cada uno de los subprogramas puedan utilizar los datos. Un objeto de datos se puede transmitir como un parámetro explícito, aunque en algunas ocasiones el uso de estos es engorroso.
Otra forma de compartición de datos es que un conjunto de subprogramas hagan uso de una tabla de datos común. Cada subprograma requiere acceso a la tabla este uso común de datos se basa ordinariamente en la compartición de asociaciones de identificador. Se usan 4 enfoques básicos en cuanto ambientes no locales en los lenguajes de programación:

1. Ambientes comunes explícitos
2. Ambientes no locales implícitos de alcance dinámico
3. Ambientes no locales implícitos de alcance estático
4. Herencia

PARAMETROS Y TRANSMISION DE PARAMETROS

Los parámetros y resultados transmitidos de manera explícita constituyen el método alternativo principal de compartición de datos. El contraste de uso de ambiente no locales de referencia, donde la compartición se consigue haciendo visibles ciertos nombres no locales para un subprograma, los objetos de datos transmitidos como parámetros se transmiten sin un nombre anexo. En el subprograma receptor, a cada objeto de dato se le da un nuevo nombre local a través del cual se puede hacer referencia a el. La transmisión de parámetros resulta mas útil cuando a un subprograma se le van a entregar datos distintos para procesar cada vez que se le llama. Mientras que un ambiente no local es mas apropiado cuando se usan los mismos objetos de datos en cada llamada.

Parámetros Reales y Formales

Un parámetro formal es una clase particular de objetos de datos local dentro de un subprograma. La definición de subprogramas enumera los nombres y declaraciones para parámetros formales.

Un parámetro real es un objeto de datos que se comparte con el subprograma de llamada, este puede ser objeto de datos local perteneciente al que llama, puede ser un parámetro formal del que llama, un objeto no local visible para el que llama, o puede ser un resultado regresado por una función invocada por el que la llama.

ESTABLECIMIENTOS DE CORRESPONDENCIA

Cuando se llama un subprograma de un alista de parámetros reales, se puede establecer una correspondencia con los formales enumerados en la definición. Dos métodos son utilizados:

1.- Correspondencia de posición: esta se establece apareando parámetros reales y formales con base en sus posiciones respectivas en las listas que los contienen. Casi todos los lenguajes utilizan este tipo de correspondencia.

1. **Correspondencia por nombre explicito:** en Ada y en algunos otros lenguajes, el parámetro formal que se va a aparear con el real se puede nombrar de manera explícita en el enunciado de llamada.

Metodos para transimitir Parámetros

Cuando un subprograma transfiere el control a otro, debe haber una asociación de parámetro real del subprograma que llama con el parámetro formal del programa llamado. Suelen emplearse dos enfoques: Se puede evaluar el parámetro real y pasar ese valor al formal, o el objeto de dato real puede pasarse al parámetro formal.

Se han ideado en general varios métodos para paso de parámetros:

1. **Llamada por nombre:** este modelo de transmisión contempla una llamada de subprograma como una substitución del cuerpo completo del subprograma. Con esta interpretación, cada parámetro formal representa la evaluación real del parámetro real particular. Precisamente como si se hicieran las sustituciones reales. Los parámetros se transmiten sin evaluar, y el subprograma llamado determina cuando. Ej.: ALGOL, Pascal y C.

2. **Llamada por referencia:** es tal vez el mecanismo más común de transmisión de parámetros. Transmitir un objeto de datos como parámetros de llamado por referencia significa que se pone a disposición del subprograma un apuntador a la localidad del objeto de datos. El objeto de datos mismo no cambia de posición en la memoria. Ej.: FORTRAN, Pascal y C.

3. **Llamada por valor**: si un parámetro se transmite por valor, el valor del parámetro real se pasa al formal llamado.

4. **Llamada por valor resultado:** en este método el parámetro formal es una variable(objeto de datos) local del mismo tipo de datos que el parámetro real. El valor del parámetro real se copia en el objeto de datos del parámetro formal en el momento de la llamada. Durante la ejecución del subprograma cuando este termina, el contenido final del objeto de datos el parámetro formal se copia en el objeto de datos del parámetro real. De esta forma el parámetro real conserva su valor original hasta la terminación del subprograma, que es cuando se asigna su nuevo valor como resultado ejm : ALGOLW.

5. **Llamada por valor constante:** si un parámetro se transmite por valor constante entonces no se permite cambio alguno en el valor del parámetro formal durante la ejecución y tampoco se puede transmitir a otro subprograma excepto como parámetro de valor constante.

6. **Llamada por resultado:** se usa para transmitir un resultado de regreso desde un subprograma. El valor inicial del objeto de datos de parámetro real no importa y el subprograma llamado no lo puede usar. El parámetro formal es una variable local sin valor inicial. Al terminar el subprograma el valor final del parámetro formal se asigna como el nuevo valor del parámetro real.

Con base en este análisis debe quedar claro que con la llamada por referencia se tiene un sinónimo del parámetro real, en tanto que la llamada por valor no se tiene esta clase de referencia, por lo tanto el parámetro formal no tiene acceso a modificar el valor del parámetro real.

Ambientes comunes { Tablas, Variables

Ambientes Comunes:

Un ambiente común establecido de manera explícita para compartir objeto de datos es el método más sencillo para compartir datos. Aun conjunto de objetos de datos que se van a compartir entre un conjunto de subprogramas se le asigna almacenamiento en un bloque con nombre por separado. Cada subprograma contiene una declaración, que nombra explícitamente el bloque compartido. Los objetos de datos dentro del bloque son entonces visibles dentro del subprograma y se puede hacer referencia a ellos por nombre en la forma usual. Un bloque compartido de este tipo se conoce por diversos nombres: bloque COMMON (común) en FORTRAN; en ADA es una forma de paquete; en C las variables individuales marcadas como EXTERN(externas)se comparten de esta manera. Las clases de C++ y Smalltalk proporcionan esta característica, pero no es ese el propósito normal de las clases. El término de ambiente común es apropiado aquí.

Especificación: un ambiente común es idéntico a un ambiente local para un subprograma, excepto que no es parte de un subprograma individual. Puede contener definiciones de variables, constantes y tipos, pero no subprogramas o parámetros formales. Ej: la especificación de PACKAGE(paquete) en ADA.

```
PACKAGE Tablas_compartidas is
        Tamaño_Tab:constant integer:=100;
        Type Tabla is array(1..Tamaño_Tab)of Real;
        Tabla1,Tabla2:Tabla;
        Entrada_Act:integer range 1..Tamaño_tab;
End
```

El ejemplo anterior de especificación define un tipo, una constante, 2 tablas y una variable entera que juntos representan un objeto de datos que varios subprogramas necesitan. La definición del paquete se da fuera de los subprogramas que utilizan las variables.

Si un subprograma P requiere acceso al ambiente común definido por este paquete, entonces se incluye un enunciado WITH explícito entre las declaraciones de P:

```
With Tablas_compartidas;
```

Dentro del cuerpo de P, cualquier nombre contenido en el paquete se puede usar ahora directamente, como si fuera parte del ambiente local para P. Usaremos el nombre calificado: nombre de paquete. Nombre de variable para hacer referencia a estos nombres. Así pues podemos escribir en P:

```
Tablas_compartidas_Tabla1(Tablas_compartidas.Entrada_Act):=Tablas_compartidas.Tabla2(Tablas_compartidas
.Entrada_Act)+1
```

El nombre del paquete se debe usar como prefijo en cada nombre, ya que un subprograma puede usar muchos paquetes, algunos de los cuales puede declarar el mismo nombre(también se puede "colocarlas" con USE).

Implementación:

En C y FORTRAN, cada subprograma que utiliza el ambiente común debe incluir también declaraciones para cada variable compartida(como si se definiera un registro)para que el compilador conozca las declaraciones pertinentes, no obstante que el ambiente común también se declara en otro lugar.

CAPITULO 6

ADMINISTRACIÓN DE LA MEMORIA

GESTION DE ALMACENAMIENTO

La gestión de almacenamiento para datos es una de las preocupaciones fundamentales del programador, del implementador de lenguajes y del diseñador de lenguajes.

En tanto que todos los diseños de lenguajes permiten ordinariamente el uso de ciertas técnicas de gestión de almacenamiento, los detalles de los mecanismos, y su representación en hardware y software, son tarea del implementador.

ELEMENTOS PRINCIPALES EN TIEMPO DE EJECUCIÓN QUE REQUIEREN ALMACENAMIENTO

El programador tiende haber la gestión de almacenamiento en gran medida en términos de almacenamiento de datos y programas traducidos. Sin embargo, la gestión de almacenamiento en tiempo de ejecución abarca muchas otras áreas.

Examinemos los principales elementos de programas y datos que requieren de almacenamiento durante la ejecución del programa.

- **Segmentos de código para programas de usuarios traducidos:** En cualquier sistema debe asignar un bloque importante de almacenamiento para guardar los segmentos de código que representan la forma traducida de programas de usuario, sin tomar en cuenta si los programas son interpretados por el hardware o software.

- **Programas de sistema en tiempo de ejecución:** Otro bloque considerable de almacenamiento que se debe de asignar a programas de sistemas que apoyan la ejecución de los programas de sistema que apoyan la ejecución de los programas de usuarios. Estos pueden ir desde simple rutinas de biblioteca hasta interpretes o traductores de software.

- **Estructuras de datos y constantes definidas por el usuario:** Debe asignarse espacio para datos de usuario destinado a todas las estructuras de datos declaradas en programas de usuario o creadas por ellos, incluso constantes.

- **Puntos de retorno en subprogramas:** Los subprogramas tienen la propiedad de que se les puede invocar desde distintos puntos de un programa. Por consiguiente, se debe asignar almacenamiento a la información de control de secuencia generada internamente, como los puntos de retorno de subprogramas planificados.

- **Entornos de referencia:** El almacenamiento de entorno de referencia (asociaciones de identificadores) durante la ejecución puede requerir espacio considerable, como, por ejemplo la lista A de LISP.

- **Temporal en evaluación de expresiones:** La evaluación de expresiones requiere el uso de almacenamiento temporal definido por el sistema para los resultados intermedios de evaluación.

- **Temporales en transmisión de parámetros:** Cuando se llama un subprograma, se debe evaluar una lista de parámetros reales y guardar los valores resultantes en almacenamiento temporal hasta que se completa la evaluación de toda la lista. Cuando la evaluación de un parámetro puede requerir evaluación de llamada recursivas de función, se puede necesitar una cantidad potencialmente ilimitada de almacenamiento, como en la evaluación de expresiones.

- **Buffers de entrada-salida:** Sirven como área de almacenamiento temporal donde se guardan datos entre el momento de la transferencia física efectiva de datos a o desde el almacenamiento externo y las operaciones de entrada y salidas iniciadas por el programa.

- **Datos diversos del sistema:** En casi todas las implementaciones de lenguajes, se requiere almacenamiento para diversos datos de sistema: tablas, información de situación para entrada-salida y diversos fragmentos de información de estado.

Además de los datos y elementos de programas que requieren almacenamiento, también resulta instructivo examinar las diversas operaciones que pueden requerir asignación o liberación de almacenamiento. Las operaciones principales son:

- **Operaciones de llamadas y retorno de subprogramas:** La asignación de almacenamiento para un registro de activación de subprogramas, el entorno local de referencia y otros datos al llamar un subprograma suele ser la operación principal que requiere asignación de almacenamiento. La ejecución de una operación de retorno de un subprograma requiere por lo común liberar el espacio asignado durante la llamada.
- **Operaciones de creación y destrucción de estructuras de datos:** Si el lenguaje suministra operaciones que permiten crear nuevas estructuras de datos en puntos arbitrarios durante la ejecución del programa (en vez de sólo al entrar al subprograma), entonces estas operaciones requieren ordinariamente asignación de almacenamiento por separado del asignado al entrar al subprograma. Ejemplo la operación NEW en PASCAL y la función MALLOC en C.
- **Operaciones de inserción y eliminación de componentes:** Si el lenguaje suministra operaciones que insertan y eliminan componentes de estructuras de datos, se puede requerir asignación y liberación de almacenamiento para implementar estas operaciones. Ejemplo las operaciones de lista de ML y LISP que insertan un componente en la lista.

FASES DE LA GESTION DE ALMACENAMIENTO

Es conveniente identificar 3 aspectos básicos de la gestión de almacenamiento:

1) **ASIGNACIÓN INICIAL:** Al inicio de la ejecución, cada segmento de almacenamiento puede estar ya sea asignado o libre. Si esta libre inicialmente, esta disponible para asignarse de manera dinámica conforme avanza la ejecución.

2) **RECUPERACIÓN:** El almacenamiento que ha sido asignado y usado, y que posteriormente queda disponible, debe ser recuperado por el gestor de almacenamiento para volver a usarlo. La recuperación puede ser muy simple, como en la reubicación de un apuntador de pila, o muy compleja, como en la recolección de basura.

3) **COMPACTACION Y NUEVOS USOS:** El almacenamiento recuperado puede estar listo de inmediato para volver a usarse, o puede requerirse una compactación para construir bloques grandes de almacenamiento libre a partir de fragmentos pequeños.

GESTION DE ALMACENAMIENTO ESTATICO

Tiene lugar durante la traducción y permanece fija a lo largo de la ejecución. Esta partición no requiere software de gestión de almacenamiento en tiempo de ejecución y, desde luego, no hay que preocuparse por la recuperación y nuevo uso. Además es incompatible con llamadas recursivas de subprogramas.

La asignación estática de almacenamiento, esta tiene lugar durante la traducción, es eficiente porque no se gasta tiempo ni espera para la gestión de almacenamiento durante la ejecución. Ejemplo de lenguajes: FORTRAN y COBOL.

GESTION DE ALMACENAMIENTO CON BASE EN PILAS

Es la técnica mas simple de almacenamiento en tiempo de ejecución, El almacenamiento libre al principio de la ejecución se establece como un bloque secuencial en la memoria. Conforme se asigna almacenamiento, el mismo se toma de localidades secuenciales en este bloque de pila a partir de un extremo. El almacenamiento se debe liberar en el orden inverso de asignación para que el bloque de almacenamiento que esta siendo liberado se encuentre siempre en el tope de la pila.

Un solo apuntador de pila es todo lo que requiere para controlarla. La compactación ocurre en forma automática como parte de la liberación de almacenamiento. Su implementación es por registro de activación, con analizador de expresiones, etc.

Ejemplos de lenguajes: PASCAL, LISP, ADA, C y PROLOG.

GESTION DE ALMACENAMIENTO EN MONTÍCULOS ELEMENTOS DE TAMAÑO FIJO

Un *montículo* es un bloque de almacenamiento dentro del cual se asignan o se liberan segmentos de alguna manera relativamente no estructurada. En este caso los problemas de asignación, recuperación, compactación y nuevo uso del almacenamiento pueden ser graves. No existe una técnica única de gestión de almacenamiento en montículos, mas bien, una colección de técnicas para manejar diversos aspectos de la administración de esta memoria.

La necesidad de almacenamiento en montículos surge cuando un lenguaje permite asignar y liberar almacenamiento en puntos arbitrarios durante la ejecución del programa.

Esta partición de memoria las técnicas de gestión se puede simplificar bastante. La compactación no representa problema pues todos los elementos disponibles son del mismo tamaño.

GESTIÓN DEL ALMACENAMIENTO EN MONTÍCULOS ELEMENTOS DE TAMAÑO VARIABLE

La gestión de almacenamiento en montículo donde se asignan y recuperan elementos de tamaño variable es más difícil que con elementos de tamaño fijo. Se presentan elementos de tamaño variable en múltiples situaciones.

Las dificultades principales con elementos de tamaño variable tienen que ver con el nuevo uso del espacio recuperado. Incluso si se recuperan dos bloques de espacio de palabras en el montículo, puede ser imposible satisfacer una solicitud posterior de un bloque de 6 palabras.

Un apuntador de montículo es adecuado para la asignación inicial. Cuando se solicita un bloque de n palabras, el apuntador de montículo se hace avanzar a n y el valor original del apuntador de montículo es devuelto como apuntador al elemento recién asignado.

Dos posibilidades de nuevo uso:

1. Usar la lista de espacios libres para asignación, examinando la lista en busca del bloque de tamaño apropiado y devolviendo el espacio sobrante después de la asignación.
2. Compactar el espacio libre moviendo todos los elementos activos a un extremo del montículo.

RECUPERACIÓN CON BLOQUES DE TAMAÑO VARIABLE

La recolección de basura tiene como antes, con una fase de marcado seguido de una fase de recolección, la solución más simple consiste en mantener junto con el bit de recolección en la primera palabra de cada bloque, activo o no, un indicador de longitud entero que especifique la longitud del bloque.

COMPACTACIÓN Y EL PROBLEMA DE FRAGMENTACIÓN DE MEMORIA

Se comienza con un solo bloque de espacio libre. Conforme el cómputo avanza, este bloque se fragmenta de manera gradual en trozos más pequeños a través de la asignación, recuperación y uso. Cierta compactación de bloques libres a bloque más grandes, la ejecución se detendrá por falta de almacenamiento libre más pronto de lo necesario.

CONCLUSIONES

Este trabajo recepcional nos sirvió para comprender, mejor los aspectos con los que algunos lenguajes de programación están diseñados, cosa que nos permitió conocerlos y comprender mejor las construcciones sintácticas de los lenguajes con los que vamos a programar y por lo tanto a trabajar con ello en nuestra profesión.

Esto también nos ayudo a saber cuales son los lenguajes que son mas convenientes a utilizar dependiendo del problema que la aplicación tiene, y por lo tanto saber que lenguaje utilizar para cada cosa o problema real.

También tenemos una visión comparativa entre lenguajes similares ó clasificados dentro del mismo tipo de aplicación, que nos permite saber cuales son mas eficientes entre uno y otro.

BIBLIOGRAFIA:

- **TERRENCE W. PRATT.** Lenguajes de Programación.
- **Ravi Sethi** . Lenguajes de Programación.
- **Tucker .** Lenguajes de Programación.

Autor: Luis Carlos L. Cruz. Email lopezluiscarlos@yahoo.com
Webmaster: Alfonso Guillèn V. Email: kenpopoder@yahoo.com

Visita la Pàgina de Luis Carlos en: http://conalep.itgo.com

Libro. Lenguajes de Programaciòn I.