(12) Ej.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

"CARACTERISTICAS GENERALES DE LOS LENGUAJES DE PROGRAMACION Y ESTUDIO DE ALGUNOS LENGUAJES"

T E S I S

Que para obtener el título de:

M A T E M A T I C O

p r e s e n t a

AMPARO LOPEZ GAONA

México, D. F. 1981

Indice

Introducción

En la actualidad, es cada vez más frecuente que el hombre utilice la computadora como una herramienta para resolver cualquier problema que se le presente . Sin embargo, para poder hacer uso de esta poderosa herramienta, es necesario que exista una comunicación entre el hombre y la máquina; la cual se realiza por medio de los lenguajes de programación.

Los lenguajes de programación han evolucionado en el transcurso del tiempo, así, los primeros lenguajes que existieron fueron los lenguajes de máquina, que se fueron desarrollando hasta llegar a los lenguajes de alto nivel con que actualmente se cuenta; pero esta evolución no termina aquí, ya que se continuan desarrollando trabajos e investigaciones en esta área.

Uno de los objetivos de esta tesis, es proporcionar las bases que permitan conocer y comprender las principales características que presentan los lenguajes de programación, para que de esta forma se pueda analizar cualquier lenguaje y en determinado momento facilitar la elección de un lenguaje apropiado y conveniente para una aplicación particular, o bien poder simular algún mecanismo o estructura que no proporcione el lenguaje que se esté empleando.

Otro objetivo, es ayudar a tener un conocimiento más amplio de lenguajes como FORTRAN IV, COBOL ANS, ALGOL de la Burroughs, PASCAL y SNOBOL4

Como en la elaboración de un programa, se debe decidir cuales estructuras de datos, operaciones, estructuras para controlar la secuencia, etcétera, son necesarias para llegar a su solución, es importante y de gran utilidad tener un conocimiento profundo del lenguaje con el que se trabaje para así poder utilizarlo de la forma más conveniente para obtener programas con alto grado de eficiencia.

En este trabajo se hablará brevemente del desarrollo histórico de los lenguajes de alto nivel (capítulo 1); así como de su definición formal (capítulo 2). Además se hará un análisis de las características principales que deben presentar los lenguajes de programación, como son datos, operaciones, control de la secuencia, control de datos y manejo de almacenamiento (capítulo 3); a fin de obtenr las bases necesarias para poder analizar lenguajes de alto nivel como FORTRAN IV (capítulo 4), COBOL ANS (capítulo 5), ALGOL Burroughs (capítulo 6), PASCAL (capítulo 7) y SNOBOL4 (capítulo 8). Finalmente se hablarás de la sintaxis de los lenguajes de programación analizados con anterioridad y se mostrarán ejemplos de programas que presenten algunas de las características propias de ese lenguaje (apéndice A).

1 Historia de los lenguajes de programación

En este capítulo se describirá, brevemente y en orden cronológico el desarrollo de los lenguajes de programación. No se intenta cubrir el desarrollo de todos y cada uno de ellos (ya que su número en 1972 era de 170), sino que se mencionan únicamente los lenguajes más importantes desde el punto de vista de que fueron la base para el desarrollo de los lenguajes con que se trabaja en la actualidad.

El período comprendido entre 1952 y 1956 fue de agrupamiento preliminar y de intentos por entender los conceptos y las limitaciones de los lenguajes de programación.

De todos los lenguajes desarrollados en este tiempo sólo FORTRAN (usado para cálculos numéricos) y APT (para control de herramientas de máquina) siguen siendo utilizados, a pesar de que ambos han sufrido numerosas revisiones.

El primer encuentro para discutir los lenguajes de alto nivel --o codificaciones automáticas, como se conocían en ese entonces- se efectuó en el "Franklin Institute" en 1956. En ese tiempo ningún sistema era ampliamente usado; FORTRAN no había sido comercializado (en efecto, no fue discutido en este encuentro); no obstante, fue presentado en 1957 en la Western Joint Computer Conference. Los otros sistemas fueron usados sólo por sus desarrolladores.

En el encuentro del Instituto Franklin se presentaron y discutieron los siguientes lenguajes:

B-0 (FLOWMATIC). Primer lenguaje diseñado para procesar datos destinados a la solución de problemas administrativos. Planeado e implementado únicamente en la UNIVAC I.

PRINT I. Pseudo-código realmente poderoso, importante porque fue idóneo para resolver problemas matemáticos en una máquina diseñada para procesar datos administrativos, llamada IBM 705.

OMNICODE. Lenguaje ensamblador en espíritu y formato pero con operaciones poderosas para cálculos científicos y comerciales. Fue diseñado para la IBM 650 y 702.

IT. Lenguaje para problemas matemáticos que fue torpe en su notación, por tener un conjunto de caracteres muy limitado en su máquina, la IBM 650. Es significativo porque fue un intento de usar máquinas pequeñas.

Matrix Compiler. Lenguaje de alto nivel que contenía operaciones para hacer cálculos matriciales. Diseñado por la UNIVAC I. Es significativo porque fue uno de los primeros lenguajes para áreas de aplicación especializadas (considerando la manipulación de matrices como un área especializada)

NCR 304. No es un lenguaje, sino una máquina. Es significativa

por haber sido el primer intento por desarrollar una computadora que hiciera innecesaria la codificación automática, ya que su código tenía un nivel bastante alto. (aunque la máquina fue, aparentemente, un éxito, ésta no eliminó a los lenguajes de alto nivel).

Los dos años más significativos en la historia de los lenguajes de programación fueron 1958 y 1959. Los siguientes acontecimientos tuvieron lugar en ese período:

1. El desarrollo y publicación de IAL (International Algebraic Language), el cual fue conocido como ALGOL 1958.

2. El desarrollo de tres lenguajes basados en las especificaciones de IAL llamados NELIAC, MAD y CLIP (este último sirvió para la formación de JOVIAL). NELIAC, MAD y JOVIAL fueron usados al menos hasta 1971 en aplicaciones militares.

3. La presentación en 1959 del formalismo de J. Backus para describir ALGOL. Esta fue la base para muchos de los trabajos teóricos hechos en lenguajes de programación, desde entonces

4. La formación en mayo de 1959, del "CODASYL Short Range --- Committe" (más tarde llamado comité de COBOL) y el término de las especificaciones en diciembre de ese mismo año.

5. El desarrollo y disponibilidad de las especificaciones de lenguaje, para AIMACO, "Commercial Translator" y FACT.

6. Los primeros trabajos para desarrollar LISP, en 1959.

7. La primera implementación de COMIT.

8. Los primeros trabajos en JOVIAL, en 1959.

9. La disponibilidad de una versión corrida de IPL-V, a principios de 1958, en la IBM 650; una nueva versión estuvo operando en la IBM 704, a fines del verano en 1959.

10. El desarrollo de una segunda versión de APT, llamada APT II para la IBM 704.

11. El desarrollo de algunos lenguajes para áreas especializadas; por ejemplo: DYANA (1958), DYNAMO (1959), primeros trabajos en AED (1959).

La década de 1960 a 1970 está considerada como de madurez en el campo de los lenguajes de programación. Durante este tiempo se fue generalizando el uso de lenguajes de alto nivel y de sistemas de programación que -- usan lenguajes de alto nivel.

Los mejores lenguajes fueron ALGOL, COBOL y PL/I de los cuales sólo los dos últimos fueron ampliamente usados en Estados Unidos. Mientras que ALGOL 68 fue definido, su implementación comenzó alrededor de 1970.

Una versión preliminar de PASCAL fue planeada hacia fines de -- 1968, siguiendo la filosofía de ALGOL-60 y ALGOL-W. Después de un desarrollo extensivo, un primer compilador llegó a operar en 1970 y su publicación fue hasta el siguiente año.

El advenimiento de tiempo-compartido trajo un sinnúmero de lenguajes en línea empezando con JOSS y más tarde seguido por BASIC que es ampli amente usado. Cada uno tenía muchos imitadores y extensiones. APL/360 estuvo

disponible hasta 1960, llegando a ser popular entre ciertos grupos específicos.

El desarrollo de los lenguajes de alto nivel para usarse en manipulación de fórmulas fue iniciado por FORMAC y FORMULA ALGOL, aunque sólo el primero fue utilizado considerablemente. El procesamiento de cuerdas y comparación de patrones se hizo popular con el advenimiento de SNOBOL.

Los lenguajes de simulación GPSS y SIMSCRIPT fueron la herramienta disponible para la mayoría de los usuarios y también fueron las bases para desarrollos de otros lenguajes de simulación.

Uno de los desarrollos prácticos más importantes, aunque desperdiciado por varios teoricistas, fue el desarrollo de los estándares oficiales para FORTRAN y COBOL, y el comienzo de la estandarización para PL/I.

A partir de 1979 se han desarrollado gran cantidad de lenguajes para aplicaciones especializadas, contándose entre los lenguajes más ampliamente utilizados los siguientes:

ILIAD. Desarrollado en la General Motors para usarse en la programación de procesos industriales. Este lenguaje fue diseñado para fomentar, en los programadores, el hábito de escribir programas bien estructurados, particularmente para resolver problemas de tiempo real.

FORTH. Lenguaje de alto nivel, desarrollado para aplicaciones

con bases de datos y aplicaciones comerciales en general, su importancia radica en el hecho de que puede trabajar con paquetes de subrutinas escritas en otros lenguajes.

ADA. Lenguaje para aplicaciones numéricas, aplicaciones de programación de sistemas y aplicaciones con requerimientos de tiempo real y ejecución concurrente. Fue desarrollado a iniciativa del Departamente de Defensa de los Estados Unidos. Su propósito inicial fue el de desarrollar un lenguaje de programación para ayudar a las aplicaciones militares. Se empezó a desarrollar en 1975.

MATHSY. Lenguaje de alto nivel, interactivo, para aplicaciones en matemáticas y graficación, desarrollado a principios de 1980.

Otros lenguajes que han aparecido en escena recientemente son SIMULA 67, CONCURRENT PASCAL y MODULA además de versiones experimentales de CLU. Estos lenguajes no sólo permiten que el programador defina sus tipos de datos (como SNOBOL y PASCAL), sino que también admiten tipos de datos abstractos. Estas facilidades dan gran claridad y modulariad en los programas.

2 Definición formal de los lenguajes

Un lenguaje puede ser considerado como un conjunto de oraciones o fórmulas - cuerdas de símbolos - , con estructuras bien definidas y usualmente también con significado.

Las reglas que especifican las construcciones válidas de un lenguaje son conocidas como sintaxis, por ejemplo , en el lenguaje algebráico - una expresión sintácticamente válida sería A + B y una expresión inválida sería A B + .

La asignación de un significado o interpretación de los símbolos y fórmulas es la semántica del lenguaje, por ejemplo, cuando se dice que 2X es la suma del valor de X dos veces se está refiriendo a la semántica del - álgebra.

El primer paso en la definición formal de un lenguaje es establecer el universo de discusión o alfabeto, es decir, es necesario especificar - los objetos que pertenecen al alfabeto. Los elementos del alfabeto son los - símbolos; los símbolos se concatenan para formar cuerdas que pueden pertenecer o no al lenguaje.

Un alfabeto T es un conjunto finito de símbolos llamados símbolos terminales. Algunos ejemplos de alfabetos son: el alfabeto binario, $\{ 0, 1 \}$ y el alfabeto griego $\{ \alpha, \beta, \gamma, \ldots, \omega \}$.

Una oración, también llamada cuerda o fórmula, es una cuerda de longitud finita compuesta de símbolos del alfabeto. La oración vacía, ε ,es aquella que no tiene símbolos.

Si T es un alfabeto, entonces T^* es el conjunto de todas las oraciones compuestas por símbolos de T, incluyendo a la oración vacía y T^+ se define como sigue: $T^+ = T^* - \{\varepsilon\}$.

Un lenguaje L, es cualquier conjunto de oraciones dentro de un alfabeto T, esto es $L \subset T^*$

Una vez definido el universo de discusión, es necesario hacer una representación del lenguaje. Si el lenguaje consta de un número finito de oraciones, la solución es escribir una lista de todas las oraciones válidas; pero para la mayoría de los lenguajes no es posible limitar el conjunto de oraciones válidas, obviamente, lenguajes de esta naturaleza no pueden especificarse con una enumeración de sus oraciones.

Existen algunos métodos para especificaciones finitas de lenguajes aunque estos sean infinitos. Un método es, usar un sistema generador llamado gramática, donde cada oración se construye usando un conjunto de reglas bien definidas. Un segundo método es, aplicar un proceso en el cual una vez presentado como entrada un conjunto arbitrario de oraciones, se detendrá y contestará: "sí, la oración es legal" o bien "no, ésta no es legal" y después de un número finito de cálculos determina si la oración pertenece o no al lenguaje.

Una gramática G es un cuarteto G = (N, T, S, P), donde:

N = conjunto finito de símbolos no terminales.

T = conjunto finito de símbolos terminales

S = un símbolo de N, llamado símbolo inicial

P = conjunto de producciones, reglas $\alpha \longrightarrow \beta$ donde $\alpha, \beta \in (N \cup T)^*$

$\alpha \neq \varepsilon$ y $N \cap T = \emptyset$

Los símbolos terminales son los símbolos del alfabeto T. Los símbolos no-terminales son un conjunto N de símbolos que no pertenecen a T y que representan estados intermedios en el proceso de generación de oraciones. El símbolo inicial es un símbolo no-terminal a partir del cual todas las oraciones serán derivadas.

Una producción es una regla de transformación de la forma $\alpha \rightarrow \beta$ donde α es una oración de V^+ y β es una oración en V^* (V = T∪N).

El proceso de generación se efectúa aplicando, en cada paso una - producción, con este proceso se obtienen las oraciones; el proceso se detiene cuando la cuerda consta únicamente de símbolos terminales.

Ejemplos de gramáticas formales:

1) Considerese una gramática G = (N , T, S, P) donde:

N = { A,B,S } ; T = { 0,1 } y

P = { S $\longrightarrow$ AB (1)

A $\longrightarrow$ 0A (2)

$A \longrightarrow 0$ (3)

$B \longrightarrow B1$ (4)

$B \longrightarrow 1$ (5)

Aplicando las reglas se tiene:

$S \xrightarrow{1} AB \xrightarrow{3} 0B \xrightarrow{5} 01$

$S \xrightarrow{1} AB \xrightarrow{2} 0AB \xrightarrow{3} 00B \xrightarrow{5} 001$

$S \xrightarrow{1} AB \xrightarrow{2} 0AB \xrightarrow{3} 00B \xrightarrow{4} 0011$

$S \xrightarrow{1} AB \xrightarrow{3} 0B \xrightarrow{4} 0B1 \xrightarrow{5} 011$

etc.

$\therefore\ L(G) = \{0^n 1^m \ ,\ n \geq 1,\ m \geq 1\}$

NOTA: El número arriba de la flecha indica el número de producción usada.

Ejemplo 2: Sea $G = (N, T, S, P)$ donde $N = \{S, B, C\}$; $T = \{a, b, c\}$ y P consta de las siguientes producciones:

1.- $S \longrightarrow aSBC$

2.- $S \longrightarrow aBC$

3.- $CB \longrightarrow BC$

4.- $aB \longrightarrow ab$

5.- $bB \longrightarrow bb$

6.- $bC \longrightarrow bc$

7.- $cC \longrightarrow cc$

i) Si se empieza con la producción 2:

$S \xrightarrow{2} aBC \xrightarrow{4} abC \xrightarrow{6} abc$

ii) Si se empieza con la producción 1 :

$S \xrightarrow{1} aSBC \xrightarrow{2} aaBCBC \xrightarrow{4} aabCBC \xrightarrow{3} aabBCC \xrightarrow{6} aabbcC \xrightarrow{7} aabbcc$

etc. $\therefore\ L(G) = \{a^n b^n c^n \ ,\ \forall\ n \geq 1\}$

2.1 Clasificación de las Gramáticas

Las gramáticas se clasifican en cuatro categorías anidadas de - - acuerdo a ciertas restricciones en la naturaleza de sus producciones, estas gramáticas producen cuatro tipos de lenguajes.

El tipo más general de gramática es aquella que no impone ningún tipo de restricción a las producciones y se llama gramática tipo 0.

Una gramática tipo 1 ó de contexto sensitivo es aquella donde toda producción $\alpha \rightarrow \beta$ de P, satisface que $|\alpha| \leq |\beta|$ donde $|\alpha|$ es el número de símbolos de α . Como ejemplo de esta gramática se tiene la siguiente:

$G = (\{A, B, S\}, \{a, b, c\}, S, P)$ donde

$P = \{S \rightarrow Abc$

$Ab \rightarrow aAbB$

$Bb \rightarrow bB$

$Bc \rightarrow bcc$

$A \rightarrow a \}$

Esta gramática genera cuerdas de la forma $a^n b^n c^n \ \forall \ n \geq 1$.

Una gramática tipo 2 ó libre de contexto es aquella donde toda - producción $\alpha \rightarrow \beta$ de P satisface que $|\alpha| = 1$ y $|\beta| \geq 1$. A continuación se muestra un ejemplo de gramática libre de contexto.

Sea $G = (\{A, B, S\}, \{a, b\}, S, P)$ donde

P tiene las siguientes producciones:

$P = \{ S \rightarrow aB \quad A \rightarrow bAA$

$S \rightarrow bA \quad B \rightarrow b$

$A \rightarrow a \quad B \rightarrow bS$

$A \rightarrow aS \quad B \rightarrow aBB \}$

El lenguaje L (G) es el conjunto de palabras que tienen igual número de a's y b's .

Una gramática tipo 3 ó regular es aquella donde toda producción de P es de la forma $A \rightarrow aB$ o bien, $A \rightarrow a$, donde $A, B \in N$, ($A \neq B$) y $a \in T$. Ejemplo:

$G = (\{ S, A, B \} , \{ 0, 1 \} , S, P)$

$p = \{ S \rightarrow oA \quad B \rightarrow 1B$

$S \rightarrow 1B \quad B \rightarrow 1$

$A \rightarrow oA \quad B \rightarrow o$

$A \rightarrow oS \quad S \rightarrow o$

$A \rightarrow 1B \quad \}$

2.2 Notación sintáctica BNF

BNF (Forma Normal de Backus) es una notación para escribir gramáticas, que comúnmente se usa para especificar la sintaxis de los lenguajes de programación. Una definición BNF de la sintaxis de un lenguaje es un conjunto de reglas, fórmulas o "ecuaciones". Con este conjunto de ecuaciones, en --

ocasiones llamadas "producciones", es posible generar todos los programas válidos del lenguaje. El mismo conjunto de reglas puede ser usado para reconocer cuando un texto dado es válido dentro de ese lenguaje.

Cada regla de sintaxis requiere de una definición. El elemento que se va a definir aparece en el lado izquierdo y la definición aparece en el lado derecho. Por ejemplo, si se desea definir la forma impresa de un entero con signo, pero sólo si éste es negativo se tendría:

<entero impreso> ::= <signo impreso><entero sin signo>

<signo impreso> ::= - | <vacío>

<vacío> ::=

Estas reglas se leerían como sigue: "un entero impreso se define como un signo impreso seguido de un entero sin signo";"un signo impreso se define como "-" (menos), o como vacío (nada)" y "vacío se define como espacio en blanco, es decir, nada". Una frase encerrada en paréntesis angulares se refiere a la clase de objetos, esto es, por ejemplo: <entero sin signo> significa "cualquier miembro de la clase de objetos que puede considerarse como enter sin signo". El símbolo ::= significa "se define como", y el símbolo | significa "o bien".

Una razón del poderío de la notación BNF, es que permite definiciones sintácticas recursivas, por ejemplo:

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

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

Una representación BNF de una serie de objetos puede hacerse escribiendo entre llaves ({ }) el elemento repetitivo. El subíndice de la llave indica el mínimo número de elementos y si se escribe un número superior indica el máximo número de elementos de la serie. Así para definir un identificador como una letra seguida de 0 a 5 letras o dígitos se escribiría:

$$\langle \text{identificador} \rangle ::= \langle \text{letra} \rangle \left\{ \langle \text{letra} \rangle \mid \langle \text{dígito} \rangle \right\}_0^5$$

3 Características de los lenguajes

Cualquier programa, independientemente del lenguaje usado, puede ser visto como un conjunto de operaciones que serán aplicadas a ciertos datos en determinado orden. Las diferencias básicas entre los lenguajes existentes están en los tipos de datos que permiten usar, en el tipo de operaciones disponibles, y en los mecanismos proporcionados para controlar el flujo de las -- operaciones que son aplicadas a los datos. Estas tres áreas - datos, operaciones y control - forman la base de este capítulo y también la base para el análisis de los lenguajes que se efectuará en capítulos posteriores.

3.1 Datos

Los datos que existen durante la ejecución de un programa caen -- dentro de dos categorías : datos definidos por el programador y datos definidos por el sistema.

Los datos definidos por el programador son aquellos que define y manipula explícitamente el programador en un programa; por ejemplo: números, arreglos y archivos.

Los datos definidos por el sistema son aquellos que la implementación del lenguaje construye para "uso doméstico" durante la ejecución de un programa; por ejemplo, stacks de puntos de regreso de subprogramas, descrip-

tores de estructuras de datos, listas de espacio libre, buffers de I/O y bits para recolección de basura. Los datos definidos por el sistema son generados automáticamente, conforme se van necesitando durante la ejecución de un programa sin una especificación explícita del programador.

Cuando un programador usa un lenguaje, para resolver un problema primero debe decidir la representación de los datos de ese problema, en términos de las estructuras de datos proporcionadas por los lenguajes de programación. De manera similar, cuando un lenguaje de programación es implementado en una computadora particular, el implementador debe determinar el tiempo de ejecución de la representación en memoria (estructura de almacenamiento) para las estructuras de datos en el lenguaje.

La representación interna de un dato dado, está compuesta por una localidad en memoria, que contiene una cuerda de bits que representan al dato y un descriptor que especifica la información adicional necesaria para descifrar la cuerda de bits.

Un descriptor incluye un "designador del tipo de datos", el cual especifica la clase general a que pertenecen los datos -números de punto fijo de punto flotante, cuerdas de caracteres, arreglos, listas, apuntadores, etcétera-, pero más generalmente, un descriptor contiene la información necesaria para descifrar completamente la cuerda de bits.

La cuerda de bits que representa a un dato puede existir en un --

bloque contiguo de memoria; en este caso, se habla de datos almacenados secuencialmente. De otra forma, la cuerda puede ser dividida en un número de -- piezas almacenadas en áreas separadas de memoria y ligadas con apuntadores; en este caso, se dice que los datos están ligados o encadenados.

Una declaración es una instrucción, que sirve para proporcionar al traductor del lenguaje, información acerca de las propiedades de los datos durante la ejecución. Una declaración puede especificar una variedad de cosas acerca de los datos como son tipo, tamaño, nombre, punto de creación, punto de destrucción, etcétera.

Una operación en un lenguaje de programación, es llamada una operación de tipo específico, sí el tipo de sus operandos y resultados es el mismo, se dice que una operación es genérica, sí el tipo de sus operandos y resultados puede variar. Por ejemplo, en un lenguaje con datos numéricos de tipo real y entero se requieren dos sumas de tipo específico, una suma real que adicione números reales y produzca un resultado real, y una suma entera que adicione números enteros y produzca un resultado entero; y sumas genéricas que acepten números reales o enteros (o una mezcla de ambos) y que produzcan un resultado de tipo real o entero dependiendo de sus operandos.

Se dice que un lenguaje permite un chequeo estático, cuando el programa necesita declaraciones que permitan checar el tipo de los datos durante la traducción y traducir operaciones genéricas a operaciones de tipo específico en el programa ejecutable. La otra alternativa es el chequeo

dinámico. Un problema que se presenta en el diseño de un lenguaje de programación es encontrar el balance entre la eficiencia de ejecución que se obtiene a través de las declaraciones y la flexibilidad posible sin ellas.

Una operación íntimamente ligada con la representación interna de los datos, es la del acceso a los elementos de una estructura. Ciertos tipos de datos, llamados datos elementales, pueden ser accesados como unidad, por ejemplo: números, valores Booleanos y apuntadores. Existen otros tipos de datos, llamados datos estructurados, en los cuales está permitido el acceso a partes de ellos, por ejemplo, en arreglos, listas y archivos de entrada. La distinción entre datos elementales y estructurados depende, obviamente, del lenguaje y del tipo de operaciones de acceso permitidas, por ejemplo, las cuerdas de caracteres en algunos lenguajes aparecen como datos elementales, que pueden ser accesados sólo como unidad para transmisión de entrada o salida a subprogramas, mientras que en otros lenguajes cada sub-cuerda puede ser accesada individualmente, y entonces una cuerda de caracteres es considerada como un dato estructurado.

Si A es un arreglo lineal, entonces accesar A[2], el segundo elemento de A, puede requerir recuperar los datos almacenados ahí, o bien recuperar la localidad del elemento, para que nuevos datos puedan ser almacenados ahí. Estas formas de acceso, se llaman acceso por valor y acceso por localidad. En la mayoría de los lenguajes ambos tipos de acceso tienen la misma sintaxis, por ejemplo: A[2] en X = A[2] + Y representa un acceso por valor, mientras que en A[2] = X + Y representa un acceso por localidad.

Es importante hablar de la diferencia que existe entre referenciar y accesar una estructura de datos. Por lo general, una estructura de datos tiene un nombre, por ejemplo, el arreglo que antes se llamó A . Cuando se escribe A [2] en un lenguaje de programación se está invocando una secuencia de dos pasos compuestos, primero de una operación de referencia, seguida de una operación de acceso (por localidad o por valor). La operación de referencia da como resultado un apuntador a la localidad de la entrada al arreglo designado con el nombre A . La operación de acceso toma el apuntador a la localidad del arreglo junto con el suscriptor 2 , que designa el elemento en el arreglo, y regresa un apuntador a la localidad del elemento dentro del arreglo.

3.1.1 Datos elementales

Un dato elemental es aquel que se accesa y modifica como una unidad. Como se mencionó antes ésta no es una distinción absoluta, ya que puede variar entre lenguajes. Números, cuerdas de caracteres, valores Booleanos, símbolos y apuntadores son los tipos básicos que serán analizados en esta sección.

3.1.1.1 Números

Algunas formas de datos numéricos son básicas en muchos lenguajes de programación, las distintas clases de números son relativamente familiares y los detalles de su tratamiento varían de lenguaje a lenguaje.

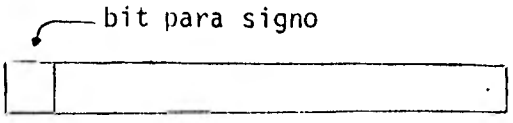
Organización Lógica .- En la mayoría de los lenguajes de programación, un número simple no tiene estructura interna más allá de su signo y magnitud (excepto en el caso de números complejos o racionales, que están compuestos de parejas de reales o enteros).

Estructura de Almacenamiento .- En general, no se usan descriptores para datos tales como enteros, reales y ocasionalmente reales de doble precisión; el número es representado como una cuerda simple de bits.

Por lo general, los números complejos se representan como parejas de números reales almacenadas en localidades consecutivas de memoria. La razón para que los números racionales sean evitados en un lenguaje, es el problema de redondeo y truncación encontrado en la representación de reales de punto flotante. Como resultado de esto es deseable representar a los racionales como parejas de enteros de longitud ilimitada.

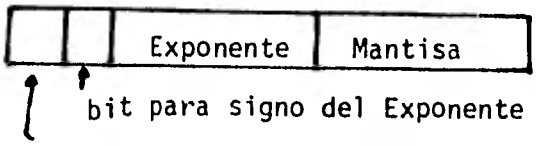
Algunas representaciones típicas para números, se ilustran a continuación:

a) Entero (punto fijo)

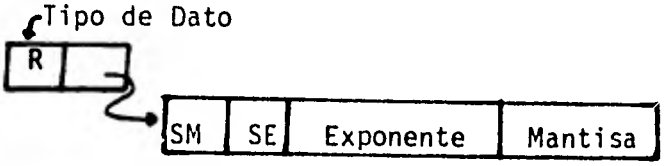


Cuerda de longitud fija, para representar un número.

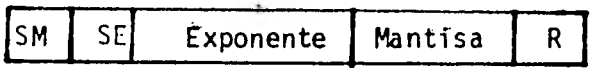
b) Real



(Representación sin descriptor).

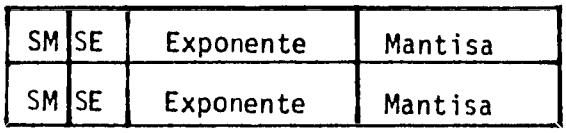


(Representación con descriptor almacenado en una palabra separada)



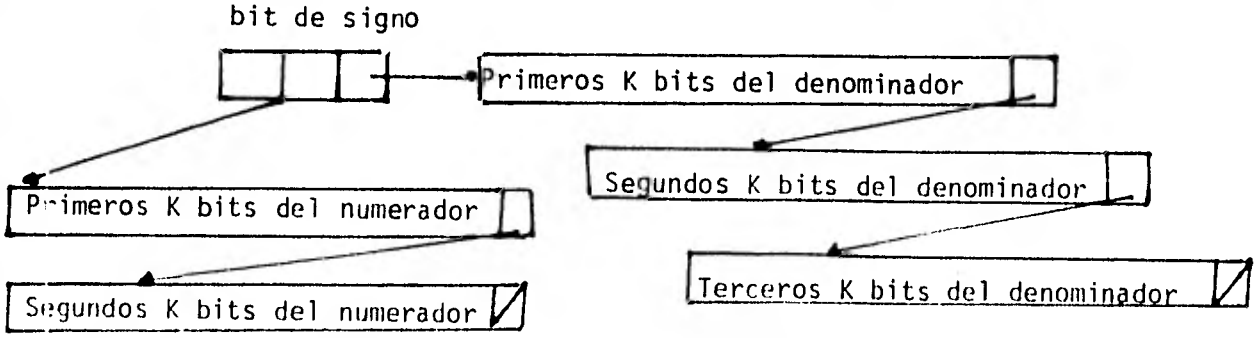
(Representación con descriptor almacenado en la misma palabra)

c) Complejo



Pareja de reales en localidades consecutivas de memoria.

d) Racional



3.1.1.2 Caracteres y cuerdas de caracteres.

Los caracteres como unidad rara vez forman un tipo de datos, los más comunes son las cuerdas de caracteres, que son series de caracteres individuales.

Organización lógica. Se pueden identificar tres diferentes formas de tratar a las cuerdas de caracteres:

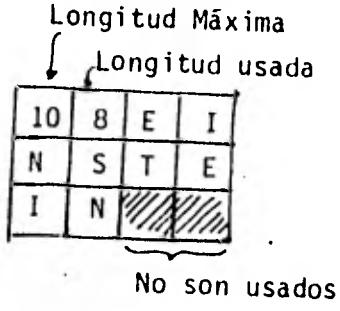
1. De longitud fija
2. De longitud variable dentro de un límite declarado
3. De longitud indefinida.

Estructura de almacenamiento. Cada caracter es representado por un código en 6 u 8 bits, y una cuerda de caracteres es representada por una serie de esos códigos almacenados en bytes consecutivos, empacados en palabras consecutivas de memoria. Cuando las cuerdas son declaradas de longitud fija, es posible representarlas sin un descriptor. Los casos de longitud variable requieren de un descriptor, al momento de ejecución, que especifique la longitud de la cuerda (y el límite en caso de fijo). Una representación común es, almacenar la longitud como si fuera el primer caracter en la cuerda de bits. Otra alternativa, cuando las cuerdas son de longitud ilimitada, es utilizar una representación de almacenamiento ligado. Estas estructuras se -- ilustran a continuación:

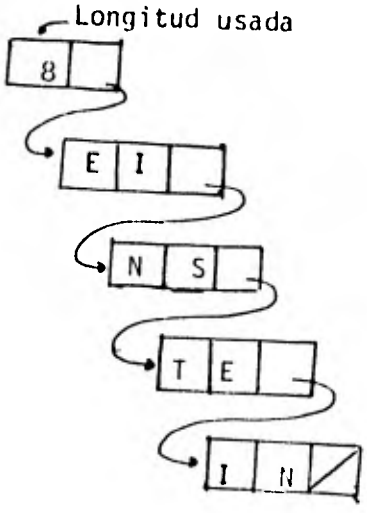
i) Cuerda de longitud fija (sin descriptor)

Código de 8 bits por caracter, almacenando 4 caracteres por palabra y llenando con blancos hasta tener 12 caracteres.

ii) Cuerda de longitud variable con cota declarada:



iii) Cuerda de longitud ilimitada (representación ligada)



Caracteres de 8 bits almacenados por parejas en cada palabra con un apuntador a la siguiente palabra.

3.1.1.3 Valores Booleanos

La mayoría de los lenguajes tienen algunas formas de datos binarios como un tipo de datos que únicamente toman el valor de cierto o falso; o en forma de cuerdas de bits. La primera forma generalmente es llamada datos de tipo Booleano. Los datos Booleanos tienen una representación obvia en la mayoría de las computadoras: como bits en el hardware. Las cuerdas de bits casi siempre son declaradas de longitud fija o acotada.

3.1.1.4 Apuntadores

Generalmente, los apuntadores, también conocidos como indicadores, referencias, localidades o direcciones, no están disponibles como datos en los lenguajes aunque pueden ser usados en la representación de almacenamiento para otros datos. Sin embargo, hay lenguajes en que una variable puede ser un apuntador y varias operaciones dan como resultado apuntadores. La mayor utilidad de hacer disponibles apuntadores como datos es, que el programador puede entonces, crear y manipular sus propias estructuras de datos en forma flexible La característica principal de los apuntadores es que estos "apuntan" a otros datos.

3.1.2 Arreglos

En esta sección, se hablará de la estructura de los arreglos lineales y sus extensiones multidimensionales. Se presentarán aquellas estructuras

de datos que son encontradas con frecuencia en los lenguajes de programación.

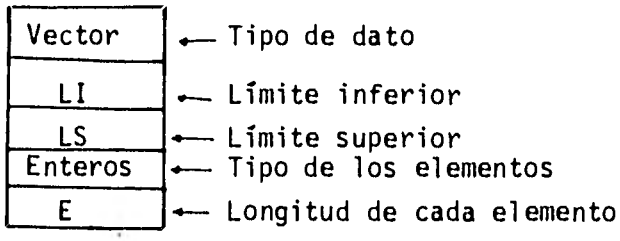
3.1.2.1 Arreglos de longitud fija

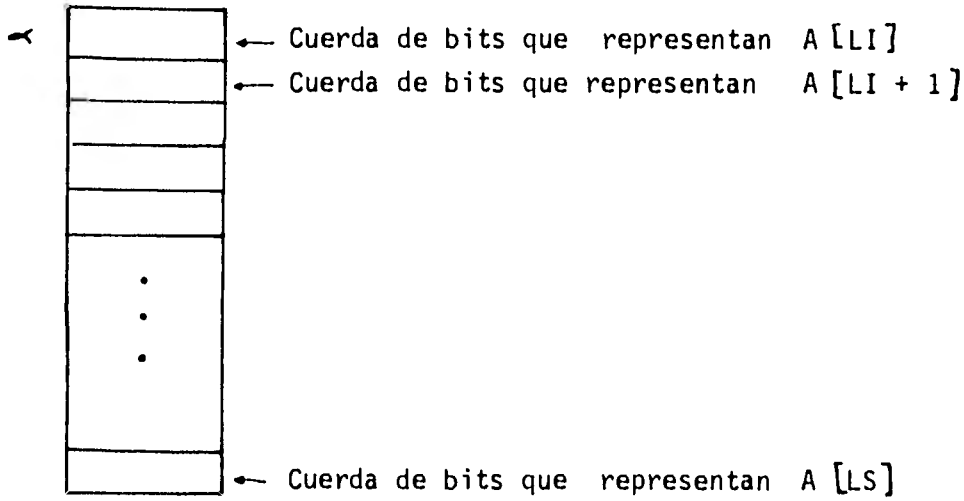
3.1.2.1.1 Vectores

Un arreglo lineal homogéneo de longitud fija es llamado un vector. La organización lógica de un vector es una serie de datos, donde cada uno puede ser accesado individualmente por su posición dentro de la serie, usando un entero llamado "subíndice". El dato individual dentro de la serie puede ser reemplazado por otros a través de asignaciones pero la cantidad de datos no puede ser alterada. Por otra parte, todos los datos tienen el mismo descriptor, que tiene tipo de datos, longitud y otros atributos comunes a todos.

La homogeneidad y longitud fija de un vector hace eficiente el almacenamiento y acceso de elementos individuales. La homogeneidad implica que la longitud y formato de la cuerda de bits, para cada dato dentro del vector es la misma, y el tamaño fijo implica que la cantidad de tales cuerdas es constante, durante el tiempo de vida del vector. Una representación secuencial es la apropiada para un vector. El descriptor para un vector debe especificar el tipo de datos del vector, el número de elementos o rango del subíndice y un descriptor para los elementos. A continuación se ilustra esta representación de almacenamiento:

Descriptor:





El acceso a un vector esta controlado por índices, cuyo valor puede ser calculado durante la ejecución. Por ejemplo, si A es el nombre de un vector, entonces A [I] se refiere al I-ésimo elemento del arreglo A . La situación no es tan simple, ya que se puede tener una cota inferior distinta de 1, por ejemplo, un vector A de 10 elementos puede tener definido un índice dentro del rango de 0 a 9, en tal caso, A [3] se refiere al cuarto elemento del vector.

La homogeneidad y el almacenamiento secuencial para los elementos

de un vector permiten localizar el I-ésimo elemento, usando la siguiente fórmula:

Localidad A [I] = α + (I - LI) * E

donde:

I = índice del elemento deseado,

α = localidad del principio de la cuerda de bits que representan al vector,

LI = (Límite Inferior) es el índice del primer elemento del vector

E = longitud de cada elemento.

3.1.2.1.2 Matrices y arreglos homogéneos de más de una dimensión

La extensión del concepto de vector a arreglos multidimensionales es inmediata. Generalmente un arreglo bidimensional homogéneo o matriz es organizado como una rejilla rectangular, de elementos en el plano; un arreglo tridimensional es visto como un paralelepípedo, etcétera. La homogeneidad implica que cada elemento del arreglo tiene el mismo descriptor, por ejemplo; una matriz de números enteros o una matriz de números reales, pero no una mezcla de las dos representaciones. Mientras que este concepto es inmediato del vector, es diferente el principio en que se basa la representación interna. Se considera a la matriz como un vector, cuyos elementos a su vez son vectores (cada uno de estos vectores es un renglón de la matriz).

En la representación interna, la división lógica de una matriz en renglones no es explícita; el almacenamiento es simplemente una gran serie de

cuerdas de bits, cada una representando un elemento de la matriz. Para arreglos multidimensionales la misma representación se extiende fácilmente al permitir un vector de vectores, de vectores, ..., a cualquier número de niveles. Al igual que con los vectores, es necesario un descriptor para los elementos del arreglo; pero además, por cada nueva dimensión hay un rango específico para su índice, y desde luego se necesita un indicador del número de dimensiones.

El acceso a arreglos multidimensionales puede realizarse usando una extensión de la fórmula utilizada para el acceso a vectores. Si A es un arreglo con n dimensiones, la localización del elemento $A[I_1, I_2, \ldots I_n]$ se realiza por medio de la siguiente fórmula:

$$\begin{aligned} \text{Localidad } A[I_1, I_2, \ldots, I_n] = \alpha &+ \left(\prod_{i=2}^{n} Ni\right)(I_1 - LI_1) + \\ &\left(\prod_{i=3}^{n} Ni\right)(I_2 - LI_2) + \ldots \\ &+ Nn\,(In - LI_{n-1}) + \\ &(I_n - LI_n) * E \end{aligned}$$

donde:

LIi = Límite inferior de la i-ésima dimensión

LSi = Límite superior de la i-ésima dimensioón

E = Longitud de cada elemento

$Ni = (LS_i - LI_i + 1) \times E$

A continuación se ilustra el almacenamiento de una matriz de 3 x 4

La matriz A:

A [1,1]	A [1,2]	A [1,3]	A [1,4]
A [2,1]	A [2,2]	A [2,3]	A [2,4]
A [3,1]	A [3,2]	A [3,3]	A [3,4]

Se vería como sigue:

Descritor:		
	Matriz	Tipo de datos
	LI_1	Límite inferior del 1er. índice
	LS_1	Límite superior del 1er. índice
	LI_2	Límite inferior del 2o . índice
	LS_2	Límite superior del 2o. índice
	Entero	Tipo de elemento
	E	Longitud de cada elemento

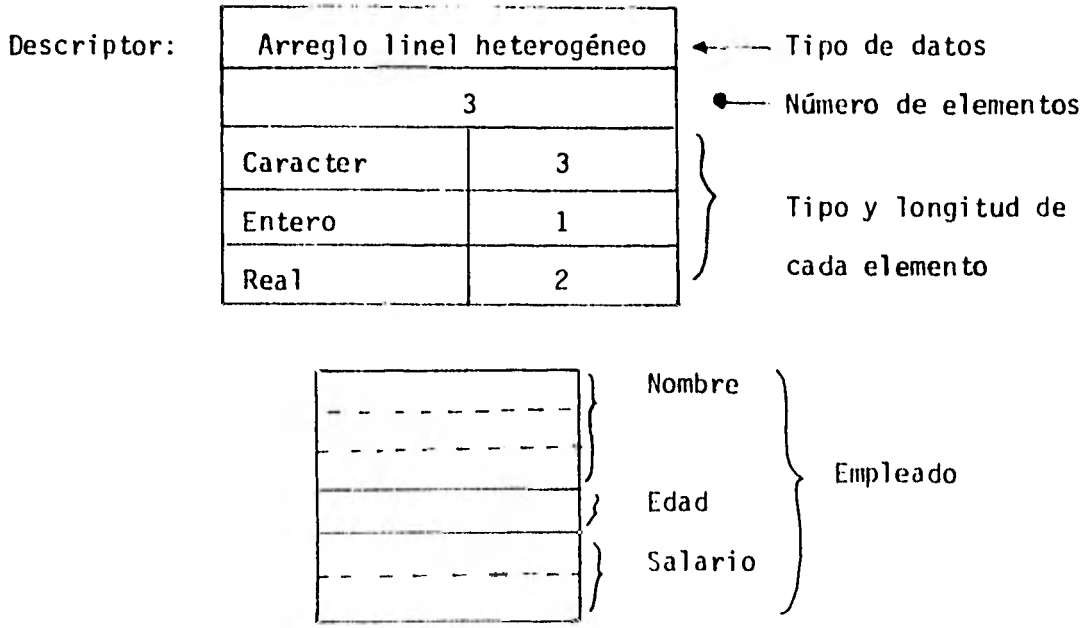
α	A [1,1]	
	A [1,2]	
	A [1,3]	1er. Renglón
	A [1,4]	
	A [2,1]	
		2o. Renglón
	A [2,4]	
	A [3,1]	
		3er. Renglón
	A [3,4]	

3.1.2.1.3 Arreglos lineales heterogéneos con declaraciones

La organización lógica de estos arreglos es idéntica a la de un vector, salvo que los elementos pueden ser de distinto tipo.

La representación interna para arreglos lineales de este tipo es similar a la de los vectores. Es usada una representación secuencial. El descriptor, para este tipo de arreglos, es necesariamente más complejo que para un vector por la variedad en el tipo de datos. Un descriptor completo debe -- constar de: tipo de datos (arreglo lineal heterogéneo), número de elementos y un descriptor para cada elemento.

La siguiente figura ilustra el almacenamiento de estas estructuras:



El acceso en un arreglo lineal heterogéneo es diferente al de un vector. El procesamiento secuencial es básico cuando se está trabajando con vectores; pero rara vez es apropiado cuando se trabaja con arreglos lineales heterogéneos, porque la variedad en el tipo de los elementos no permiten un tratamiento uniforme. Como resultado de esto, sólo se permite el acceso aleatorio (random) de elementos. La fórmula de acceso es la siguiente:

$$\text{Localidad } A[I] = \alpha + \sum_{j=1}^{I-1} \text{longitud de } A[J]$$

donde la sumatoria es necesaria por la posibildad de diferentes longitudes en cada elemento.

3.1.2.1.4 Arreglos multidimensionales heterogéneos con declaraciones

La extensión de arreglos lineales heterogéneos a arreglos multidimensionales se obtiene simplemente permitiendo que cada elemento del arreglo lineal sea otro arreglo lineal heterogéneo.

Como con los arreglos homogéneos multidimensionales, el almacenamiento, es por renglones. Los cálculos para el acceso de cualquier elemento en esa estructura pueden ser reducidos a la simple suma de una constante a la dirección base, que designa la localidad de la estructura en memoria.

3.1.2.1.5 Arreglos heterogéneos lineales sin declaraciones

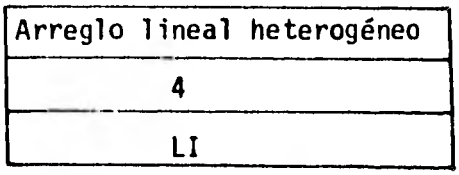
Son usadas diferentes representaciones internas y técnicas de acceso para arreglos heterogéneos, cuando no se dan declaraciones. No sólo el

tipo de cada elemento dentro del arreglo puede ser diferente, sino que se asume que cada asignación de un nuevo valor a un elemento puede cambiar el tipo del elemento almacenado ahí. Esto es, el tipo y en particular la longitud de cada elemento del arreglo puede variar dinámicamente durante la ejecución. La organización lógica de estos arreglos, no difiere de otros arreglos lineales, es una simple serie de elementos de tipo variable.

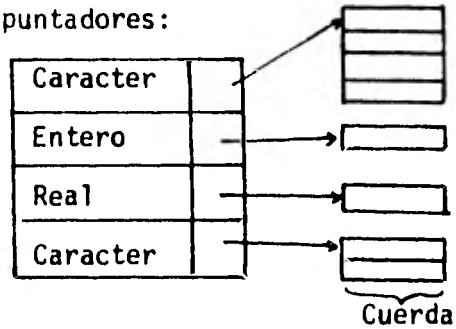
Aquí el número de elementos en el arreglo es fijo durante la ejecución, sólo la longitud de los elementos individuales puede variar. Bajo estas condiciones, es apropiado representar el arreglo como un vector de apuntadores a valores, y almacenar las cuerdas de bits que representan los valores en otra parte de memoria en localidades arbitrarias. El vector de apuntadores puede ser almacenado secuencialmente, porque el tamaño de cada apuntador es constante, independientemente del tipo del valor al que esté apuntando.

Los descriptores para cada elemento del arreglo pueden ser almacenados durante la ejecución; ya que los descriptores no son conocidos al momento de compilar. Estos descriptores pueden estar almacenados con las cuerdas de bits, para cada elemento individual. Si los descriptores son pequeños, pueden almacenarse con cada apuntador en el vector de apuntadores. Esta representación se ilustra a continuación:

Descriptor:



Vector de apuntadores:



El arreglo requiere un descriptor, que especifíque el tipo de datos del arreglo lineal, el número de elementos y el límite inferior en el -- rango del índice.

El acceso de A [I] , la posición del I- ésimo apuntador en el vector de apuntadores es calculado como si fuera un vector ordinario. Este apuntador entonces da la localidad en memoria de la cuerda de bits que representan al valor.

3.1.2.2 Arreglos de longitud variable

En la mayoría de las representaciones de los datos para un problema, éstos pueden ser introducidos a través de un dispositivo de entrada, y su

extensión no ser conocida, o pueden ser generados internamente en un programa de manera impredecible. Para permitir la representación natural y manipulación de estos datos, los lenguajes permiten arreglos lineales que aumentan o disminuyen su tamaño en forma dinámica, durante la ejecución del programa. Estos arreglos de longitud variable son conocidos por varios nombres: pilas, colas, listas y tablas por mencionar algunos. La organización lógica de estos arreglos no es muy diferente a la de los arreglos de longitud fija; sin embargo, el hecho de que la longitud varia hace diferente la representación interna y las técnicas de acceso.

En los arreglos de longitud fija, cada elemento puede ser accesado individualmente por un índice. El acceso en arreglos de longitud variable tiende a ser relativo; accesar el elemento que está antes (o después) de éste, traer el último registro, etcétera. El acceso por índice es poco común porque cada elemento del arreglo puede alterar su posición, si el tamaño del arreglo cambia.

3.1.3 Conjuntos

Un conjunto es una colección desordenada de elementos con tres -- operaciones básicas:

1. Prueba de pertenencia. ¿El dato X, es un miembro del conjunto S ?.
2. Suma de un elemento. Agregar un dato X a un conjunto S es posible si $X \notin S$.

3. Supresión de un elemento. Borrar el dato X del conjunto S.

La representación más común de almacenamiento para un conjunto se basa en la técnica conocida como dispersión o "hash". Se reserva un bloque de almacenamiento para el conjunto y los elementos son dispersados aleatoriamente dentro del bloque. El truco es almacenar cada nuevo elemento, de tal forma que su presencia o ausencia pueda ser determinada inmediatamente sin tener que buscar en el bloque.

Supongase que se desea agregar el elemento X, representado por la cuerda de bits Bx, al conjunto S, que está representado por el bloque de almacenamiento Ms. Primero se debe determinar si X ya es miembro de S y si no, agregarlo. Para determinar una posición para Bx dentro de Ms se debe aplicar una función de "hashing" a los bits de Bx; dicha función mezcla los bits de Bx y luego extrae una dirección Ix del resultado. Esta dirección es usada como índice en el bloque Ms. Si la posición indicada por Ix está vacía, entonces X debe ser almacenado ahí, en caso contrario se dice que X ya pertenece a S. Como se ve, para saber si un elemento existe o no, en el bloque, no es necesario buscar en ninguna tabla.

Lo óptimo sería que la función de dispersión repartiera los datos dentro del bloque sin problema alguno, pero es inevitable que al aplicar la función a dos datos distintos den la misma dirección, a esto se llama colisiones.

La forma más común de tratar las colisiones es ligando todos los elementos que tienen la misma dirección, de esta manera al aplicarle la función de dispersión a una cuerda Bx , se busca en la lista de los elementos ligados a la dirección dada por la función y si no se encuentra se agrega al final de la lista.

La técnica del "hash" es usada cuando el universo del conjunto es muy grande; pero cuando éste es pequeño, la representación de almacenamiento más apropiada es usar una cuerda de bits. Supongase que el universo tiene N elementos: e_1, ... , en, entonces el conjunto es representado usando una cuerda de N bits, donde el valor de i-ésimo bit es 1 si e_i está en la conjunto y es cero en caso contrario. La cuerda de bits representa la función característica del conjunto. Con esta representación, insertar un nuevo elemento es tan fácil como poner un 1 en el lugar correspondiente dentro de la cuerda de bits y suprimir un elemento requiere poner un 0 en dicho lugar.

3.1.4. Archivos externos

Los datos pueden estar en medios externos de almacenamiento, para próposito de entrada-salida, o para almacenamiento temporal cuando no hay espacio disponible en memoria central. Estos archivos externos tienden a tener organizaciones relativamente simples. Los más comunes son los archivos secuenciales, directos y secuenciales indexados.

Los archivos secuenciales son los más utilizados, en la mayoría de

los lenguajes. Son vistos como una sucesión de registros de tamaño fijo. Estos registros deben ser accesados en el orden en que ellos aparecen dentro del archivo, es decir, no se puede regresar o avanzar hasta un registro particular.

Los archivos de acceso directo se organizan como un conjunto de registros, sin orden alguno. El acceso es a través de una dirección que indica la posición del registro en el dispositivo externo.

Los archivos secuenciales-indexados son el punto medio entre un archivo secuencial puro y uno de acceso directo. Un archivo secuencial indexado se organiza como un archivo secuencial ordenado, donde cada registro contiene un campo que sirve como llave de acceso. El acceso aleatorio a registros es permitido, utilizando las llaves como subíndices y el archivo también puede ser procesado secuencialmente desde cualquier punto dado aleatoriamente.

3.2 Operaciones

Las operaciones forman el complemento de los datos en programación Los datos representan el componente pasivo, la información almacenada y las operaciones representan el componenete activo, que crea, destruye y tranforma los datos.

Las operaciones pueden clasificarse en operaciones con datos definidos por el programador y operaciones con datos definidos por el sistema. Las primeras incluyen aquellas usadas por el programador como son suma, resta,

raíz cuadrada, prueba de igualdad, etcétera. Las otras incluyen aquellas que forman parte de la estructura de control de los lenguajes como son GO TO's, llamadas a subprogramas, transmisión de parámetros, etcétera.

Las operaciones con datos definidos por el programador pueden dividirse en operaciones primitivas,operaciones construídas por el lenguaje, y operaciones definidas por el programador o subprogramas.

3.2.1 Operaciones elementales.

Cada lenguaje tiene incorporado un conjunto de primitivas que ejecutan operaciones básicas con datos simples; por ejemplo: operaciones aritméticas, operaciones lógicas y conversiones del tipo de datos.

Las operaciones aritméticas básicas de suma, resta, multiplicación, división y exponenciación, son primitivas en la mayoría de los lenguajes. En lenguajes usados para cálcualos científicos se extiende el conjunto de operaciones incluyendo raíz cuadrada, operaciones trigonométricas y otras operaciones especializadas.

Las pruebas de relación en números (igual, diferente, menor, mayor, menor o igual, mayor o igual) son otro conjunto de primitivas encontradas en los lenguajes de programación. Las operaciones de relación toman la forma de funciones simples, aceptando dos números como operandos y produciendo un valor Booleano (cierto o falso) como resultado. Las operaciones de rela

ción pueden ser extendidas a cuerdas de caracteres usando una "secuencia comparada", para inducir un orden en las cuerdas de caracteres (una extensión del orden alfabético usual).

Las operaciones Booleanas son primitivas en los lenguajes. Una operación Booleana acepta sólo valores Booleanos como operandos y produce un valor Booleano como resultado. Las operaciones básicas son AND, OR y NOT, pero en ocasiones son extendidas a equivalencia Booleana, implicación, OR exclusivo, NAND (NOT-AND) y NOR (NOT-OR). Las operaciones Booleanas pueden ser extendidas a cuerdas de bits. Por ejemplo, el AND de dos cuerdas de bits de la misma longitud, es una cuerda donde cada bit se determina tomando el AND de los bits correspondientes en las cuerdas de operandos.

Las operaciones de conversión de tipo, usadas para convertir datos en diferentes representaciones son importantes en varios lenguajes. Estas operaciones pueden ser ejecutadas directamente, pero en general, son ejecutadas implícitamente cuando se presenta un conflicto de tipos al intentar hacer otras operaciones.

La asignación es la operación básica para modificar las estructuras de datos.La asignación difiere de las operaciones antes tratadas, en que no produce un valor para una función en el sentido usual, en lugar de esto modifica sólo uno de sus operandos. Para operaciones de asignación se requiere un valor que será almacenado y un apuntador a una localidad en la estructura de datos, en donde el valor será almacenado.

3.2.2 Creación de estructuras de datos e inserción de elementos

Las operaciones que crean nuevas estructuras de datos, o que amplian las estructuras existentes insertando nuevos elementos, son de gran importancia en todo lenguaje.

La creación de una estructura de datos involucra cuatro pasos básicos:

1. Creación de un descriptor para la estructura;
2. Localización de almacenamiento para la estructura;
3. Especificación de los valores de los elementos de la estructura, y
4. Creación de trayectorias de acceso para los datos.

Las operaciones de inserción involucran localidades de almacenamiento para los nuevos elementos, especificación de sus valores y, en ocasiones, ajuste de apuntadores.

Anteriormente se presentaron los descriptores y las cuerdas de bits, que representan los valores de los elementos para varias estructuras de datos. La creación de una trayectoria de acceso para una nueva estructura se realiza a través de la asociación de la estructura con un identificador, su nombre, o almacenando un apuntador a la estructura en otra estructura que ya se encuentre accesible.

3.2.3 Destrucción de estructuras de datos y supresión de elementos

En la discusión de las operaciones de destrucción se debe, ser cuidadoso de, distinguir la operación de destrucción (la cual hace que la estructura de datos sea lógicamente inaccesible) de la operación de recuperación de almacenamiento para la estructura.

Una dificultad adicional es que, rara vez, un lenguaje proporciona una operación explícita que el programador pueda usar para destruir una estructura de datos ya existente. La estructura es "destruída" cuando todas las trayectorias de acceso se han destruído.

Cuando se crea una estructura de datos, también debe ser creada una trayectoria de acceso; de otra forma no es posible llegar a la nueva estructura. La forma más común de hacer esto es asociando la estructura con un identificador o almacenando un apuntador a ella en otra estructura. Durante el tiempo de vida de la estructura, la trayectoria de acceso inicial puede ser aumentada por otras, es muy común que existan múltiples trayectorias de acceso a una estructura simple.

Supongase que el programador puede indicar explícitamente cuando una estructura particular será destruída. Primero debe indicar una trayectoria de acceso a la estructura y la operación de destrucción puede rápidamente destruir tanto la trayectoria de acceso, como la estructura misma. Sin embargo, si existe alguna otra trayectoria de acceso a la estructura, se presenta

el problema de pretender destruir una trayectoria de acceso a una región de almacenamiento que no está definida.

Esas trayectorias de acceso a estructuras inexistentes, se llaman referencias colgantes. Las referencias colgantes pueden ser suprimidas usando una operación que elimine una trayectoria de acceso; pero sin liberar la localidad de almacenamiento hasta que la última trayectoria de acceso sea destruída. Sin embargo, no es factible tener a la vista todas las trayectorias de acceso a una estructura, es por eso que se pueden perder todas las ligas de acceso a una estructura, sin que ésta haya liberado el espacio. En este caso, se dice que la estructura es basura. Una estructura así, es aquella que existe pero que, no puede ser accesada porque no existe ninguna trayectoria de acceso para ella. La dificultad con estas estructuras es la recuperación del almacenamiento que están ocupando.

Cuando el espacio libre está completamente agotado y se necesita almacenar algo más, entonces se utiliza un procedimiento instituído llamado recolector de basura. Para poder utilizarlo cada elemento, que forma parte de una estructura de datos accesible, debe tener un bit que sirva para indicar si es basura o no; el recolector de basura checa ese bit en cada elemento, libera el espacio que ocupan aquellos elementos que lo tengan prendido y apaga el bit indicador.

3.2.4 Patrón de Igualdad

Las operaciones de patrón de igualdad difieren de las anteriores en que, éstas determinan dinámicamente la parte de la estructura, donde se va a operar en términos de relaciones entre varios elementos de la estructura. Se distinguen dos tipos de estas operaciones:

1. Patrón de igualdad simple. El patrón de igualdad puede ser usado simplemente para probar ciertas relaciones de los elementos dentro de la estructura.

2. Patrón de igualdad con reemplazamiento. El patrón de igualdad puede ser usado para identificar una sub-parte de la estructura, la cual será reemplazada por una nueva estructura designada. Si la nueva estructura es el elemento nulo, el resultado es una supresión, si es la sub-parte original más algún nuevo dato, el resultado es una inserción.

En el patrón de igualdad simple el resultado será un valor Booleano (cierto o falso) dependiendo de si la sub-cuerda fue encontrada o no.

Las operaciones de construcción de patrones de igualdad simples pueden ser combinadas usando las siguientes operaciones:

1. Concatenación. Si A y B son patrones, entonces la concatenación de A y B es un patrón que compara una sub-cuerda, que empieza con A inmediatamente seguida de B.

2. Alternancia. Si A y B son patrones, entonces la alternancia de A o B, es un patrón que compara cualquier sub-cuerda de A o de B.

Las técnicas de entrada-salida, en la mayoría de los lenguajes, implican el uso de un tipo de patrón de igualdad para convertir la representación externa de datos en forma de cuerdas de caracteres a representación interna en memoria. El patrón es definido como formato. Durante la entrada, el formato especifica el patrón de campos conteniendo números que serán encontrados en los registros de entrada, y la forma interna en que serán convertidos. El reverso es efectuado durante la salida, el formato define el patrón de caracteres que serán creados en el registro de salida, incluyendo conversiones de números en forma binaria a cuerdas de caracteres, dejando espacios entre números, insertando comentarios o títulos, etcétera.

3.2.5 Operaciones definidas por el programador

Supongase que un programador no está satisfecho con las operaciones primitivas del lenguaje que está usando, así que, desea extender el lenguaje incluyendo nuevas operaciones diseñadas por el mismo. Las facilidades para construir nuevas operaciones, en la forma de subprogramas, son básicas en todo lenguaje de programación.

El análisis de los subprogramas se divide en varias partes: las estructuras de control que involucran la llamada y el mecanismo de regreso, el de transferencia de datos y de subprogramas serán tratados después; aquí se tratarán los subprogramas vistos como operaciones definidas por el programador.

Dos categorías de subprogramas se pueden distinguir dependiendo si el subprograma está escrito en el mismo lenguaje, que programa principal o no. Generalmente los subprogramas son escritos en el mismo lenguaje. El cuerpo de un subprograma normalmente, toma la misma forma que el programa principal, sólo con una proposición inicial que lo distinga. Cualquier algoritmo que pueda ser programado en ese lenguaje puede entrar en el cuerpo.

El segundo tipo de subprogramas es cuando son codificados en otro lenguaje. Estos subprogramas son usados con poca frecuencia. Su utilidad está restringida a circunstancias especiales, por la dificultad de comunicación entre programas escritos en diferentes lenguajes y las diferencias en los requerimientos de almacenamiento.

Los subprogramas se dividen en funciones (si regresan un resultado explícito) y subrutinas (si no regresan un resultado explícito). Las funciones son usadas en expresiones, donde el resultado sirve inmediatamente de entrada a otra operación. Las subrutinas deben ser llamadas en forma separada por lo general, con una proposición CALL.

Los operandos de los subprogramas son llamados "parámetros" o "argumentos". Un operando puede ser cualquier dato. Los resultados de subprogramas pueden aparecer como valores de funciones explícitas o como modificaciones de los parámetros o de las variables no-locales. En muchos lenguajes los resultados de los subprogramas están restringidos a no interferir con el manejo de almacenamiento. Como con las operaciones primitivas, es importante tener

declaraciones del tipo del resultado de una función.

3.3 Control de secuencia

Las estructuras de control en un lenguaje de programación proporcionan el marco dentro del cual las operaciones y los datos se combinan en un programa. Hasta aquí, se han visto los datos y las operaciones en forma aislada ahora se considerará su organización dentro de un programa ejecutable. Esto involucra dos aspectos, el control del orden de ejecución de las operaciones, el cual se llamará control de secuencia y el control de transmisión de datos entre conjuntos de operaciones, el cual se llamará control de datos.

Diferentes estructuras de control de la secuencia, son usadas en varios lenguajes de programación y se dividen en tres grupos:

1. Estructuras usadas en expresiones, tales como: reglas de precedencia y paréntesis;
2. Estructuras usadas entre proposiciones o grupos de ellas, tales como: proposiciones condicionales y de iteración, y
3. Estructuras usadas entre subprogramas, tales como: llamadas a subprogramas y corutinas.

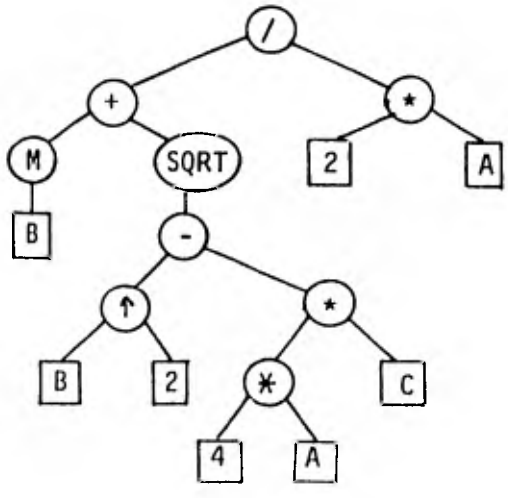
Las estructuras de control de secuencia pueden ser implícitas o explícitas. Implícitas (o por omisión) son aquellas definidas por el lenguaje y que tienen efecto a menos que sean modificadas por el programador con alguna estructura explícita. Las estructuras de control explícitas son aquellas que

pueden ser usadas por el programador, para modificar la secuencia implícita de las operaciones definidas por el lenguaje, por ejemplo, el uso de paréntesis dentro de las expresiones o una proposición GO TO y etiquetas.

3.3.1 Control de secuencia dentro de expresiones

Los mecanismos para controlar la secuencia en una expresión, operan para determinar el orden en que se realizarán las operaciones dentro de la expresión. Estos mecanismos son composiciones funcionales. En una operación principal los operandos pueden ser constantes o el resultado de otras operaciones, cuyos operandos pueden ser, a su vez constantes o el resultado de otras operaciones y así sucesivamente. La composición funcional proporciona a una expresión la estructrua de un árbol, donde la raíz representa a la operación principal, los nodos entre la raíz y las hojas representan operaciones intermedias y las hojas representan constantes o referencias de datos. Por ejemplo, la expresión de la forma cuatrática puede ser representada de la siguiente forma:

```
S := (-B + SQRT (B ** 2 - 4 * A * C)) / (2 * A )
```



Donde M es el menos unario, SQRT es la raíz cuadrada y ↑ es la exponenciación.

Antes de determinar el orden exacto de evaluación (por ejemplo, cuando -B ó B↑2, será evaluado primero), es apropiado conocer varias representaciones sintácticas para expresiones.

Notación prefija. En esta notación se escriben los símbolos de las operaciones, seguidos de los operandos. Una variante es llamada notación polaca (o libre de paréntesis). Ejemplo: (A + B) * (C - A) en notación polaca quedaría * + A B - C A.

El problema con esta notación, es la dificultad para descifrar la expresión. En efecto, la forma polaca no puede ser descifrada sin conocer el número de operandos que cada símbolo requiere.

Notación sufija. La notación sufija o postfija, es similar a la notación prefija, excepto que los símbolos de operación siguen a los operandos. Por ejemplo, la expresión (A + B) * (C - A) quedaría representada como A B + C A - * .

Notación infija. Esta notación está disponible sólo para operaciones binarias, es decir, operaciones con dos operandos. En la notación infija, el operador se escribe entre sus operandos. Como esta notación es usada en las matemáticas para operaciones aritméticas básicas, de relación y lógicas, entonces ha sido ampliamente adoptada en los lenguajes de programación. La expresión (A + B) * (C - A), en notación infija quedaría igual.

Aunque la notación infija es de uso común en los lenguajes de pro-

gramación también tiene ciertos problemas entre los que cabe señalar los siguientes:

a) Como la notación infija es aplicable sólo para operadores binarios, un lenguaje no puede usar únicamente esta notación, así que debe combinar las notaciones infija y prefija. Esta mezcla hace que la traducción de expresiones sea más compleja.

b) Cuando más de un operador infijo aparece en una expresión, la notación se hace ambigua, a menos que se usen paréntesis. Por ejemplo, la expresión infija A * B + C puede representar cualquiera de estas dos expresiones: (A * B) + C ó A * (B + C).

Los paréntesis pueden ser usados explícitamente para indicar el agrupamiento; pero en expresiones complejas los paréntesis anidados se prestan a confusiones. Por esta razón, los lenguajes introducen reglas de control implícitas, que hacen innecesario el uso de paréntesis. Los dos tipos más comunes de reglas son:

1 . Jerarquía de operación (reglas de precedencia). Los operadores que aparecen en una expresión son puestos en orden de precedencia. Ejemplo de jerarquía de operaciones:

Jerarquía	Operación
3	↑
2	* , /
1	+ , -
0	< , ≤ , = , ≠ , ≥ , > .

En una expresión se ejecutará primero, la operación con más alta jerarquía. Por ejemplo, en A * B + C , la multiplicación tiene más prioridad que la suma, así que primero se ejecutará la multiplicación.

2. Asociatividad. En una expresión con operaciones en el mismo nivel jerárquico, es necesaria una regla de asociatividad para definir por completo el orden de las operaciones. Por ejemplo, en A + B - C ¿se ejecuta primero la suma o la resta? En general, la asociatividad se realiza de izquierda a derecha, así que A + B - C es tratada como (A + B) - C.

3.3.2 Control de secuencia entre proposiciones

La regla implícita que gobierna el orden de ejecución de las proposiciones, es la que dice que la ejecución procede de acuerdo al orden físico de las proposiciones dentro de un programa. La ejecución empieza con la primera proposición, cuando la ejecución termina, se ejecuta la segunda proposición, seguida de la tercera, etcétera.

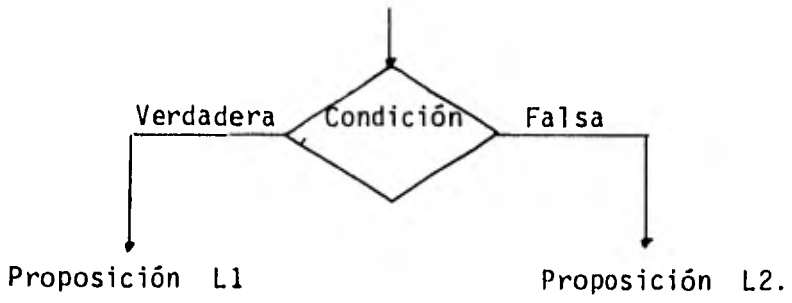
Una forma muy común de alterar el orden de ejecución, es transfiriendo explícitamente el control a alguna proposición etiquetada usando la proposición GO TO.

El uso de etiquetas y proposiciones GO TO crean una gran controversia, tanto que en los nuevos lenguajes su uso ha sido eliminado. La principal desventaja del uso de proposiciones GO TO, es que un programa que los

usa es difícil de depurar y aún más difícil de entender y dar mantenimiento. La dificultad, en el seguimiento de la estructura del programa, estriba en que dado que el control puede ser transferido a una proposicón etiquetada desde - cualquier lugar dentro del programa, no hay forma de determinar de donde -- ocurrió sin ratrear todo el programa.

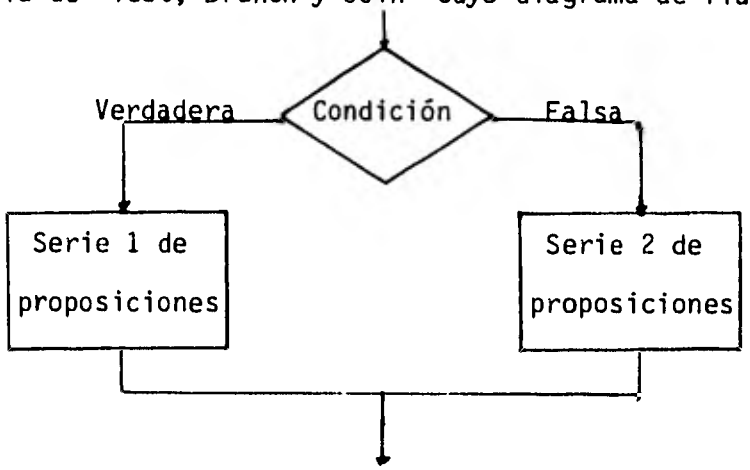
Esta discusión lleva a otras estructuras de control que sirvan para complementar o suplantar el uso de GO TO's. Una manera es, usar la "proposición condicional" que se encuentra en la mayoría de los lenguajes y en su forma más simple aparece como "prueba y salta"; pero existen otras formas más sofisticadas.

La forma más simple de la proposición condicional está representada en el siguiente diagrama de flujo:



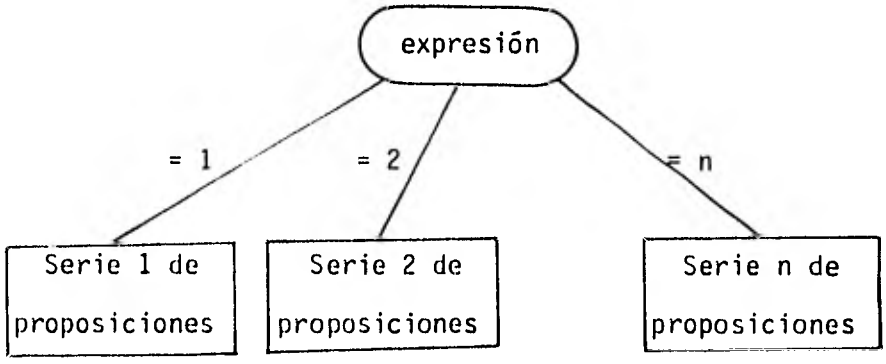
Esta proposición tiene la misma desventaja que el GO TO simple: su uso puede propiciar programas pobremente estructurados.

Una forma más sofisticada, que la proposición condicional antes mencionada, es la de "Test, Branch y Join" cuyo diagrama de flujo es como sigue



Una estructura de este tipo es más fácil de analizar que la anterior. Si la condición es verdadera efectúa la serie 1 de proposiciones, si es falsa efectuá la serie 2, en ambos casos el programa continua en una proposición común.

La proposición CASE, disponible en lenguajes recientes, es una extensión de la proposición "Test, Branch y Join". La forma típica del CASE es como sigue:



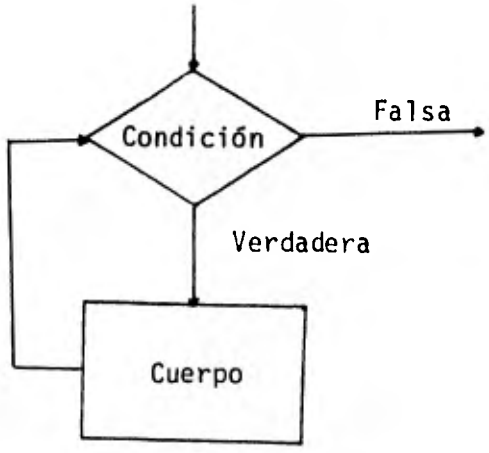
La expresión debe tener un valor entero en el rango 1,2, ..., n y la serie de proposiciones correspondiente es ejecutada. La proposición CASE es una proposición condicional, en la que el programador especifica n-diferentes caminos, en oposición a la característica de dos caminos de las condicionales simples. En resumen varias alternativas serán ejecutadas. El uso de esta proposición tiende a reducir la necesidad de usar GO TO's en los programas

Las proposiciones iterativas proporcionan otra alternativa para controlar la secuencia. A diferencia de las condicionales, las proposiciones de iteración pueden ser reemplazadas por series de proposiciones simples en un programa. En efecto, no es poco común encontrar la semántica de las proposiciones de iteración definida en términos de series equivalentes de proposiciones simples.

La estructura básica de una proposición de iteración consta de un cuerpo y un encabezado. El cuerpo por lo general, está compuesto de una serie de proposiciones; el encabezado consta de una expresión que designa el número de veces que será ejecutado el cuerpo. A continuación se describirán algunas formas comunes de estas proposiciones.

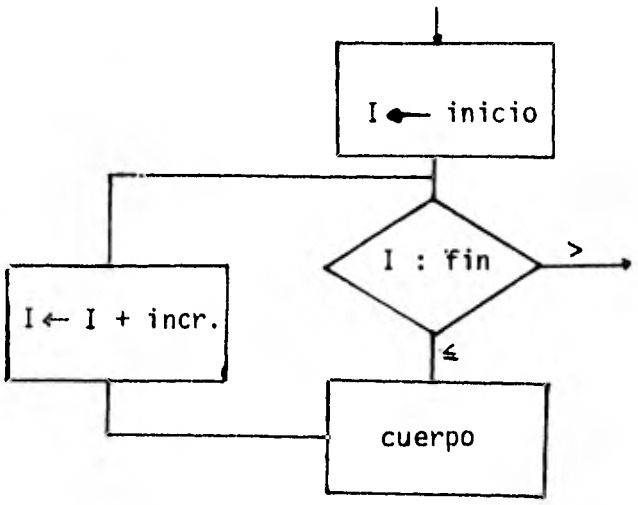
El tipo más simple de esta proposición es aquel en el cual, el encabezado especifica que el cuerpo será ejecutado sólo un número fijo de veces.

Otra forma más compleja de iteración puede ser construída, usando un encabezado "WHILE REPEAT". El significado de esta construcción puede ser representado en el siguiente diagrama de flujo.



En esta forma de iteración, la condición es evaluada cada vez que se termina la ejecución del cuerpo, además se espera que durante la ejecución del cuerpo cambien algunos valores de las variables, que aparecen en la condición; porque de otra forma una vez que empieza la iteración no termina nunca.

La tercera forma de proposición iterativa es aquella en la cual, el encabezado especifica una variable que sirve como contador o índice durante la ejecución. Se debe especificar un valor inicial, uno final y un incremento y el cuerpo se ejecutará repetidamente usando primero el valor inicial como valor de índice, luego el valor inicial más el índice, después el valor inicial más dos veces el índice, etcétera, hasta que el valor final sea alcanzado. Esta proposición se ejemplifica en el siguiente diagrama de flujo:



Las proposiciones de iteración constituyen otra alternativa para el mejor uso del mecanismo del GO TO. Como con las proposiciones condicionales, las iteraciones no requieren el uso de etiquetas y por su estructura -- son fáciles de analizar. Por estas razones, las proposiciones de iteración son las preferidas para controlar la secuencia en programas con estructuras repetitivas.

3.3.3 Control de secuencia en subprogramas

Un programa está compuesto de un programa principal, el cual durante su ejecución puede llamar a varios suprogramas que a su vez pueden llamar a otros sub-subprogramas y así sucesivamente. Cada subprograma al terminar su ejecución regresa el control al programa de donde fue llamado. Durante la ejecución de un subprograma, la ejecución del programa que lo llamó es suspendida temporalmente y cuando el subprograma termina, el programa continúa su ejecución en la proposición inmediata después de la llamada al subprograma.

Esta estructura de control es explicada por la regla de copia. El efecto del CALL al subprograma, es el mismo que se habría obtenido si la proposición de llamada (CALL) se reemplazara por una copia del cuerpo del subprograma (con las substituciones pertinentes para los parámetros e identificadores conflictivos), antes de la ejecución. Desde este punto de vista, las llamadas a subprogramas pueden ser consideradas como estructuras de control, que hacen innecesaria la copia de gran cantidad de proposiciones idénticas, o casi idénticas que ocurren en más de un lugar de un programa. Sin embargo, si el subprograma contiene una o dos proposiciones es mejor escribirlas que llamar a un subprograma.

Antes de considerar las diferentes estructuras de control que involucran subprogramas, es importante distinguir entre definir un subprograma y activarlo. Hasta aquí se ha usado el término subprograma para referirse tanto a su definición (que es la representación de un subprograma cuando es escrito por un programador), como a su activación (que es la representación del subprograma durante la ejecución). Para subprogramas simples esta confusión es irrelevante; sin embargo, la distinción es crucial cuando se consideran subprogramas recursivos y otras estructuras de control de subprogramas más generales.

Un subprograma es directamente recursivo, si contiene una llamada a sí mismo y es indirectamente recursivo si llama a otro subprograma que llama al subprograma original o inicia una serie de llamadas a subprogramas hasta que una de estas llamadas es al subprograma original. La recursividad, es una de las estructuras de control de secuencia más importante en programación

ya que muchos algoritmos son representados más naturalmente usando recursividad.

Mientras que sintácticamente no hay diferencia entre subprogramas recursivos y no recursivos, la simulación necesaria para la implementación de llamadas recursivas es diferente, de la necesaria para llamadas no recursivas.

La dificultad es explicada simplemente en términos de activación de subprogramas. Para cada activación de un subprograma, es necesario almacenar un punto de retorno, porque cada activación puede realizarse desde cualquier punto (con la proposición CALL o referencia de una función). Con la recursividad hay varias activaciones del mismo subprograma, obviamente, un único punto de retorno no es suficiente. En vista de que no se puede predecir el tamaño máximo de un bloque de almacenamiento para los puntos de retorno, es más cómodo usar un stack que pueda ser expandido tanto como se necesite durante la ejecución.

Asociando un stack con cada subprograma para el almacenamiento de puntos de retorno es fácil la simulación para llamadas recursivas. El stack es usado de la siguiente forma: en la primera llamada (no recursiva) a un subprograma se guarda el punto de retorno en la primera localidad del stack. En la primera llamada recursiva, se guarda el punto de retorno para esta llamada en la segunda localidad del stack. En la segunda llamada recursiva, se usa la tercera localidad del stack y así sucesivamente. Cuando se termina la ejecución del subprograma y se debe regresar, es necesario encontrar la última --

entrada al stack y usarla como localidad de retorno, borrándola después.

El mecanismo de simulación, que asocia un stack de puntos de retorno con cada subprograma, puede ser simplificado usando un sólo stack central de puntos de retorno para todos los subprogramas.

Una alternativa importante en las estructuras de control se obtiene cuando se llama al subprograma al presentarse una condición particular, en lugar de llamarse cuando la ejecución de un programa llega a un punto específico. Por ejemplo:

a) Llamar a un subprograma que procese un error cuando una operación aritmética causa "overflow" o, cuando se refiere a un elemento de un arreglo con un subscriptor fuera de rango.

b) Llamar a un subprograma que maneje encabezados especiales de salida al final de una página de impresión o, cuando el proceso indica un fin de archivo en un archivo de entrada.

c) Llamar a un subprograma que imprima un rastro durante un programa indicando cuando entró o salió de un subprograma o, cuando cambia el valor de una variable.

Es posible hacer, explícitamente, pruebas para esas condiciones y llamar al subprograma apropiado, sin embargo, la estructura de control "INTERRUPT" con una llamada implicita es mucho más simple. Típicamente esta estructura de control se divide en dos partes:

1. Especificación de la rutina de interrupción y su asociación

con una condición particular. Esta especificación casi siempre aparece como declaración al principio de un programa en la siguiente forma:

ON <condición> CALL <subprograma>

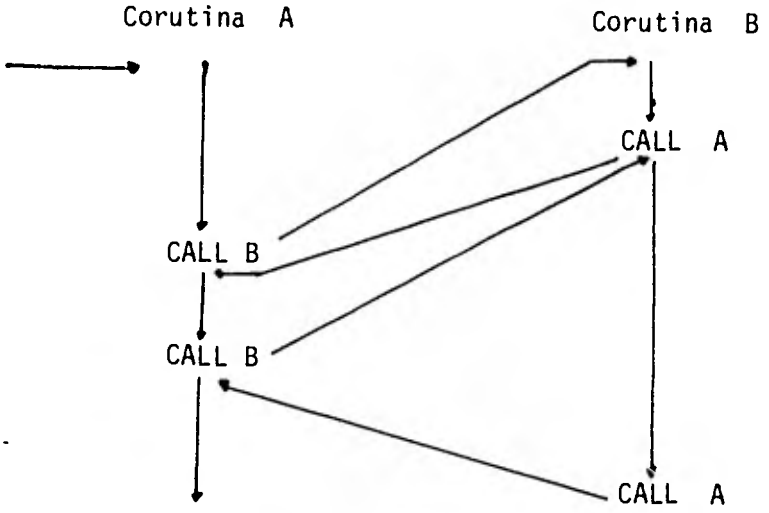
Por ejemplo, ON OVERFLOW CALL SUB.

2. Habilidad de interrumpir el chequeo de la condición. Esto se desea rara vez, por lo general el chequeo continúa a lo largo del programa.

La ventaja de esta estructura de control es que, permite que condiciones extraordinarias sean colocadas fuera del programa principal, sin obscurecer el algoritmo básico. Además, la mayoría de los interruptores usados son peculiares para la prueba del programa. En estos casos, la prueba implícita y las llamadas que involucran pueden ser borradas fácilmente cuando se terminan las pruebas del programa, sin tener que modificar el cuerpo del mismo.

Un subprograma que no se ejecuta completamente antes de regresar el control al programa que lo llamó, se conoce como corutina. Cuando una corutina recibe el control de otro subprograma, ésta se ejecuta sólo parcialmente. La ejecución de la corutina es suspendida cuando regresa el control y en un punto después, la llamada en el programa puede reanudar la ejecución de la corutina en el punto que fue suspendida.

Las corutinas no son comunes en los lenguajes de programación fuera de los lenguajes de simulación discreta. A continuación se ilustra la transferencia de control entre dos corutinas.



3.4 Control de datos

Cuando se escribe un programa, se sabe qué operaciones se van a realizar y en qué orden; pero esto, rara vez es aplicable a los operandos de esas operaciones. Por ejemplo, si en un programa se tiene la proposición --- W = X + Z * 2; una rápida revisión indica que se efectuarán tres operaciones: una multiplicación, una suma y una asignación, en ese orden. ¿Pero qué hay de los operandos para estas operaciones? Un operando claramente es el 2, pero los otros son sólo identificadores W, X y Z que designan operandos. X puede representar un número real, un entero o el nombre de una función que será ejecutada para calcular el operando, o quizá el programador se equivocó y X tiene un valor Booleano o es una etiqueta. En pocas palabras, el problema central del control de datos, es el de saber el significado de X en una proposición de asignación.

3.4.1 Conceptos básicos para el control de datos

El problema central en el control de datos, es conocer el significado de los nombres. La palabra "nombre" generalmente se usa para designar - cualquier expresión utilizada en un programa, para especificar un operando para una operación. Anteriormente se hizo la distinción entre referenciar y accesar variables con índice. Referenciar es la operación que determina cual estructura está asociada con un nombre particular y, accesar es la operación que, dado un índice calcula la localidad del elemento designado en la estructura. El acceso ya fue discutido ampliamente hacer referencia es el tema principal de esta sección.

El control de datos tiene que ver en gran parte con las asociaciones entre identificadores y el programa o datos. Cada asociación puede representarse como una pareja; el identificador y su elemento asociado, o un apuntador al elemento.

Las cinco operaciones básicas de control de datos implicadas con asociaciones de identificadores son:

1. Nombramiento. Es la operación de crear una asociación entre un identificador y un dato o programa.

2. Desnombramiento. Es la operación de destruir una asociación entre un identificador y su objeto asociado.

3. Activación. Es la operación de hacer activa una asociación entre un identificador y un objeto, dejando disponible la asociación para --

usarse en una referencia.

4. Desactivación. Es la operación que deja inactiva una asociación existente.

5. Referenciación. Es la operación de recuperar el dato u objeto del programa, asociado con un identificador dado, usando la única asociación activa para el identificador.

En cualquier punto durante la ejecución de un programa cierta cantidad de identificadores están activos, este conjunto de asociaciones activas se conoce como "medio ambiente de referencia" de ese punto del programa. Siempre que se hace referencia a un identificador, es el medio ambiente de referencia el que determina la asociación apropiada para esa referencia. Las operaciones de nombramiento, desnombramiento, activación y desactivación modifican el medio ambiente de referencia.

Una regla de extensión es aquella que sirve para determinar el medio ambiente de referencia en un programa. Con frecuencia, estas reglas especifican el patrón de activación, desactivación y desnombramiento asociado con una operación de nombramiento, esto es, definiendo los puntos durante la ejecución del programa donde un identificador particular está activo.

Las reglas de extensión se clasifican en dinámicas y estáticas. Dinámicas son aquellas que definen el alcance en términos de ejecución del programa y estáticas son aquellas que definen el alcance en términos de la estructura del programa, al momento de traducción. Las reglas de extensión está

ticas, son características de lenguajes en donde la rápidez de ejecución es importante.

Las referencias a identificadores se clasifican en:

a) Referencias locales. Son aquellas que usan la asociación activa, sólo dentro del bloque o subprograma que se está ejecutando.

b) Referencias globales. Son aquellas que hacen referencia a una asociación activa durante toda la ejecución del programa.

c) Referencias no-locales. Son aquellas que como su nombre lo indica no son locales.

Es conveniente usar el término "medio ambiente local", para designar aquellas asociaciones locales introducidas en el último cambio en el medio ambiente de referencia. El medio ambiente de referencia no-local, es el resto del medio ambiente de referencia en cualquier punto. Normalmente el medio ambiente local de un subprograma contiene los parámetros y variables locales declaradas al principio del subprograma y el medio ambiente no-local contiene los identificadores que son compartidos con otros subprogramas.

3.4.2 Estructura de bloque

El concepto de estructura de bloque es encontrado en lenguajes como ALGOL, PASCAL y PL/I. En un lenguaje estructurado como bloque cada subprograma o programa está organizado como un conjunto de bloques anidados Cada bloque empieza con una serie de declaraciones que sirven para dos propósitos:

1. Cada declaración construye asociaciones para uno o más identificadores. Estas forman el medio ambiente local de referencia para el bloque.

2. Algunas de las declaraciones pueden también definir estructuras de datos o variables simples, que serán creadas en la entrada del bloque.

La estructura de bloque se presta para el uso de reglas de extensión estáticas, permitiendo la producción de código ejecutable más eficiente, además puede ser considerada como una estructura especial de control de datos, operando dentro de subprogramas, que permite cambios del medio ambiente de referencias en puntos arbitrarios durante la ejecución de subprogramas, en lugar de sólo en la entrada y en la salida.

3.4.3 Técnicas para la transmisión de parámetros

La comunicación entre subprogramas a través de parámetros, es más común que la comunicación a través de medio ambientes no-locales. Los parámetros son muy útiles cuando un subprograma va a trabajar con diferentes datos en cada llamada; el uso de medio ambientes no-locales es más apropiado cuando serán empleados los mismos datos en cada llamada.

Se debe distinguir entre parámetro actual (o argumento) y parámetro formal. Un parámetro formal es un identificador usado para nombrar los datos o elementos transmitidos en el subprograma. Los parámetros formales generalmente se específican al principio de la definición del subprograma. Un --

parámetro actual es la expresión usada en la llamada al subprograma, para especificar los datos o elementos que serán transmitidos al subprograma.

La transmisión de parámetros proporciona un mecanismo para permitir que un subprograma accese datos y elementos no-locales del programa, a través de identificadores locales. El uso de parámetros evita la complejidad de los medio-ambientes no-locales.

Existen tres tipos de transmisión de parámetros: transmisión por valor, por referencia y por nombre.

En la transmisión de parámetros por valor, la regla básica es que el parámetro actual se evalua al momento de hacer la llamada al subprograma. El valor del parámetro actual es transmitido al subprograma y es el valor inicial asociado con su correspondiente parámetro formal. La característica que distingue de mejor forma la transmisión por valor, es que la transmisión es sólo de datos al subprograma, en general un subprograma no puede regresar resultados al programa a través de los parámetros que fueron transmitidos por valor, es decir, al regresar al programa los parámetros actuales conservan el valor que tenían antes de entrar al subprograma.

En la transmisión por referencia, se transmite un apuntador a la localidad en donde se encuentra el valor del dato. Cualquier asignación al parámetro formal en el subprograma cambia el valor del parámetro actual al -- regresar al programa donde se efectuó la llamada, esto es, la transmisión por

referencia permite que los datos sean transmitidos a y de subprogramas.

El concepto básico en la transmisión de parámetros por nombre es que los parámetros actuales se transmitirán sin evaluar y el subprograma llamado determina cuando deben ser evaluados, se substituye en cada parámetro -- formal el parámetro actual. También permite transmisión de datos del programa principal y de subprogramas.

3.5 Manejo de almacenamiento

El almacenamiento es uno de los recursos más escasos, en cualquier sistema de cómputo, por lo tanto una de las mayores inquietudes tanto del implementador y diseñador del lenguaje, como del programador es el manejo adecuado del almacenamiento. En esta sección se discutirán algunas técnicas para el manejo de almacenamiento.

Tres aspectos básicos del manejo de almacenamiento son:

1. Asignación inicial . Cualquier sistema para manejar el almacenamiento requiere de técnicas para asignar almacenamiento, durante la ejecución de un programa.

2 . Recuperación . el almacenamiento que ha sido asignado y usado por algún tiempo y que después llega a estar disponible debe ser recuperado por el manejador del almacenamiento para su uso posterior.

3 . Compactación y re-uso . El almacenamiento recuperado puede estar listo inmediatamente para volver a usarse, o puede ser necesaria una

compactación para construir grandes bloques de almacenamiento libre.

La forma más simple de asignación es la "asignación estática", que es hecha durante la traducción y permanece fija durante la ejecución. Generalmente el almacenamiento de los programas del sistema es asignado estáticamente. La asignación estática no requiere manejo de almacenamiento durante el tiempo de ejecución y por supuesto no hay recuperación ni re-uso.

La asignación estática es eficiente porque no se necesita expander el espacio para el manejo de almacenamiento durante la ejecución; sin embargo, es incompatible con llamadas a subprogramas recursivos con estructuras de datos cuyo tamaño depende de la cantidad de datos de entrada.

La técnica más simple para manejo de almacenamiento al momento de ejecución se basa en el uso de stacks. El almacenamiento libre al principio de la ejecución, es construido como un bloque secuencial en memoria. Cuando el almacenamiento es asignado se toman localidades secuenciales en el stack, comenzando desde el final del mismo. El espacio debe ser liberado en orden -- contrario al de asignación. Con esta organización, son triviales los problemas de recuperación, compactación y re-uso de almacenamiento.

Un apuntador al stack es todo lo que se necesita para controlar el manejo de almacenamiento, dicho apuntador siempre apunta a la siguiente -- palabra disponible en el bloque del stack. Todo el almacenamiento usado en el stack está debajo de la localidad apuntada por el "apuntador al stack" y todo

el almacenamiento libre está arriba de ese apuntador. Cuando un bloque de K localidades es asignado simplemente se mueve el apuntador K localidades hacia arriba y cuando se desea liberar un bloque de K localidades se mueve el apuntador K localidades hacia atrás en el stack.

El uso de un stack para activación de subprogramas es muy común en la mayoría de los lenguajes. El manejo de almacenamiento basado en un stack es indudablemente la técnica más usada para manejo de almacenamiento al momento de ejecución.

El tercer tipo básico de manejo de almacenamiento puede ser llamado manejo de almacenamiento amontonado. Un montón ("heap") es un bloque de almacenaje, dentro del cual las piezas son asignadas y liberadas en forma relativamente desordenadas. Con esta técnica los problemas de asignación, recuperación, compactación y reuso son muy serios.

La necesidad de almacenamiento amontonado, se presenta cuando un lenguaje requiere que el almacenamiento sea asignado y liberado en puntos arbitrarios durante la ejecución, así como cuando un lenguaje permite al programador crear, destruir o extender estructuras de datos en puntos arbitrarios del programa.

4 FORTRAN

FORTRAN (Formula Translation) es un lenguaje ampliamente utilizado para resolver problemas científicos, que implican gran cantidad de cálculos numéricos. Las estructuras de este lenguaje son simples y poco elegantes, pero cumplen con el propósito del lenguaje ser eficiente en su ejecución. Las estructuras simples y la eficiencia de ejecución lo han hecho, durante muchos años, un lenguaje estándar para muchas aplicaciones científicas.

La simplicidad de FORTRAN en parte proviene de sus orígenes, ya que fue el primer lenguaje de alto nivel ampliamente utilizado, las primeras versiones fueron diseñadas e implementadas en 1957. En esa época, la utilidad de un lenguaje de alto nivel fue cuestionada, por los programadores en lenguaje ensamblador, en relación a la eficiencia de ejecución de un código compilado de un lenguaje de alto nivel. Como resultado de esto, las primeras versiones de FORTRAN fueron orientadas hacia la eficiencia de ejecución.

Un programa FORTRAN consta de un programa principal y un conjunto de subprogramas, cada uno de los cuales es compilado independientemente y luego deben ser encadenados para posteriormente, poder ejecutar el programa.

4.1 Datos

Los datos elementales en FORTRAN pueden clasificarse en cinco --

tipos: enteros, reales, doble precisión, complejos y Booleanos o lógicos.

Todos los descriptores para las variables simples y arreglos deben ser declarados en el programa. Las variables enteras y enteras pueden declararse en forma implícita o explícita; en forma implícita, el tipo se determina por la primera letra del nombre del identificador para la variable, si ésta es I, J, K, L , M o N, la variable es entera, en otro caso se trata de una variable real.

Las variables de doble precisión, complejas y Booleanas deben declararse en forma explícita y, además, pueden declararse de esta forma variables reales y enteras. Por ejemplo:

```
DOUBLE PRECISION     Q, R
COMPLEX              I
LOGICAL              A, B, C
INTEGER              Z
REAL                 J
```

Los arreglos, en FORTRAN, son estructuras multidimensionales homogéneas, que deben declararse explícitamente con la proposición DIMENSION. En esta declaración se especifica el número de dimensiones del arreglo y el límite superior de cada dimensión. Los límites inferiores son siempre iguales a 1 Por ejemplo, con la declaración

```
DIMENSION  A(5, 7) , K (10)
```

se está declarando un arreglo A de dos dimensiones, con 5 renglones y 7 co-

lumnas; los elementos de este arreglo son reales. Además, se está declarando un vector K con 10 elementos enteros.

Ya que en FORTRAN las áreas de almacenamiento son designadas estáticamente durante la traducción, un arreglo que es usado únicamente en una parte del programa no puede ser destruído y el área de almacenamiento recuperada para crear otro arreglo. Para poder volver a utilizar cierta área de almacenamiento reservado para variables y arreglos, FORTRAN tiene la proposición EQUIVALENCE. Por ejemplo:

EQUIVALENCE (X , Y) esta declaración especifica que las variables X y Y tienen asociada la misma localidad de almacenamiento.

EQUIVALENCE (Z(1,1), K(1)) esta declaración junto con la declaración DIMENSION Z(10,12), K(40) define que el primer elemento de la matriz Z y el primer elemento del vector K comparten área de almacenamiento Asumiendo una representación secuencial de almacenamiento tanto para Z, como para K , se tiene que los 40 elementos de K comparten almacenamiento con los primeros 40 elementos de Z (en FORTRAN el almacenamiento de matrices es por columnas).

En pocas palabras, la proposición EQUIVALENCE permite que variables, dentro de un mismo programa, compartan localidades de almacenamiento.

4.1 Operaciones

Los operadores utilizados, en FORTRAN, se clasifican en aritméticos de relación y lógicos. A continuación se muestran.

Operadores Aritméticos

**	(exponenciación)
*	(multiplicación)
/	(división)
+	(suma)
-	(resta)

Operadores de Relación

.LT.	(menor que)
.LE.	(menor o igual a)
.EQ.	(igual a)
.NE.	(distinto de)
.GE.	(mayor o igual a)
.GT.	(mayor que)

Operadores Lógicos.

.NOT.

.AND.

.OR.

FORTRAN cuenta con gran número de funciones intrínsecas, entre las que se cuentan funciones trigonométricas, logaritmos, raíz cuadrada, valor absoluto, etcétera.

La operación básica para modificar los valores de cualquier variable, es la de asignación. Su sintaxis es como sigue:

⟨variable⟩ = ⟨expresión⟩ | ⟨constante⟩

Otra forma de asignar valores iniciales a las variables es por medio de la proposición DATA. Por ejemplo:

```
DATA  I, J, A, B / 10, 7, 8.5, -3.2 /
```

Con la proposición anterior, se están dando valores iniciales a las variables I, J, A, B . Es equivalente a tener las siguientes proposiciones:

```
I = 10
J = 7
A = 8.5
B = -3.2
```

Las operaciones de entrada-salida se realizan a través de las proposiciones READ y WRITE que tienen muchas variantes. La más común de ellas es especificar el número de archivo que se desea leer o escribir, seguido de una etiqueta que está asociada a una proposición FORMAT; además se debe especificar una lista de variables que se van a leer o a escribir.

La proposición FORMAT sirve para especificar el formato con el que se van a transferir los datos.

Ejemplo:

```
      READ (5, 100) A, B
100   FORMAT (2F4.1)
```

Se está indicando que se van a leer dos variables A y B del archivo 5 (generalmente es un archivo de tarjetas), con el formato 100. El formato 100 indica que se van a leer dos variables reales.

4.3 Control de secuencia

Las proposiciones ejecutables, siempre se realizan en el orden físico en que aparecen dentro de un programa a menos que la secuencia sea explícitamente modificada por una proposición de control. A continuación, se hablará de las proposiciones de control que proporciona FORTRAN.

La más simple de las proposiciones de control es el GO TO incondicional. Su forma es GO TO <etiqueta> y su función es transferir el control a la proposición <etiqueta>, por ejemplo, GO TO 80 transferirá el control a la proposición etiquetada con el número 80.

Una variación del GO TO, llamada GO TO calculado, permite transferir el control a alguna etiqueta dependiendo del valor de una variable entera y tiene la siguiente forma:

GO TO (<etiqueta> $\left\{ , \text{ <etiqueta>} \right\}_{0}^{n-1}$) J

Donde J es un entero que puede tomar valores entre 1 y n. Dependiendo del valor de J, el control se tranfiere a la J-ésima etiqueta.

La tercera forma del GO TO es llamada GO TO asignado y tiene la siguiente forma:

GO TO K (<etiqueta> { , <etiqueta> }$_{0}^{n-1}$)

donde K es un entero cuyo valor debe ser asignado por medio de la proposición ASSIGN y debe ser un número de etiqueta contenida en el GO TO asignado.

Entre las proposiciones condicionales se encuentra el IF aritmético que es de la forma:

IF (<expresión aritmética>) <etiqueta> , <etiqueta> , <etiqueta>

Si la expresión es menor a cero el control se transfiere a la primera etiqueta, si dicho valor es cero se transfiere el control a la segunda etiqueta y si es mayor que cero, a la tercera etiqueta. Por ejemplo, la proposición:

IF (X) 20, 30,10

si el valor de X es negativo el control pasará a la proposición con etiqueta 20; si es cero pasará a la proposición con etiqueta 30 y si es positivo a la proposición con etiqueta 10.

El IF lógico permite la ejecución de una instrucción, siempre y cuando la <expresión lógica> sea verdadera. Su forma es:

IF (<expresión lógica>) <instrucción>

donde la <instrucción> debe ser diferente de la proposicón DO y de otra proposición IF.

La proposición iterativa en FORTRAN tiene la siguiente forma:

DO <etiq> <variable> = <valor inicial> , <valor final> , <incr>

donde la <variable> es un entero que sirve como contador del número de veces que se debe realizar el grupo de instrucciones que están entre el DO y la proposición etiquetada con <etiq> . La <variable> toma como valor inicial -- <valor inicial> y cada vez que termina un ciclo suma a su valor <incr.> , esto se repite hasta que el valor de la <variable> sea mayor o igual al -- <valor final> . Si el <incr.> no se especifica entonces se asumen incrementos de 1 en 1.

Tanto las funciones definidas por el programador, como las funciones intrínsecas son llamadas, dentro de una expresión, con sólo escribir su nombre y entre paréntesis la lista de sus parámetros actuales, por ejemplo: -- X = Y + F (I, J), aquí la función se llama F y sus parámetros son I, J .

Para llamar a subrutinas se utiliza la proposición CALL, seguida del nombre de la subrutina y su lista de parámetros actuales, Ejemplo: --- CALL SUBR (X, Y).

Los subprogramas (subrutinas y funciones) regresan el control al punto donde fueron llamadas, una vez que aparece la proposición RETURN y el programa principal termina su ejecución al encontrar la proposición STOP.

4.4 Control de datos

En FORTRAN, todas las variables son locales al subprograma en que se declaran, sin embargo, existe la proposición COMMON que permite que variables en el programa principal y sus subprogramas compartan localidades de almacenamiento, es decir, permite hacer globales ciertas variables.

La transmisión de parámetros en FORTRAN es únicamente por referencia. Los parámetros actuales pueden ser variables simples, arreglos, funciones, subrutinas o expresiones.

4.5 Manejo de almacenamiento

Un programa FORTRAN consta de un programa principal y un conjunto de subprogramas, cada uno de los cuales es compilado independientemente de los otros y luego deben encadenarse para poder ejecutar el programa, después. Cada subprograma es compilado, en un área asignada estáticamente que contiene el código ejecutable del subprograma o programa y áreas para datos definidos por el programador y por el sistema. No se proporciona manejo de almacenamiento - al momento de ejecución, es decir, todas las estructuras de datos son creadas y almacenadas durante la traducción y únicamente pueden cambiar sus valores - durante la ejecución.

5 COBOL

La palabra COBOL es una abreviatura de Common Business Oriented Language, que quiere decir lenguaje común orientado a las aplicaciones comerciales; fue diseñado con el fin de servir a la administración, de tal suerte que la mayoría de los fabricantes de equipos de computadora, han implementado en sus máquinas compiladores de COBOL.

La universalidad de COBOL, permite además gran flexibilidad. Una compañía puede usar computadoras de distintas manufacturas, mientras tengan un lenguaje de programación en común. De igual manera, la conversión de un modelo de computadora a otro más nuevo no presenta gran problema.

Desde su creación en 1959, el lenguaje COBOL ha experimentado muchos refinamientos para hacerlo más estándar. La American National Standards Institute (ANSI), es una asociación de manufactureros y usuarios de computadoras, que ha desarrollado un COBOL estándar llamado COBOL ANS.

COBOL definitivamente no es un lenguaje breve, su objetivo es ser natural, donde natural significa ser como el Inglés. COBOL no permite un mínimo de escritura, al contrario pero el beneficio ganado con esto es la facilidad para leer y entender un programa al verlo sin necesidad de ser un experto en sistemas de cómputo.

La estructura mínima de todo programa en COBOL, consta de cuatro grandes divisiones:

IDENTIFICATION DIVISION (División de Identificación). Esta división sirve para identificar el programa a la computadora, provee información útil para la documentación de un programa, la cual puede ser leída fácilmente por cualquier persona que no se dedique al procesamiento de datos.

ENVIRONMENT DIVISION (División del Medio Ambiente). Sirve para definir los recursos físicos necesarios para el proceso; tales como: la configuración del equipo con el que se va a trabajar y los dispositivos que se necesitan en la ejecución del programa.

DATA DIVISION (División de Datos). Su función es describir los formatos de la información de entrada y salida que son procesados por el programa, así como también, definir cualquier constante o área intermedia de trabajo necesaria para procesar los datos.

PROCEDURE DIVISION (División de Procedimientos). Su función es describir la lógica del programa, por lo tanto, contiene todas las órdenes (instrucciones) que llevan a la solución del problema.

Los objetivos de trabajar en secciones son por un lado, separar las partes del programa dependientes de la máquina (ENVIRONMENT) y las independientes, y por otro, se pretende tener a la definición de datos y varia-

bles separadas de los procedimientos.

5.1 Datos

Cada dato usado en un programa COBOL debe ser explícitamente declarado en la DATA DIVISION. Esta división está compuesta por dos secciones, la primera es la sección de archivos (FILE SECTION) que se encarga de describir completamente las áreas de almacenamiento que se están utilizando para los archivos de entrada y/o salida y la segunda sección de almacenamiento de áreas de trabajo para todos aquellos campos que no pertenecen a la entrada y salida, pero que son necesarios para el proceso "intermedio" de la información ---- (WORKING-STORAGE SECTION).

Todos los datos en COBOL son almacenados como cuerdas de caracte - res, al momento de ejecución, con excepción de aquellos que sean declarados con la claúsula USAGE IS COMPUTATIONAL. Esta representación en cuerdas de caracteres tiene dos propósitos importantes:

1 . Ya que COBOL es un lenguaje orientado hacia aplicaciones con grandes cantidades de entrada salida, permite que los datos se almacenen en memoria central en una forma que puedan ser transmitidos directamente a un archivo externo sin tener que efectuar ninguna conversión. Los números, en particular, no son convertidos automáticamente a una representación binaria, en la entrada, en lugar de eso son converitdos sólo cuando se utilizan como operandos en alguna operación aritmética.

2 . La representación en cuerdas de caracteres permite que las

descripciones de datos sean casi siempre independientes de las características particulares del hardware de la máquina, en que se este trabajando, tales como, la longitud de palabra o representación numérica. Como resultado de esto los programas en COBOL son relativamente faciles de transportar de una máquina a otra.

COBOL proporciona cinco tipos de datos elementales: enteros, reales, de doble precisión, alfabéticos y alfanuméricos.

La declaración para una variable simple (o un elemento de una estructura) se especifica por medio de la claúsula PICTURE. La forma más simple de la claúsula PICTURE especifica unicamente la cantidad de caracteres ocupados por el dato, así como su tipo. Por ejemplo, en la declaración

```
77  VARIABLE  PICTURE  X (10).
```

VARIABLE es el nombre de la variable; 77 indica que es una variable simple y X(10) especifica que se trata de una variable alfanumérica cuya longitud son 10 caracteres.

Otro ejemplo sería:

```
77  SUELDO  PICTURE  9(4) V 99
```

Aquí el nombre de la variable es SUELDO y el PICTURE indica que se trata de una variable compuesta por 6 dígitos (esto es por los 9's) con un punto decimal posicionado entre el cuarto y quinto dígito (especificado por la V).

Para la salida se necesita un formato más elaborado, por ejemplo,

TOTAL PICTURE $$, $$$, $$9.99

El valor de TOTAL es un "número editado", será un número menor que 10 millones con dos dígitos a la derecha del punto decimal. El punto decimal debe ser explícitamente insertado entre el séptimo y octavo dígito, todos los ceros a la izquierda del último dígito significativo serán suprimidos y reemplazados por blancos, las comas serán insertadas en las posiciones usuales cuando se necesiten y un signo de pesos ($) será impreso inmediatamente a la izquierda del primer dígito significativo del número.

Los tres ejemplos anteriores ilustran algunas de las posibilidades para la construcción de PICTURE's.

La declaración de una variable o elemento de una estructura de datos puede contener otras claúsulas, además de la claúsula PICTURE. La claúsula VALUE puede ser usada para definir un valor inicial para la variable; la claúsula JUSTIFIED determina si el valor de la cuerda de caracteres será justificada a la derecha o a la izquierda, cuando el valor del campo ocupa menos posiciones que las declaradas; la claúsula USAGE IS especifíca la forma en que los datos serán almacenados dentro de la computadora.

La estructura de datos básica en COBOL es el arreglo heterogéneo multidimensional llamado un registro. Dado que todos los registros usados en un programa son declarados en la DATA DIVISION, entonces se hace un chequeo de tipo estático, el almacenamiento es relativamente eficiente y es posible accesar los elementos del registro. Los registros son declarados usando un for

mato de niveles anidados, cada uno con un número de nivel como se muestra a continuación:

```
01  EMPLEADO .
    02  NOMBRE-EMPLEADO .
        03  NOMBRE      PIC A (10).
        03  A-PATERNO   PIC A (10).
        03  A-MATERNO   PIC A (10).
    02  EDAD            PIC 99.
    02 SALARIO           PIC  9 (5) V 99.
```

Los registros pueden contener arreglos homogéneos (llamados tablas) de una a tres dimensiones; por ejemplo: si se desea construir un vector de diez elementos, cada uno de 5 dígitos, la declaración sería:

```
01  ARREGLO.
    02 A  OCCURS 10 TIMES  PICTURE 9(5).
```

ARREGLO es el nombre del vector; los elementos individuales serán referenciados por A(1), A(2), A(3), ..., A(10).

Si se desea construir un arreglo tridimensional homogéneo de 20 X 30 X 10, cuyos elementos sean enteros de cuatro dígitos, se declararía de la siguiente manera:

```
01  ARREGLO-TRI-DIMENSIONAL.
    02  PLANO  OCCURS  20  TIMES.
        03  RENGLON  OCCURS  30  TIMES.
            04  ELEMENTO  OCCURS  10  TIMES  PICTURE  9(4).
```

Con una declaración de este tipo se pueden hacer referencias del siguiente tipo ELEMENTO (7,21,3)

También , existen registros en los que sus componentes son una mezcla de homogéneos y heterogéneos. Por ejemplo:

```
01  DATOS-DE-POBLACION.
    02  TOTAL-POLACION-MEX      PIC  9(10).
    02  ESTADO OCCURS  32 TIMES.
        03  NOMBRE-ESTADO       PIC  X(30).
        03  POB-ESTADO          PIC  9(9).
        03  NOMBRE-CAPITAL      PIC  X(30).
        03  POB-CAPITAL         PIC  9(8).
```

Este vector tiene un dato elemental TOTAL-POBLACION-MEX y un vector ESTADO de 32 elementos, cada uno de los cuales es un arreglo heterogéneo de cuatro elementos.

El concepto de redefinición del área de almacenamiento es importante en COBOL. La claúsula REDEFINES sirve para redefinir una misma área de memoria, es decir, permite utilizar una misma área con diferentes formatos para accesar la información que contiene. Por ejemplo, la declaración:

```
01  ARR.
    02  A  OCCURS  10 TIMES  PICTURE  9 (5).
```

Reserva un área de 50 caracteres estructurada en un vector, que

contiene diez enteros de cinco dígitos cada una. Si esta declaración va inmediatamente seguida por la siguiente declaración

```
01  ARR-REDEFINIDO  REDEFINES  ARR.
    02  P        PICTURE  9 (5).
    02  C  OCCURS 5  TIMES.
        03  D    PICTURE  X (4).
        03  E    PICTURE  9 (5).
```

entonces el mismo bloque de 50 caracteres será usado por el registro ARR-REDEFINIDO.

5.2 Operaciones

La secuencia de operaciones que se deben ejecutar en un programa COBOL se define en la PROCEDURE DIVISION.

Las operaciones aritméticas de suma, resta, multiplicación y división pueden escribirse como proposiciones separadas; por ejemplo:

```
ADD  A, B, C  GIVING D
SUBTRACT  B  FROM  A
MULTIPLY  A  BY  B  GIVING  C  ROUNDED
DIVIDE  P  BY  Q  GIVING  R  REMAINDER  S.
```

O en la mayoría de los casos, usando la proposición COMPUTE, ejemplo:

```
COMPUTE  A = A + B * C
```

Las operaciones de relación se realizan utilizando los siguientes

operadores:

LESS THAN	(menor que)
NOT GREATER THAN	(menor o igual a)
EQUAL	(igual a)
NOT EQUAL	(distinto de)
NOT LESS THAN	(mayor o igual a)
GREATER THAN	(mayor que)

Además, de los siguiente operadores lógicos:

AND

OR

NOT

La primitiva utilizada para asignar datos a un campo dado es el operador MOVE. La proposición:

MOVE A TO b

asigna el valor de A a la variable B.

Existe gran cantidad de primitivas relacionadas con las operaciones de entrada-salida. Las operaciones básicas en archivos externos son OPEN, READ, WRITE y CLOSE. Las operaciones ACCEPT y DISPLAY son utilizadas para entrada y salida a dispositivos, tales como la consola de operación.

Como las operaciones de ordenación son muy comunes. La mayoría de las implementaciones de COBOL incluyen una primitiva SORT que permite que

un archivo sea ordenado usando una llave dada.

Otros tipos de primitivas, son porporcionadas por COBOL. Una proposición SEARCH permite una búsqueda eficiente de un elemento, que satisface una condición o condiciones especificadas dentro de un arreglo homogéneo. Algunas primitivas simples son proposiciones que permiten examinar el contenido de una cuerda; ya sea para contar el número de veces que aparece una subcuerda dada o reemplazarla por otra subcuerda. Existe un módulo opcional llamado escritor de reportes (REPORT WRITER), el cual como su nombre lo indica permite generar reportes en forma muy eficiente; ya que proporciona un conjunto de primitivas para formatear las páginas de salida, en lugar de trabajar con líneas simples como se hace con el WRITE.

5.3 Control de secuencia

La ejecución de un programa sigue la secuencia física de las proposiciones escritas en la PROCEDURE DIVISION, a menos que se use una proposición GO TO, IF ó PERFORM, para transferir el control a otra parte del programa.

Con un GO TO simple, se puede transferir el control a cualquier parráfo o sección etiquetada. Una variante del GO TO tiene la siguiente forma:

GO TO <etiqueta> $\left\{, \text{<etiqueta>}\right\}_{o}^{n}$ DEPENDING ON <identificador>

y el control es transferido a la i-ésima etiqueta si el valor del identifica-

dor es i. La etiqueta especificada en un GO TO simple dentro de un parráfo puede ser modificada al momento de ejecución por la proposición ALTER ; por ejemplo:

ALTER <etiqueta-1> TO PROCEED TO <etiqueta-n>

donde <etiqueta-1> es la etiqueta del parráfo que contiene el GO TO que será modificado.

La proposición condicional IF tiene la siguiente forma:

IF <condición> <proposiciones 1> ELSE <proposiciones 2>

Si la condición se cumple entonces se ejecutan las <proposiciones 1> y se ignoran las <proposiciones 2> en caso contrario, es decir, si la condición no se satisface se ejecutan las <proposiciones 2> y se ignoran las <proposiciones 1> .

La proposición PERFORM sirve como proposición de iteración y como llamada a subprogramas sin parámetros. En sus formas más simples

PERFORM <etiqueta 1> , o bien,

PERFORM <etiqueta 1> THRU <etiqueta 2>

El control es transferido al parráfo etiquetado con <etiqueta 1> ; las proposiciones en este parráfo o en los parráfos siguientes hasta el parráfo <etiqueta 2> , serán ejecutadas y el control regresa a la proposición inmediatamente después del PERFORM. Aquí no se involucran parámetros o variables locales, sólo ocurre una transferencia de control.

La proposición:

PERFORM <etiqueta1> THRU <etiqueta2> K TIMES

sirve para que se llame al subprograma K veces.

La proposición :

PERFORM <etiqueta1> THRU <etiqueta2> UNTIL <condición>

ejecuta el subprograma varias veces hasta que la condición evaluada sea cierta

La proposición :

PERFORM <etiqueta1> THRU <etiqueta2> VARYING <índice> FROM <valor inicial> BY <incremento> UNTIL <condición 1>
AFTER <índice2> FROM <inicio2> BY <incremento2> UNTIL
<condición2> AFTER <índice3> FROM <inicio3> BY <incremento3>
UNTIL <Condición 3>

permite la ejecución iterativa de subprogramas moviendo de 1 a 3 índices dentro de rangos enteros.

Existen otras estructuras para el control de secuencia que permiten la ejecución de una o más proposiciones, cuando se cumple una condición especial durante la ejecución de una instrucción; por ejemplo, la proposición:

ADD A TO B ON SIZE ERROR <proposiciones>

causa que se ejecuten las <proposiciones> en caso que el valor de la suma exceda el espacio permitido para B. Otras condiciones de chequeo especial incluyen el checar fin de archivo en las proposiciones READ y checar fin de página en las proposiciones WRITE.

5.4 Control de datos

El mecanismo de control de datos en COBOL es muy primitivo. La DATA DIVISION de un programa construye un único medio ambiente de referencias globales, el cual es usado a través de la PROCEDURE DIVISION para referenciación. Los únicos identificadores que no aparecen en la DATA DIVISION son las etiquetas, pero no pueden ser hechas locales; ya que implícitamente son globales.

En versiones de COBOL desde 1968, se permiten subprogramas (escritos como programas completos de COBOL) que son compilados en forma separada, tienen parámetros y su propio medio ambiente de referencia local (definido por la DATA DIVISION de cada subprograma). Todas las declaraciones para los parámetros formales deben ser proporcionadas en la LINKAGE SECTION de la DATA -- DIVISION del subprograma.

5.5 Manejo de almacenamiento

El diseño de COBOL no requiere manejo de almacenamiento al momento de ejecución, es decir, proporciona un tipo de almacenamiento estático. Este lenguaje está diseñado para permitir la producción de un código ejecutable eficiente para eso, son necesarias las declaraciones para todos los datos utilizados en un programa, haciendo que el manejo de almacenamiento permanezca - estático al momento de ejecución y lo único que varie sean los valores de los datos.

6 ALGOL

ALGOL (Algorithmic Language) fue diseñado por un cómite internacional a fines de la década de los 50's y principios de los 60's culminando en 1963 con un "Reporte Revisado del Lenguaje para Algoritmos ALGOL 60".

La definición de ALGOL fue un evento clave en la historia de los lenguajes de programación, ya que ha tenido mucha influencia en el diseño y definición de lenguajes, a pesar de no ser un lenguaje ampliamente utilizado para trabajos prácticos en Estados Unidos, sin embargo, es muy utilizado en Europa.

ALGOL, generalmente es clasificado como un lenguaje para cálculos científicos, por tener datos numéricos y estructuras de datos homogéneas. Sin embargo, es muy utilizado en otras áreas por la claridad y elegancia de sus estructuras.

Un programa en ALGOL está compuesto de un programa principal y un conjunto de subprogramas constituidos por bloques. Un bloque está compuesto de un conjunto de declaraciones seguido de una serie de proposiciones, todo esto limitado por un BEGIN al principio del bloque y un END al final del mismo. El programa principal es simplemente un bloque. Cada subprograma está compuesto de un encabezado, en el que se especifica el nombre del mismo, los nombres y tipos de cada uno de los parámetros formales y un cuerpo que en general es

un bloque.

6.1 Datos

Los datos elementales permitidos en ALGOL son de tipo real (REAL) entero (INTEGER), de doble precisión (DOUBLE), Booleanos (BOOLEAN) y alfabéticos (ALPHA). Como datos estructurados únicamente se tienen arreglos multidimensionales homogéneos.

Todos los identificadores para variables simples y arreglos deben ser declarados al principio del bloque donde serán utilizados. Los arreglos pueden tener cualquier número de dimensiones y tanto el límite superior como el inferior, para cada dimensión, deben ser especificados. El tamaño de los arreglos puede definirse hasta el momento de ejecución, pero una vez creado el arreglo, su tamaño permanece fijo hasta que el arreglo se destruye. Por ejemplo, la declaración:

```
INTEGER ARRAY  X [-2:5 , 1:M]
```

define un arreglo bi-dimensional de números enteros, llamado X. X tiene 8 renglones que pueden ser referenciados por -2, -1, 0, 1, 2,3,4, 5 y M columnas donde M se determina al momento de ejecución; las columnas pueden ser referenciadas por 1,2,3, ... , M-1, M.

Generalmente los arreglos y variables simples son creados al momento de entrar al bloque donde se declaran y son destruídos al salir de dicho bloque. Cuando la declaración de una variable o de un arreglo está precedi

da de la palabra OWN, la estructura (variable o arreglo) es creada al entrar por primera vez, al bloque en que se declara, pero no es destruida a la salida permitiendo que al volver a entrar a dicho bloque se conserven los valores que tenía la última vez que se salió del bloque.

6.2 Operaciones

Las operaciones básicas en ALGOL se realizan haciendo uso de los siguientes operadores:

a) Operadores Aritméticos

**	(exponenciación)
*	(multiplicación)
/	(división)
+	(suma)
-	(resta)

b) Operadores de Relación

LSS	<	(menor que)
LEQ		(menor o igual a)
EQL	=	(igual a)
NEQ		(distinto de)
GEQ		(mayor o igual que)
GTR	>	(mayor que)

c) Operadores Lógicos

AND

OR

NOT

Además, se cuenta con un conjunto de funciones estándar, como son las funciones trigonométricas (SIN, COS, etcétera), raíz cuadrada (SQRT), etc.

La operación de asignación está representada por el símbolo :=

Para poder manipular cuerdas de caracteres, es necesario que éstas se encuentren almacenadas en vectores (si están en un arreglo multidimensional, los renglones del arreglo se consideran como vectores), además de, utilizar un tipo especial de variables llamadas "pointer" para apuntar a un sólo caracter (byte) de la cuerda.

Las operaciones permitidas con las cuerdas de caracteres son las siguientes:

a) Reemplazar una cuerda por otra

b) Examinar el contenido de una cuerda

c) Obtener el valor numérico de una cuerda de caracteres

ALGOL proporciona un mecanismo llamado, "palabras parciales", mediante el cual se pueden referenciar, directamente, los bits de cualquier palabra que se desee.

Otro tipo de operaciones permitidas en ALGOL, son aquellas que define el programador usando la proposición DEFINE. Ejemplo:

```
DEFINE
    REEMPLAZAR (A, N) = REPLACE P : POINTER (A) BY Q FOR N #;
```

Con la proposición anterior, se está definiendo una función llamada REEMPLAZAR que tiene dos parámetros, y lo que hace es reemplazar el contenido del vector apuntado por P por una cuerda Q durante N caracteres.

ALGOL permite, también la creación de subprogramas en el sentido usual, o funciones que regresen un valor numérico o Booleano y se permite -- gran variedad de parámetros.

6.3 Control de secuencia

ALGOL proporciona varias estructuras, para controlar la secuencia dentro de un programa, tanto a nivel de expresiones como de proposiciones. El control de subprogramas, se restringe a llamadas a subprogramas que pueden ser recursivos.

En ALGOL existen tres tipos de expresiones: aritméticas, Booleanas y designacionales.

Las expresiones aritméticas están compuestas de variables simples o con índice, constantes reales o enteras, llamadas a funciones y los operadores aritméticos antes mencionados. Se emplea la jerarquía usual de las funcio

nes con asociatividad de izquierda a derecha y el uso de paréntesis para control explícito, cuando es necesario. Una característica de las proposiciones aritméticas en ALGOL es el uso de condicionales. Por ejemplo:

```
X := Y + (IF A = B  THEN  A + 1  ELSE A) * B
```

es una expresión válida, que toma el valor de Y + (A + 1) * B cuando A = B, en caso contrario su valor es Y + A * B

Las expresiones Booleanas son similares a las expresiones aritméticas, excepto que éstas involucran variables y constantes Booleanas y además los operadores de relación y Booleanos. Las expresiones Booleanos pueden, también, contener condicionales; por ejemplo:

```
B :=  (IF  X>0  THEN  C  ELSE  D)  OR  E
```

donde todas las variables, excepto X deben ser Booleanas.

Las expresiones designacionales son aquellas cuyo valor es una etiqueta. Estas expresiones únicamente pueden construirse en un GO TO, un SWITH o en un IF ... THEN ... ELSE, como se verá más adelante.

A continuación, se hablará de las proposiciones que tiene ALGOL para controlar la secuencia.

Las proposiciones etiquetadas en ALGOL son aquellas que están precedidas por la etiqueta seguida de dos puntos (":"), así, la instrucción GO TO L1, transferirá el control a la proposición que empiece con L1:.

Una variación de la transferencia de control básica (GO TO) es conocida como switch. Un switch es un vector de etiquetas, su declaración al principio de un bloque especifica las etiquetas que contiene. Por ejemplo, la declaración:

SWITH S := L1, L2, SIGUE, ALTO;

al principio de un bloque construye un vector S con cuatro valores, las etiquetas L1, L2 , SIGUE y ALTO. Una proposición de la forma GO TO S [3] transferirá el control a la proposición que tiene etiqueta SIGUE.

El contenido de un "switch" no puede ser modificado al momento de ejecución.

Otra variación del GO TO simple es aquella que permite el uso de proposiciones designacionales. Por ejemplo:

GO TO IF X = Y THEN L1 ELSE IF X > Y THEN L2 ELSE L3

permite que el control sea transferido a L1 si X= Y; a L2 si X > Y, o bien a L3 si X < Y.

El uso de proposiciones designacionales se extiende también a los switch's, por ejemplo:

SWITCH S:= L1, IF X = Y THEN L2 ELSE L3, SIGUE, ALTO

En este caso, la instrucción GO TO S [2] transferirá el control a L2 si X = Y o bien, a L3 si X ≠ Y.

Las proposiciones condicionales en ALGOL, tienen la siguiente forma

IF <expresión Booleana> THEN <proposición 1> ELSE <proposición 2>

Si la <expresión Booleana> es verdadera se ejecuta la <proposición 1> y se ignora la <proposición 2> en caso contrario, se ejecuta la <proposición 2> y se ignora la <proposición 1> .

La proposición CASE es una generalización de la proposición IF. La proposición IF permite que se ejecute una de dos acciones y la proposición CASE permite que se ejecute una de varias acciones dependiendo del valor de una <expresión aritmética>; su sintaxis es la siguiente:

```
CASE <expresión aritmética> OF
BEGIN
<proposición 0> ;
<proposición 1> ;
        .
        .
        .
<proposición n> ;
END;
```

Si el valor de la <expresión aritmética> es I, entonces ejecuta únicamente la proposición I-ésima. Si la <expresión aritmética> es negativa o mayor que n ocurrirá un error.

Las proposiciones de iteración pueden tomar cualquiera de las siguientes formas:

a) DO <proposición> UNTIL <expresión Booleana>

Ejecuta la <proposición> al menos una vez hasta que <expresión Booleana> sea verdadera.

b) WHILE <expresión Booleana> DO <proposición>.

Esta es similar a la anterior, excepto que la <proposición Booleana> se evalúa antes de ejecutar la <proposición> .

c) THRU <expresión aritmética> DO <proposición>.

La <expresión aritmética> se evalúa y redondea para obtener un entero, que es el número de veces que se va a ejecutar la <proposición>

d) FOR <variable> := <lista de valores> DO <proposición>.

Ejecuta la <proposición> tantas veces como valores tenga la <lista de valores> . Cada vez, la <variable> toma un valor de la <lista de valores> . Por ejemplo, la instrucción

FOR I := 2, 3, 5,8,11, 20 DO <Proposición> .

ejecuta 6 veces la <proposición> , una para cada valor de I

e) FOR <variable> := <expresión> WHILE <expresion Booleana>
DO <Proposición>

Inicializa el valor de la <variable> y realiza la <proposición> mientras la <expresión Booleana> es verdadera.

f) FOR <variable> := <valor inicial> STEP <incremento>
UNTIL <valor final> DO <proposición> .

La ⟨proposición⟩ es ejecutada para cada valor que toma la ⟨variable⟩ desde que tome el ⟨valor inicial⟩ hasta que sea mayor o igual al ⟨valor final⟩ . Cada vez que ejecuta la ⟨proposición⟩ el valor de la ⟨variable⟩ se hace igual al valor que tiene más el ⟨incremento⟩ .

ALGOL permite únicamente llamadas a subprogramas que pueden ser simples o recursivos. El retorno de un subprograma al programa que lo llamó, se lleva a cabo hasta que se llega al final del subprograma, es decir, no existe una instrucción explícita para retornos de subrutinas.

Un subprograma puede ser llamado como una función dentro de una expresión o como una subrutina con sólo escribir,como una instrucción, el nombre del subprograma (PROCEDURE o DEFINE) y la lista de los parámetros actuales por ejemplo: SUBPROGR (A, B, C).

6.4 Control de datos

ALGOL proporciona dos técnicas para la transmisión de parámetros: por valor o por nombre. La técnica usada se especifica listando los parámetros que serán transmitidos por valor a continuación del encabezado del procedimiento y precedidos por la palabra VALUE después se listan todos los argumentos -- para especificar su tipo, los que no aparecen en la primera lista serán transmitidos por nombre. Por ejemplo:

```
PROCEDURE SUMA (V, N);
VALUE N;
INTEGER N;
ARRAY V [*] ;
```

En el ejemplo anterior, el procedimiento A tiene dos parámetros V y N , donde N es un entero que será transmitido por valor y V es un vector de números reales que se trasmitirá por nombre.

Los parámetros actuales pueden ser constantes numéricas o Booleanas, arreglos, variables simples, nombres de subprogramas, etiquetas, cuerdas de caracteres o expresiones. Cuando el parámetro actual es una expresión y debe transmitirse por nombre, se transmite sin evaluar.

Otra técnica para control de datos utilizada por ALGOL es su estructura de bloque. Cada bloque o subprograma introduce un conjunto de identificadores y define sus asociaciones, las cuales tienen efecto a lo largo de la ejecución del bloque o subprograma y son destruidas a la salida del mismo. Con las entradas y salidas dinámicas de los bloques y procedimientos, durante la ejecución, se controla que estén vigentes las asociaciones de los identificadores pero la estructura de bloque estática (al momento de compilación) controla las asociaciones que son usadas para resolver referencias a identificadores no-locales.

6.5 Manejo de almacenamiento

El manejo de almacenamiento en una implementación de ALGOL, se basa en un área estática de almacenamiento y un stack central asignado dinámicamente. Los programas son traducidos en bloques de código ejecutable y almacenados estáticamente, antes de que empiece la ejecución. Parte de esta área estática de almacenamiento contiene el código de rutinas necesaria durante la ejecución, por ejemplo de funciones estándares, como son el Seno y la Raíz cuadrada.

El área de almacenamiento reservado por un stack al momento de ejecución es utilizada para activar registros asociados con los bloques y subprogramas. De esta forma, cuando se entra a un bloque o subprograma se crea un registro de activación y el almacenamiento se asigna en el tope del stack. -- Cuando se sale del bloque o subprograma se libera el espacio asignado por el registro de activación correspondiente..

En particular, el registro de activación contiene las asociaciones para los identificadores locales, almacenamiento temporal para la evaluación de expresiones, almacenamiento para descriptores, y valores de arreglos y variables además de almacenamiento para las listas de parámetros durante la transmisión.

7 PASCAL

El lenguaje PASCAL, es el resultado de muchos años de trabajo para definir un sucesor de ALGOL 60. Estos esfuerzos producen el lenguaje --- ALGOL-W, el cual puede ser considerado como el predecesor directo de PASCAL. La principal innovación en PASCAL fue una variedad de métodos para estructurar los datos, y en particular un tipo de datos definido por el programador. El uso de PASCAL tiende a decrementar los esfuerzos de programación e incrementar la facilidad de lectura de un programa.

Un programa en PASCAL se divide en un encabezado y un cuerpo llamado bloque. El encabezado da al programa un nombre y lista sus parámetros (archivos) a través de los cuales el programa se comunica con el medio ambiente.

7.1 Datos

Los datos en PASCAL se dividen en estáticos y dinámicos. Dentro de los estáticos están los datos simples y los estructurados.

Cada variable tiene un tipo asociado a ella, el cual determina los valores que puede tomar y las operaciones que pueden realizarse con ellas. Las literales y constantes también tienen tipo, pero es determinado por el compilador dependiendo del valor que tengan; sin embargo, cuando se declara una varia

ble se debe especificar su tipo.

Los tipos estándar en PASCAL son enteros, reales, Booleanos y caracteres. A continuación se muestran algunos ejemplos:

```
CONS
    LIMITE  = 2000;
    BLANCO  = ' ' ;
VAR
    CONT, IND   : INTEGER   ;
    RAIZ        : REAL      ;
    FIN         : BOOLEAN   ;
    LETRA       : CHAR      ;
```

Con las intrucciones anteriores, se declararon dos constantes: LIMITE que es entera y tiene valor de 2000 y BLANCO que es un caracter igual a un espacio en blanco. Además se tienen dos variables enteras (CONT e IND), una real (RAIZ), una Booleana (FIN) y otra de tipo caracter (LETRA).

Además, de los ya mencionados tipos estándar para datos elementales, en PASCAL, el programador puede definir otros tipos de datos, simplemente haciendo una lista de los valores que pueden ser asumidos por una variable de ese tipo. Las variables declaradas en esta forma se llaman escalares. Por ejemplo, la declaración:

```
TYPE DIAS (LUN, MAR, MIE, JUE, VIE, SAB, DOM)
```

indica que las variables del tipo DIAS sólo pueden tomar los valores que están

dentro del paréntesis. Dichos valores son constantes.

El tipo DIAS, puede ser usado en la declaración de variables de la misma forma que los tipos estándar. Ejemplo:

```
VAR  DLABORALES, DDESCANSO  : DIAS
```

Otro tipo de datos, definido por el programador, puede ser un subrango de un escalar definido con anterioridad, un subrango de enteros o un subrango de caracteres. Para definir subrangos lo único que hay que hacer es especificar los límites inferior y superior. Ejemplo:

```
TYPE  INDICE  = 1 . . 20;
      DLAB    = LUN . . VIE;
```

Con estas declaraciones se está especificando que las variables del tipo INDICE sólo pueden tomar valores entre 1 y 20, y que las variables del tipo DLAB sólo pueden tomar valores entre LUN y VIE.

Entre los datos estructurados de PASCAL se encuentran los arreglos (ARRAY), registros (RECORD), conjuntos (SET) y archivos secuenciales (FILE).

Los arreglos en PASCAL son multidimensionales homogéneos. Los componentes de un arreglo son almacenados en palabras consecutivas de memoria. Esto es eficiente para almacenar componentes reales o enteros; sin embargo, para almacenar variables de otro tipo no siempre es eficiente y se desperdicia almacenamiento, como es el caso de las cuerdas de caracteres; ya que al almacenar

un caracter por palabra se desperdicia gran parte de la palabra, en PASCAL este problema se resuelve declarando un arreglo empacado (PACKED ARRAY), de esta forma cada elemento del arreglo ocupa únicamente un byte.

Los arreglos pueden tener cualquier número de dimensiones y cada dimensión puede tener un límite arbitrario, pero constante, es decir, no existen los arreglos dinámicos en PASCAL.

Los registros al igual que los arreglos son variables estructuradas, pero la diferencia es que los componentes de un registro pueden tener distintos tipos y que son accesados por su nombre, no por un índice. Por ejemplo:

```
TYPE PERSONA = RECORD
               NOMBRE   : PACKED ARRAY [1 .. 30] OF CHAR;
               EDAD     :   18 .. 35  ;
               ESTATURA : REAL ;
               SEXO     : (MASC, FEM)
               END;
VAR  DATO  :  PERSONA;
```

Con las instrucciones anteriores se ha declarado una variable DATO del tipo PERSONA, es decir, va a ser un registro con cuatro campos NOMBRE, EDAD ESTATURA y SEXO.

Para referenciar cualquier componente del registro, el nombre del registro va seguido por un punto y luego el nombre del campo. Por ejemplo:
DATO.EDAD

Los registros en PASCAL puede ser de longitud fija o variable. En caso de ser de longitud variable, primero deben especificarse los campos que permanecerán fijos en el registro, seguidos de los campos que pueden aparecer o no, en el registro dependiendo de cierta condición.

Un SET define el conjunto de todos los subconjuntos de valores del tipo base incluyendo el conjunto vacío. El tipo base, puede ser un escalar o un subrango. Por ejemplo:

```
TYPE PRIMARIO = (ROJO, AMARILLO, AZUL);
     COLOR    = SET OF PRIMARIO;
VAR
     CONJ     : COLOR;
```

Los valores que puede tomar la variable CONJ son los subconjuntos que se puedan formar con los valores del escalar PRIMARIO, incluyendo al conjunto vacío, por ejemplo:

```
CONJ := [AZUL, ROJO] ;
CONJ := [ ]
```

En PASCAL, un registro puede ser generado dinámicamente usando un procedimiento llamado NEW, cuyo parámetro debe ser un "pointer" (que no es otra cosa que la dirección de almacenamiento del último registro creado).

Los registros que se crean en forma dinámica, no pueden ser referenciados directamente por un identificador, en lugar de eso, se accesan por medio de un "pointer", es decir, por medio de un apuntador.

Cuando un "pointer" no apunta a ningún elemento, su valor es NIL. Los apuntadores ("pointers") son una herramienta usada para construir estructuras de datos complicadas y flexibles (usando registros) como son las listas ligadas, árboles, colas, stack's, etcétera.

Ejemplo :

```
TYPE  LIGA   = @ OBJETO;
      OBJETO = RECORD
                 SIGUIENTE : LIGA ;
                 DATO      : CHAR
                 END;
```

Con las instrucciones anteriores se está declarando un "pointer" LIGA a un registro llamado OBJETO. EL registro OBJETO podrá utilizarse como nodo, en la formación de estructuras ligadas.

7.2 Operaciones

Las operaciones aritméticas básicas, en PASCAL, están representadas como sigue:

SQR	exponenciación (cuadrado)
*	multiplicación
/	división real
DIV	división entera
+	suma
-	resta

Los operadores de relación, son los siguiente:

<	(menor que)
<=	(menor o igual a)
=	(igual a)
<>	(distinto de)
>=	(mayor o igual a)
>	(mayor que)

Los operadores Booleanos son:

AND

OR

NOT

También se incluye un conjunto de funciones estándar, tales como, SQRT, SIN, COS, TRUNC, etcétera.

La operación de asignación tiene la siguiente sintaxis:

<variable> := <expresión> |<constante>

Las operaciones que se pueden realizar con los SET's son aquellas definidas para los conjuntos, en álgebra.Los siguientes operadores son aplicables en las operaciones con conjuntos:

+	Unión
*	Intersección
-	Diferencia

Los operadores de relación aplicables a los conjuntos son:

=	(igualdad de conjuntos)
<>	(desigualdad de conjuntos)
<=	(contenido en)
>=	(contiene a)

Para probar la existencia de un elemento dentro de un conjunto se utiliza la palabra IN.

Las operaciones de entrada-salida, en PASCAL, se realizan a través de las funciones READ, READLN, GET, WRITE, WRITELN y PUT cuyos parámetros son el nombre del archivo -el cual puede omitirse si se va a trabajar en los archivos por omisión (INPUT, OUTPUT)- y la lista de variables que serán leídas o escritas. Estas operaciones, generalmente, se utilizan junto con las funciones EOF (fin de archivo) y EOL (fin de línea).

Las operaciones definidas por el programador están representadas, en PASCAL, por los subprogramas, en el sentido usual (PROCEDURE's) y las funciones (FUNCTION's)

7.3 Control de secuencia

PASCAL proporciona varias estructuras, para controlar la secuencia de ejecución de las proposiciones dentro de un programa, a continuación se verán estas estructuras. Una proposición en PASCAL puede ser simple o compuesta

Una proposición compuesta, es un BEGIN, seguido de una serie de proposiciones simples y finalmente un END. Cada proposición simple debe ser separada de -- otra por medio de un punto y coma (;).

Las expresiones pueden ser aritméticas o Booleana. Una expresión aritmética está compuesta de variables simples o con índice, constantes enteras o reales, llamadas a funciones y las operaciones aritméticas primitivas. Se asume la jerarquía usual de las funciones, asociando de izquierda a derecha y el uso de paréntesis para control explícito, cuando es necesario. Las expresiones Booleanas son similares a las expresiones aritméticas, excepto que éstas involucran variables y constantes Booleanas, así como los operadores de relación y Booleanos.

Los mecanismos usados para controlar la secuencia son los GO TO's las proposiciones condicionales y las de iteración.

La proposición GO TO <etiqueta> permite transferir el control a la proposición etiquetada con <etiqueta> . La etiqueta usada debe ser un entero positivo sin signo y debe declararse antes de ser utilizada. Una proposición GO TO puede utilizarse para saltar dentro de un mismo nivel o para saltar de un nivel interno a uno externo, pero no al contrario. En particular, se puede usar un GO TO para salir de un subprograma, pero no para entrar a la mitad del mismo.

Las proposiciones condicionales tienen la siguiente forma:

IF <expresión Booleana> THEN <proposición 1> ELSE <Proposición 2>
Si la <expresión Booleana> es verdadera (toma el valor TRUE) realiza la <proposición 1> e ignora la <proposición 2> , en caso contrario, ejecuta la <proposición 2> ignorando la <proposición 1> .

La proposición CASE es una generalización de la proposición IF, ésta permite que el proceso ejecute una de varias acciones dependiendo del valor de la <expresión> . Su sintaxis es la siguiente:

CASE <expresión> OF <constante> : <Proposición> END

Por ejemplo:

```
CASE control OF
2,5    : <proposición 1>  ;
3,7,11 : <proposición 2>  ;
8      : <proposición 3>
END
```

Es conveniente, hacer notar que la <expresión> puede ser un escalar o un número entero.

A continuación se hablará de las proposiciones de iteración que proporciona PASCAL.

a) REPEAT <proposición> UNTIL <condición>

Ejecuta la <proposición> por lo menos una vez hasta que la <condición> se satisface.

b) WHILE ‹condición› DO ‹proposición›

Esta es similar a la anterior, excepto que la condición se evalua al principio del "loop" en lugar de al final

c) FOR ‹variable› := ‹valor inicial› TO ‹valor final› DO ‹proposición› , o bien:

FOR ‹variable› := ‹valor inical› DOWNTO ‹valor final› DO ‹proposición›

Realiza la proposición hasta que la ‹variable› toma el ‹valor final›, empezando con el ‹valor inicial› especificado e incrementándolo o decrementándolo en 1 dependiendo se se usa TO o DOWNTO.

PASCAL permite llamadas recursivas a subprogramas. Para el retorno de un subprograma no existe una instrucción explícita, sino que se realiza hasta que se ejecuta la última instrucción del subprograma.

Un subprograma puede ser llamado como una función dentro de una expresión o como una subrutina, simplemente escribiendo el nombre del subprograma y la lista de los parámetros actuales. Por ejemplo, BUSCAP (A, B, 4).

7.4 Control de datos

PASCAL proporciona dos técnicas para la transmisión de parámetros por valor o por referencia. El tipo de técnica usada se especifica en la declaración de los parámetros formales, en el encabezado del subprograma. Por

ejemplo, la proposición:

PROCEDURE EJEM (VAR B: INTEGER; J: BOOLEAN);

está indicando que se tiene un procedimiento llamado EJEM, el cual a su vez tiene dos parámetros B y J; B es pasado por referencia ya que, va precedido de la palabra VAR y J es transmitido por valor. Cuando el parámetro se va a transmitir por valor el parámetro actual, puede ser una expresión, pero cuando se va a transmitir por referencia, el parámetro actual debe ser una variable, nunca una expresión o constante.

Los parámetros pueden ser enteros, reales, Booleanos, caracteres, funciones, procedimientos o escalares.

7.5 Manejo de almacenamiento

El manejo de almacenamiento en PASCAL, se basa generalmente, en una área estática de almacenamiento, un stack, y un "heap" asignado dinámicamente.

Los programas son traducidos en bloques de código ejecutable y almacenados estáticamente, antes de empezar su ejecución.

El área de almacenamiento reservada por el stack, al momento de ejecución, se utiliza para activar los registros asociados a bloques (en PASCAL son siempre el cuerpo de porcedimientos, funciones o del programa principal). El uso de un stack garantiza un almacenamiento óptimo, ya que cada dato exis-

te sólo durante la ejecución del bloque al que pertenece, es decir, se crean al entrar al bloque y se destruyen al salir del mismo.

El "heap" se utiliza al momento de ejecución para almacenar los elementos creados con el procedimiento estándar NEW.

8 SNOBOL4

El lenguaje SNOBOL fue diseñado, especialmente, para manipular fácilmente cuerdas de caracteres. Fue creado en los laboratorios "Bell Telephone" a principios de 1960. La poderosa versión conocida como SNOBOL4 es la culminación (en 1962) de los desarrollos de SNOBOL.

Actualmente, SNOBOL4 es un lenguaje ampliamente utilizado para resolver problemas, en donde se tienen que procesar en diversas formas gran cantidad de cuerdas de caracteres; por ejemplo: problemas de simulación, traducción de lenguajes naturales, lingüística y análisis musical.

SNOBOL4 tiene muchas características, que no son comúnmente encontradas en otros lenguajes de programación, entre ellas se cuentan:

a) Un conjunto poderoso de primitivas para manipular cuerdas, empleando el concepto de patrón de igualdad.
b) Una sintaxis única y muy compacta
c) Permite al programador extender el lenguaje, definiendo tipos de datos que le sean de utilidad.
d) Conversión automática del tipo de datos
e) Una tabla central de cuerdas, es mantenida, la cual contiene una entrada única para cada cuerda que aparezca en el programa, ya sea como identificador o como valor de alguna variable.

8.1 Datos

En SNOBOL4 las variable utilizadas dentro de un programa no se declaran, es decir, no tienen un tipo fijo, el tipo asociado con cada variable depende del último valor asignado a ella por lo tanto, cada variable incluye un descriptor al momento de ejecución.

SNOBOL4 proporciona como datos elementales números enteros, números reales y cuerdas de caracteres, estas últimas constituyen el principal tipo de datos. Cada cuerda se representa como un bloque secuencial que contiene la serie apropiada de caracteres, junto con un descriptor que contiene la longitud de la cuerda.

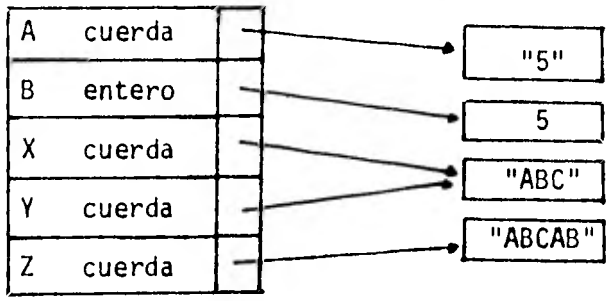
La característica principal de SNOBOL4 es el permitir que cada cuerda utilizada en un programa, sea almacenada una sola vez; cada variable con valor igual a una cuerda dada tiene un apuntador a la celda de la tabla central de cuerdas que contiene dicha cuerda. Estas ideas se ilustran a continuación:

Considerese el siguiente programa en SNOBOL4:

```
1        A   =  "5"
2        B   =   5
3        X   =  "AB" "C"
4        Y   =  "ABC"
5        Z   =  X "AB"
6        X   =  B + 1
7        B   =  Y A
```

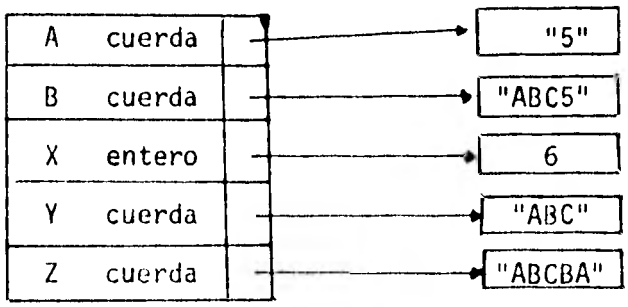
Cuando se han ejecutado las primeras cinco líneas del programa, se tiene la siguiente situación:

Tabla Central de Cuerdas:



En la figura, se puede observar que tanto X como Y apuntan, en la tabla central de cuerdas, a la celda que contiene la cuerda "ABC"; aunque los valores de X y Y resultaron de distintas expresiones (líneas 3 y 4 del programa). Sin embargo, la unicidad de almacenamiento de cuerdas no se extiende a nivel de subcuerdas.Así la variable Z tiene un apuntador a la celda que contiene "ABCAB", aunque los tres primeros caracteres de la cuerda aparezcan en otra celda.

Una vez ejecutadas las siete líneas del programa el almacenamiento es como sigue:



Es decir, la variable B que antes era un entero, ahora es una cuerda, y X que era una cuerda, ahora es un entero.

Junto con las cuerdas, los patrones son un tipo de datos de gran importancia en SNOBOL4, porque las operaciones de este lenguaje se centran en el uso de "patrones de igualdad". Durante la ejecución, un patrón se representa internamente como áreas de almacenamiento ligadas, formando una estructura de árbol, el cual refleja la estructura natural de un patrón como un conjunto de alternativas y sub-patrones concatenados.

Un arreglo en SNOBOL4 es un dato, que tiene un conjunto de apuntadores a otros datos, de esta forma cada elemento del arreglo puede tener cualquier tipo inclusive ser otro arreglo; lo único que permanece fijo, es la longitud del mismo.

Los arreglos son creados con la función ARRAY, la cual tiene el siguiente formato:

ARRAY (< límite > $\left\{, \text{<límite>}\ {}^{n}_{0}\right\}$, <valor inicial>).

Por ejemplo:

ARRAY ('3,4','X')

define un arreglo bi-dimensional de 3 x4 y al momento de crearlo sus elementos tendrán como valor inicial X.

La proposición:

VECTOR = ARRAY (10)

crea un arreglo de 10 elementos, el apuntador a este arreglo es asignado como valor a la variable VECTOR, almacenada en la tabla central de cuerdas. Al escribir VECTOR <3> se está haciendo referencia al tercer elemento del arreglo.

La asignación:

Y = VECTOR

causa que el mismo apuntador sea asignado a la variable Y, así el arreglo tiene ahora dos nombres y tanto VECTOR < 3 > como Y < 3 > se refieren al mismo elemento del arreglo.

Una tabla en SNOBOL4 es un arreglo lineal heterogéneo de tamaño variable, en el cual los subíndices para accesar sus elementos son cuerdas de caracteres. Cada entrada de la tabla consta de una pareja formada por un subíndice, que es un apuntador a una cuerda en la tabla central de cuerdas y por un valor que es un número, o bien un apuntador a otro dato. Las entradas a una tabla no pueden borrarse.

Las tablas se crean por medio de función TABLE y en sus argumentos se especifican tanto el tamaño inicial de la tabla, como el tamaño de incremento. Por ejemplo, la proposición:

T = TABLE (10,5)

causa la creación de un bloque de almacenamiento para una tabla de 10 parejas subíndice-valor. La función TABLE, regresa un apuntador a ese bloque y se asigna a la variable T. Si se requieren entradas adicionales, éstas son asignadas en bloques de 5, así la asignación de un valor al 11-avo elemento de T crea un nuevo bloque de almacenamiento para 5 parejas y éste se liga al bloque original por medio de un apuntador; la asignación al 16-avo elemento crea otro bloque igual al anterior, de tal forma que una tabla, desde el punto de vista de almacenamiento, es una lista ligada de bloques de igual longitud, excepto el primero.

La tabla central de cuerdas que contiene todos los identificadores es aumentada por dos tablas auxiliares. La tabla de etiquetas, que contiene parejas de apuntadores, uno a la cuerda utilizada como etiqueta y otro a la posición dentro del código ejecutable en donde aparece la etiqueta. La tabla de subprogramas consta de parejas similares de apuntadores, uno a la cuerda utilizada como nombre del subprograma y otro a la posición donde se definió el subprograma.

La presencia de estas tablas al momento de ejecución permite que cualquier cuerda sea leída o calculada y usada en un GO TO o como nombre de subprogramas en la llamada de los mismos.

SNOBOL4 permite que el programador defina sus propias estructruas de datos. Una definición de este tipo básicamente especifica una clase de arreglos lineales heterogéneos de longitud fija, cuyos elementos se pueden referenciar por subíndices que son cuerdas de caracteres. La definición de estos datos se realiza con la función DATA. Por ejemplo, la proposición:

```
DATA("EMPLEADO (NOMBRE, EDAD, DIRECCION)")
```

define un tipo de datos llamado EMPLEADO. Cada dato de esta estructura consta de un arreglo lineal de tres elementos referenciables por NOMBRE, EDAD y DIRECCION.

Con la proposición:

```
X = EMPLEADO ("MYRTHA",28,"AV. TOLUCA # 110")
```

se crea un arreglo de tres elementos del tipo EMPLEADO, con valores iniciales

"MYRTHA", "28" y "AV. TOLUCA # 110". El valor regresado por la función EMPLEADO es un apuntador al nuevo arreglo y se asigna como valor de X.

El programador puede construir estructuras ligadas arbitrariamente, permitiendo que el valor de un elemento en una estructura sea un apuntador a otra estructura. De esta forma pueden crearse y procesarse listas ligadas, árboles, etcétera.

Algunos parámetros y switch's internos de SNOBOL4 pueden utilizarse por medio de palabras clave, éstas deben ir precedidas por un & . Por ejemplo, el programador puede determinar, en algún punto, cuantas proposiciones han sido ejecutadas, en este caso haría N = &STCOUNT.

Otro ejemplo sería, si se sabe que cuando el valor de &DUMP es un entero distinto de cero al finalizar un programa, se imprime un vaciado (dump) de las variables utilizadas en el mismo, y se desea obtener dicho listado, se hará: &DUMP = 1

Las palabras clave pueden ser protegidas, en este caso sus valores pueden ser accesados, pero no modificados como &STCOUNT, o bien desprotegidas, en este caso sus valores pueden ser accesador y modificados como &DUMP.

El código objeto es un tipo de datos, en SNOBOL4, al igual que las cuerdas, patrones y arreglos. Durante la ejecución de un programa es posible, usando la función CODE, convertir una cuerda de caracteres en código objeto. El

argumento de CODE es una cuerda que representa una o varias proposiciones en SNOBOL4, separadas por punto y coma. El valor de una llamada a CODE es el código objeto ejecutable, de la cuerda que se pasó como parámetro.

8.2 Operaciones

Aunque SNOBOL4 es un lenguaje principalmente utilizado para manipular cuerdas de caracteres, también cuenta con operaciones aritméticas para números enteros y reales; las cuales utilizan los siguientes operadores:

**	(exponenciación)
/	(división)
*	(multiplicación)
+	(suma)
-	(resta)

Los operadores de relación estan representados por las siguientes funciones:

LT

LE

EQ

NE

GE

GT

Las cuerdas de caracteres son comparadas utilizando la función

IDENT.

En SNOBOL4 no existen los valores Booleanos TRUE y FALSE, en lugar de eso una operación puede fallar o ser un éxito, y su resultado puede ser examinado en el campo de GO TO de la proposición.

El operador de asignación es = , sin embargo; la operación de asignación es novedosa en el sentido que el lado derecho de la misma puede crear cualquier tipo de dato. En general, el lado derecho de una expresión evalúa un apuntador y la asignación implica hacer una copia de ese apuntador el cual es asignado como nuevo valor de la variable designada en el lado izquierdo

La concatenación, es la operación básica para combinar dos cuerdas de caracteres y formar una tercera. No existe un operador explícito para la concatenación, basta con un espacio en blanco entre las cuerdas que se van a concatenar. Por ejemplo,

A = "VILLA" "NUEVA"

con la instrucción anterior, la variable A tendrá el valor de "VILLANUEVA".

La operación de examinar cuerdas para encontrar una subcuerda específica, es fundamental en SNOBOL4. Las operaciones que involucran patrón de igualdad tienen dos formas:

1. Proposiciones de pruebas de patrón cuya sintaxis es:

   <cuerda a examinar> <patrón>

2. Proposiciones de reemplazamiento, con la siguiente sintaxis:

‹cuerda a examinar› ‹patrón› = ‹cuerda reemplazante›

Por ejemplo, si

CUERDA = "PROGRAMADOR"

PATRON = "GRAM"

la proposición:

CUERDA PATRON

busca la subcuerda "GRAM" dentro de la cuerda "PROGRAMADOR"

Al efectuar la siguiente proposición, reemplazará la subcuerda "DOR" por "DORA", en la cuerda "PROGRAMADOR":

CUERDA "DOR" = "DORA"

Un patrón es una estructura de datos, en la cual sus valores no pueden ser modificados, su uso está restringido a servir de argumento en una operación de patrón de igualdad.

Un patrón puede especificarse, simplemente, como una cuerda en una operación de patrón de igualdad, como en la última proposición del ejemplo anterior, también es posible especificar patrones más complejos, para esto existen dos operaciones válidas con patrones, ellas son: alternativas y concatenación.

Las alternativas se especifican separando cada patrón por un | y un espacio en blanco, por ejemplo:

LLAVE = "COMPUTADORA" | "PROGRAMA"

con la instrucción anterior, se está definiendo un patrón llamado LLAVE, con el cual se compararán las cuerdas "COMPUTADORA", o bien "PROGRAMA". Además se pueden agregar elementos al patrón de la siguiente forma:

```
LLAVE  =  LLAVE | "ALGORITMO"
```

La concatenación de dos patrones, se especifica de la misma forma que la concatenación de dos cuerdas, es decir, separando los patrones con un espacio en blanco. El valor de la expresión es un patrón que comparará cualquier subcuerda que empiece con algún elemento del primer patrón seguida de algún elemento del segundo patrón.

Es posible asociar a una variable un componente de un patrón, si el patrón de igualdad se realiza con éxito, la subcuerda encontrada se asigna a la variable. El operador . es el operador de asignación del valor condicional. Por ejemplo:

```
BASE  =  ("HEX" | "DEC").B2
```

asigna a BASE un patrón cuyos elementos son HEX ó DEC. Si BASE es utilizado exitosamente, en un patrón de igualdad, el valor de B2 será la subcuerda encontrada en el patrón de igualdad.

Generalmente, los valores de un patrón permanecen fijos al utilizarlos en un patrón de igualdad, pero en ocasiones se desea dejar ciertas variables o expresiones como parámetros en un patrón y evaluarlos sólo al momento de utilizarlos en un patrón de igualdad. El operador * indica que una variable o sub-expresión, permanecerá sin evaluar hasta el momento de utilizarlo

en un patrón de igualdad, por ejemplo:

PAT = LEN (*N)

crea un patrón con parámetro N, cuyo valor puede variar cada vez que PAT se utilice, es decir, buscará cualquier cuerda de longitud N.

Las expresiones sin evaluar en los patrones, también permiten patrones recursivos, por ejemplo, el patrón

P = "A" | *P "A"

buscará cualquier serie de A's.

El sistema de entrada-salida en SNOBOL4 se basa en la transmisión directa de cuerdas de caracteres entre la memoria central y archivos secuenciales externos. Usando las operaciones de patrón de igualdad y otras primitivas para manipular cuerdas, las cuerdas de entrada pueden dividirse en componentes y la salida formatearse fácilmente. Como resultado de esto las operaciones de entrada y/o salida sólo necesitan transmitir cuerdas de caracteres usando las variable INPUT y OUTPUT.

Los subprogramas en SNOBOL4 tienen algunos aspectos novedosos. Durante la traducción el programa principal y el conjunto de subprogramas forman una gran lista de proposiciones sin distinción sintáctica, entre los elementos de las diferentes rutinas. Antes de que un subprograma pueda ser llamado debe primero ser definido por medio de la siguiente función

DEFINE (" <identificador> (<parámetros>) <variables locales>",
" <etiqueta> ")

Al momento de ejecución la función DEFINE introduce en la tabla de subprogramas definidos, el nombre del subprograma, el número y nombre de sus variables locales y la etiqueta de la primera proposición en el cuerpo del subprograma. Si la etiqueta se omite, entonces el nombre del subprograma es también la etiqueta de la primera proposición.

Las proposiciones que forman el cuerpo de un subprograma pueden ir en cualquier lugar del programa y no necesitan existir cuando el subprograma es definido, ya que pueden leerse o construirse como una cuerda y después traducirse con la función CODE, antes de que el subprograma sea llamado.

8.3 Control de secuencia

Las expresiones en SNOBOL4 producen valores numéricos, cuerdas o patrones. El orden de precedencia es el usual para operaciones aritméticas, extendiendolo a las operaciones de concatenación, aternación y asignación condicional. Por ejemplo, la concatenación tiene menor prioridad que las operaciones aritméticas, así:

```
RENGLON  =  "K"
NUM      =  22
LOCALIZA =  RENGLON  NUM  + 4/2
```

la última proposición es equivalente a LOCALIZA = RENGLON (NUM + 4/2), por lo tanto el valor de LOCALIZA es K24.

Una etiqueta es una letra o dígito seguida de cualquier número de

caracteres hasta encontrar un espacio en blanco.

La transferencia a una proposición etiquetada se especifica en el campo de GO TO que aparece al final de una proposición y se separa del resto de la proposición por dos puntos (:). Dos tipos de transferencia pueden especificarse: condicional e incondicional.

En la transferencia condicional se especifica la etiqueta escribiendola entre paréntesis y precedida por una F o una S, correspondiente a fracaso o éxito de la proposición, por ejemplo, con la proposición:

TEXTO = INPUT : F (FIN)

se lee un registro y se asigna a la variable TEXTO, si no existe dato, entonces la proposición fracasa y el control se transfiere a la etiqueta FIN.

Cuando un subprograma es llamado, el control es transferido a la primera proposición del mismo, de ahí el control puede pasar a cualquier otra proposición, ya sea del programa principal o de otros subprogramas. El regreso de un subprograma tiene lugar cuando se especifica un GO TO a las etiquetas RETURN, NRETURN ó FRETURN, las dos primeras se utilizan para un regreso normal y la última cuando la proposición falla.

8.4 Control de datos

El medio ambiente de referencia de un programa en SNOBOL4 se restringe a una tabla central de cuerdas, utilizada para almacenarlas. Cada cuerda

creada durante la ejecución de un programa puede ser referenciada como variable utilizando el operador de referencia indirecta, $, este operador aplicado a cualquier cuerda recupera el valor asociado con la misma. Por ejemplo:

```
MES  =  "FEBRERO"
$MES =  "LOCO"
```

esta última proposición es equivalente a FEBRERO = "LOCO".

El hecho de que cada cuerda pueda ser utilizada automáticamente como variable, y que ésta forme parte del medio ambiente de referencia actual permite que la tabla central de cuerdas sirva también como tabla de medio ambiente de referencia. Cada entrada a la tabla, es indicada por un apuntador al dato que sirve como dato de la cuerda asociada.

Todas las cuerdas son introducidas en la tabla central de cuerdas como variables globales, de esta manera, su existencia no se ve afectada por entradas o salidas a subprogramas. Las variables locales son designadas cuando se define un subprograma.

El efecto de utilizar la tabla central de cuerdas como medio ambiente de referencia, es básicamente hacer referencias no-locales por medio de la regla de asociación más reciente. En la entrada a un subprograma las asociaciones para todos los identificadores locales son desactivadas y en la salida son restauradas, pero todos los otros identificadores retienen sus asociaciones existentes (y además las más recientes)

La transmisión de parámetros a subprogramas es únicamente por valor, Ni los nombres de subprogramas, ni las etiquetas pueden transmitirse directamente como parámetros; en lugar de eso se transmite una cuerda que representa el nombre del subprograma o de la etiqueta. Dentro del subprograma la cuerda puede utilizarse como objeto de un GO TO o como llamada a otro subprograma, referenciando el parámetro formal asociado (utilizando $, o bien APPLY). Con está técnica se ignoran los problemas de transmisión de medio ambiente de referencia que , generalmente, se asocia cuando se pasan como parámetros el nombre de un subprograma o una etiqueta.

El último valor asignado al nombre del subprograma, dentro del mismo, es regresado como valor de la llamada al subprograma. Si el resultado se va a utilizar inmediatamente como parte de una asignación, entonces el regreso debe ser a través de un GO TO a la etiqueta NRETURN, de otra forma se utiliza la etiqueta RETURN. En ambos casos la salida es exitosa, en caso de no ser exitosa se utiliza la etiqueta FRETURN y no se obtiene ningún resultado.

8.5 Manejo de almacenamiento

Para manejar el almacenamiento de un programa en SNOBOL4 es necesaria una organización tripartita de la memoria, un área estática para almacenar las rutinas del sistema y varias tablas de longitud fija; un stack para almacenar registros de activación de subprogramas y un "heap" para almacenamiento de programas traducidos, estructuras de datos del usuario, tablas de longitud variable y otros datos del sistema. El "heap" es asignado en bloques de

longitud variable y la recolección de basura y compactación tiene lugar cuando ya no hay espacio libre.

El registro de activación en el stack, contiene los valores desactivados para los parámetros formales, variables locales e identificadores para nombres de subprogramas, como se ve aquí, el stack no juega un papel directo en el referenciamiento.

El "heap" es el área de almacenamiento central en SNOBOL4, ya que permite al programador crear cuerdas, arreglos, tablas y estructuras de datos definidos por él, en puntos arbitrarios durante la ejecución del programa y como resultado de esto todos los datos definidos por el programador son asignados en un espacio del "heap". Además, el traductor puede llamarse en cualquier punto durante la ejecución, para traducir una cuerda de caracteres que represente un conjunto de proposiciones en SNOBOL4.

Conclusiones

Los lenguajes de programación son el medio a través del cual una persona puede comunicarse con la computadora, de ahí que sea importante conocer las características de los mismos para que esa comunicación se establezca de la mejor forma posible.

Al analizar detalladamente las características que presentan los lenguajes particulares se profundiza en el conocimiento de los mismos logrando, de esta manera, emplearlos de la mejor forma posible en la solución de un problema dado, o bien tener las bases con el fin de poder formarse criterios para la elección del lenguaje más conveniente para una aplicación particular.

Tomando en cuenta que los lenguajes de programación sirven para lograr la comunicación entre el hombre y la computadora, todos los desarrollos en esa área están encaminados a lograr que dicha comunicación sea más fácil para el hombre. Por esta razón, continuamente se están desarrollando lenguajes para aplicaciones especializadas, debido a que tienen grandes ventajas para los usuarios porque tratan únicamente los temas que le conciernen directamente para una aplicación específica.

Entre otra tendencias importantes de los lenguajes desarrollados últimamente, se encuentran las siguientes: permiten que el usuario defina sus propias operaciones y/o tipos de datos (FORTH, ILIAD), son interactivos

(FORTH, MATHSY), proporcionan mecanismos para manejo de errores al momento de ejecución (ILIAD), permiten programas concurrentes y fácilmente transportables a distintas máquinas (ADA, FORTH, ILIAD).

Sin embargo, lo ideal sería que el lenguaje con el que se llevara a cabo esa comunicación fuera el lenguaje natural de la persona (Inglés, Español, etcétera), es por eso que existe la tendencia muy marcada a utilizar lo que se llama "software transparente", en donde personas con escasos o nulos conocimientos de computación pueden accesar la computadora para sus aplicaciones particulares y en su lenguaje natural, pero esto es debido a que la computadora tiene implementados paquetes, que son conjuntos de programas interactivos, elaborados por personas con amplios conocimientos en el área.

Apéndice A

Sintaxis y Ejemplos

A.1 FORTRAN

El alfabeto de FORTRAN está constituido por los siguientes símbolos:

a) Caracteres alfabéticos:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

b) Caracteres numéricos:

0 1 2 3 4 5 6 7 8 9

c) Caracteres especiales:

* / + - . , ( ) $ = ' & " espacio en blanco

Para construir proposiciones válidas en FORTRAN se utilizan tanto las palabras reservadas del lenguaje, como los identificadores definidos por el programador. Un identificador en FORTRAN consta de uno a seis caracteres - alfanuméricos empezando con una letra.

Normalmente, sólo se codifica una proposición FORTRAN en cada línea, sin embargo si la proposición es demasiado larga pueden utilizarse otras líneas, escribiendo un caracter numérico, distinto de cero, en la columna -- seis, para indicar que son continuación de la línea anterior. Las cinco primeras columnas de la tarjeta, están reservadas para las etiquetas. Una etiqueta es un entero sin signo que consta de uno a cinco dígitos. De la columna 7 a la 72 se codifican las proposiciones FORTRAN. Además, para escribir cualquier comentario del programa, se debe escribir una letra C en la columna uno.

Un programa, en FORTRAN, consta de un programa principal, seguido de un conjunto de subprogramas.Cada subprograma tiene la siguiente estructura: un encabezado (excepto el programa principal), seguido de las declaraciones, proposiciones ejecutables del programa y finalmente la palabra END.

Las líneas individuales de un programa, en FORTRAN, generalmente son fáciles de entender; sin embargo, la estructura de un programa, completo, no es clara por el amplio uso de etiquetas y GO TO's como mecanismos para controlar la secuencia, haciendo díficil seguir el flujo de un programa en -- FORTRAN. Otra dificultad, en cuanto a entender un programa es que no pueden utilizarse nemónicos como identificadores, ya que éstos deben tener a lo más seis caracteres.

A continuación se muestra un programa en FORTRAN que efectúa la suma, resta y/o multiplicación de dos matrices.

```
C
C   ESTE PROGRAMA CALCULA LA SUMA,RESTA Y/O MULTIPLICACION DE DOS
C   MATRICES DE CUALQUIER DIMENSION (MENOR QUE 10,10).
C
C   SI IOPC = 1 CALCULA LA SUMA
C   SI IOPC = 2 CALCULA LA RESTA
C   SI IOPC = 3 CALCULA LA MULTIPLICACION
C   SI IOPC = 4 CALCULA LAS TRES OPERACIONES
C
C   N Y M SON LAS DIMENSIONES DE LA MATRIZ "A".
C   N1 Y M1 SON LAS DIMENSIONES DE LA MATRIZ "B".
C   EL RESULTADO SE ALMACENA EN LA MATRIZ "C".
C
C
C   EN ESTE PROGRAMA SE UTILIZA LA PROPOSICION "COMMON" PARA HACER
C   GLOBALES LAS MATRICES Y SUS DIMENSIONES Y DE ESTA FORMA EVITAR
C   USAR PARAMETROS EN LAS SUBRUTINAS.
C
C
      COMMON A(10,10),B(10,10),C(10,10),N,M,N1,M1,IOPC
C   LEE LA OPCION DESEADA
      READ (5,*) IOPC
C   LEE LAS DIMENSIONES
      READ (5,*)N,M
      READ  (5,*) N1,M1
C   VALIDA LAS DIMENSIONES
      IF  (N.GT.10.OR.M.GT.10.OR.N1.GT.10.OR.M1.GT.10) GO TO 999
      IF  (IOPC.EQ.3.AND.M.NE.N1) GO TO 999
      IF  (IOPC.NE.3.AND.N.NE.N1.OR.M.NE.M1) GO TO 999
```

```
      C*    LEE LA MATRIZ A
            DO 110 I=1,N
            READ (5,*)  (A (I,J),J=1,M)
  110       CONTINUE
      C*    LEE LA MATRIZ B
            DO 120 I=1,N1
            READ (5,*)  (B (I,J),J=1,N1)
  120       CONTINUE
            DO 10  (100,200,300,400) IOPC
  110       CALL SUMA
            STOP
  200       CALL RESTA
            STOP
  300       CALL MULTI
            STOP
  400       CALL SUMA
            CALL RESTA
            CALL MULTI
            STOP
  999       WRITE (6,998)
  998       FORMAT (" ERROR EN LAS DIMENSIONES DE LAS MATRICES")
            STOP
            END
            SUBROUTINE SUMA
            COMMON A(10,10),B(10,10),C(10,10),N,M,N1,M1,IOPC
            DO 10 I=1,N
            DO 10 J=1,M
            C (I,J) = A (I,J) + B (I,J)
   10       CONTINUE
             IOPC = 1
            CALL ESCRI
```

```
      RETURN
      END
      SUBROUTINE RESTA
      COMMON A(10,10),B(10,10),C(10,10),N,P,Q,N1,M1,IOPC
      DO 10 I=1,N
      DO 10 J=1,M
      C (I,J) = A (I,J) - B (I,J)
   10 CONTINUE
      IOPC = 2
      CALL ESCRI
      RETURN
      END
      SUBROUTINE MULTI
      COMMON A(10,10),B(10,10),C(10,10),N,P,Q,N1,M1,IOPC
      DO 10 I=1,N
      DO 10 J =1,M1
      DO 10 K =1,N1
      C (I,J) = C (I,J) + A (I,K)* B (K,J)
   11 CONTINUE
      IOPC = 3
      CALL ESCRI
      RETURN
      END
      SUBROUTINE ESCRI
      COMMON A(10,10),B(10,10),C(10,10),N,P,Q,N1,M1,IOPC
      WRITE (6,100)
  100 FORMAT ('1',' ESTA FORMULA ES PARA CENTRAR LA MATRIZ')
      MARG = (132 -5 * M)/2
      DO 10 I=1,N
      WRITE (6,200) MARG,' ',(A (I,J),J=1,M)
   10 CONTINUE
```

```
      MAR = (132 - 5 * M1)/2

      WRITE (6,200)
      DO 20 I =1,N1

      WRITE (6,500)  MARG,M1,  (B (I,J),J=1,M1)
   20 CONTINUE

      IF  (ID.EQ.1) WRITE (6,300)
      IF  (ID.EQ.2) WRITE (6,400)

      IF  (ID.EQ.3) WRITE (6,500)
      MAR = (132 - 5 * N)/2
      DO 30 I=1,N

      WRITE (6,500) MARG,M1,    (C (I,J),J=1,M1)

   30 CONTINUE

      RETURN
  100 FORMAT (1H1,///,53X," A MATRIZ A E'")
  200 FORMAT (    ///,53X," A MATRIZ B E'")
  300 FORMAT (    ///,53X," A MATRIZ A + B E' ")
  400 FORMAT (    ///,53X," A MATRIZ A - B E' ")
  500 FORMAT (    ///,53X," A MATRIZ A * B E' ")
  600 FORMAT ('K=',I5)

      END
```

LA MATRIZ A ES:

1	2	3
4	45	6
3	7	8

LA MATRIZ B ES:

7	9	0
1	1	0
3	4	6

LA MATRIZ A + B ES

8	13	3
5	45	6
6	11	14

LA MATRIZ A ES:

1	2	3
4	45	6
3	7	8

LA MATRIZ B ES:

7	9	C
1	1	D
3	4	6

LA MATRIZ A - B ES

-6	-5	3
3	44	6
C	3	2

LA MATRIZ A ES:

1	2	3
4	45	6
3	7	8

LA MATRIZ B ES:

7	5	0
1	1	0
3	4	6

LA MATRIZ A * B ES

12	15	21
94	145	42
52	65	50

A.2 COBOL

El alfabeto de COBOL está constituido por los siguientes símbolos:

a) Caracteres alfabéticos:

A B C D E F G H I J K L M N O P Q R S T U V W X Y S

b) Caracteres numéricos:

0 1 2 3 4 5 6 7 8 9

c) Caracteres especiales:

* / + - = . , : ; " ' ( ) $ < > [ ]

espacios en blanco.

Una palabra o identificador, en COBOL, se construye empezando con un caracter alfabético seguido de cero a 29 caracteres alfanuméricos pudiendo emplearse los guiones ("-").

Como se mencionó en el capítulo correspondiente, un programa en COBOL, está constituido por cuatro divisiones (IDENTIFICATION, ENVIRONMENT DATA y PROCEDURE), cada una compuesta de secciones, que a su vez se dividen en parráfos, excepto las dos primeras.

El formato de codificación en COBOL es como sigue: si en la columna 7 aparece un guión significa que esta línea es continuación de la anterior Los nombres de las divisiones, secciones y parráfos, así como los niveles 01

y 77 deben empezar en la columna 8, y todo lo demás a partir de la columna 12. Además todas las proposiciones deben terminar con un punto.

Ya que es relativamente simple y natural la sintaxis, los programas en COBOL no son díficiles de escribir, pero si es tediosa la programación por la gran cantidad de código que produce y porque se requieren declaraciones detalladas para cada dato que se utilice.

A continuación se muestra un programa en COBOL, en él que se calcula el promedio de las calificaciones de un alumno y se ordenan las materias en orden alfabético, haciendo uso del SORT de COBOL.

```
100           IDENTIFICATION DIVISION.
200           PROGRAM-ID.CARDEX DE ALUMNOS.
300          *DADOS LOS DATOS DE UN ALUMNO, ORDENARLOS POR NOMBRE DE MATERIA Y
400          *SACAR EL PROMEDIO GENERAL
500           AUTHOR. AMPARO LOPEZ GAONA.
600           DATE-COMPILED.
700           ENVIRONMENT DIVISION.
800           CONFIGURATION SECTION.
900           SOURCE-COMPUTER.B6700.
1000          OBJECT-COMPUTER.B6700.
1100          I-O SECTION.
1200          FILE-CONTROL.
1300         *ASIGNACION DE ARCHIVOS
1400              SELECT ENTRADA ASSIGN TO DISK.
1500              SELECT SALIDA ASSIGN TO PRINTER.
1600         *ARCHIVO PARA SORTEAR
1700              SELECT SORTED ASSIGN TO SORT DISK.
1800          DATA DIVISION.
1900          FILE SECTION.
2000         *DESCRIPCION DE ARCHIVOS
2100          FD ENTRADA
2200              RECORD CONTAINS 80 CHARACTERS
2300              DATA RECORD IS TARJETA.
2400          01 TARJETA.
2500            03 CLAVEMAT PIC X(4).
2600            03 CREDITOS PIC 99.
2700            03 NOMBREMAT PIC A(36).
2800            03 NOMBREPROF PIC A(38).
2900         *LA CALIFICACION VA A ESTAR DADA EN FORMA ALFABETICA
```

```
3400           03 CALIF PIC 99.
3500         FD SALIDA
3600             RECORD CONTAINS 132 CHARACTERS
3700             DATA RECORD IS LINEA.
3800         01 LINEA PIC X(132).
3900         SD SORTED
4000             RECORD CONTAINS 80 CHARACTERS
4100             DATA RECORD IS ORDENA.
4200         01 ORDENA.
4300        *ESTE ARCHIVO Y EL DE ENTRADA DEBEN TENER LOS MISMOS CAMPOS
4400           03 CLAVEMAT PIC X(5).
4500           03 CREDITOS PIC 99.
4600           03 NOMBREMAT PIC A(36).
4700           03 NOMBREPROF PIC A(35).
4800           03 CALIF PIC 99.
4900        *ORIGINAL STORAGE SECTION.
5000        *DESCRIPCION DE TITULOS
5100         01 LINEA1.
5200           03 FILLER PIC X(52) VALUE SPACES.
5300           03 FILLER PIC X(27) VALUE "ALUMNO : AMPARO LUNEZ GAUNA".
5400           03 FILLER PIC X(53) VALUE SPACES.
5500         01 LINEA2.
5600           03 FILLER PIC X(8)  VALUE SPACES.
5700           03 FILLER PIC X(5)  VALUE "CLAVE"   .
5800           03 FILLER PIC X(7)  VALUE SPACES.
5900           03 FILLER PIC X(4)  VALUE "CRED".
6000           03 FILLER PIC X(18) VALUE SPACES.
6100           03 FILLER PIC X(17) VALUE "NOMBRE DE MATERIA".
6200           03 FILLER PIC X(26) VALUE SPACES.
6300           03 FILLER PIC X(19) VALUE "NOMBRE DEL PROFESOR".
6400           03 FILLER PIC X(15) VALUE SPACES.
```

```
05400            03 FILLER PIC X(5)  VALUE "CALIF".
05500            03 FILLER PIC X(7)  VALUE SPACES.
05600          01 LINEA3.
05700            03 FILLER PIC X(55) VALUE SPACES.
05800            03 FILLER PIC X(23) VALUE "TOTAL DE CREDITOS : ".
05900            03 TCREDITOS PIC 999.
06000            03 FILLER PIC X(54) VALUE SPACES.
06100          01 LINEA4.
06200            03 FILLER PIC X(55) VALUE SPACES.
06300            03 FILLER PIC X(20) VALUE "P R O M E D I O  :  ".
06400            03 PROMEDIO PIC 99.99.
06500            03 FILLER PIC X(52) VALUE SPACES.
06600          *DESCRIPCION DE LA LINEA DE SALIDA
06700          01 LINEADET.
06800            03 FILLER PIC X(8)  VALUE SPACES.
06900            03 CLAVEMAT PIC X(4).
07000            03 FILLER PIC X(9)  VALUE SPACES.
07100            03 CREDITOS PIC 99.
07200            03 FILLER PIC X(9)  VALUE SPACES.
07300            03 NOMBREMAT PIC A(36).
07400            03 FILLER PIC X(9)  VALUE SPACES.
07500            03 NOMBREPROF PIC A(36).
07600            03 FILLER PIC X(9)  VALUE SPACES.
07700            03 CALIF PIC AA.
07800            03 FILLER PIC X(6)  VALUE SPACES.
07900          *AQUI SE VA A CALCULAR EL PROMEDIO
08000          77 PROM PIC 99V99.
08100          *AQUI SE VA A CALCULAR EL TOTAL DE CREDITOS
08200          77 TCRED PIC 999 VALUE ZEROS.
08300          *AQUI SE VA CALCULAR EL TOTAL DE CALIFICACIONES
08400          77 TCALIF PIC 999 VALUE ZEROS.
```

```
      *AHORA SE VA A CALCULAR EL TOTAL DE MATERIAS
       77 MAT PIC 99 VALUE ZEROS.
       PROCEDURE DIVISION.
      *COMO SE VA A USAR UN "SORT" TODAS LA ETIQUETAS DEBEN IR SEGUIDAS
      *DE LA PALABRA "SECTION"
       INICIO SECTION.
       COMIENZO.
      *ABRIR ARCHIVOS
           SET ENTRADA (FILETYPE) TO 7.
           OPEN OUTPUT SALIDA.
      *ESCRIBE TITULOS
           WRITE LINEA FROM LINEA1 BEFORE ADVANCING 3 LINES.
           WRITE LINEA FROM LINEA2 BEFORE ADVANCING 3 LINES.
      *ORDENA POR NUMERO DE MATERIA USANDO EL ARCHIVO DE DATOS DE ENTRADA
      *(POR ESO NO SE ABRE ESE ARCHIVO)  Y UN PROCEDIMIENTO DE SALIDA
           SORT SORTED ON ASCENDING KEY NOMBREMAT USING ENTRADA OUTPUT P
      -    ROCEDURE SAL.
      *AL TERMINAR DE ORDENAR MUEVE CAMPOS E IMPRIME EL TOTAL DE CREDITOS
           MOVE T-CRED TO T-CREDITOS.
           WRITE LINEA FROM LINEA3 AFTER ADVANCING 5 LINES.
      *CALCULA EL PROMEDIO, MUEVE CAMPOS Y LO ESCRIBE
           COMPUTE PROM = TCALIF / MAT.
           MOVE PROM TO PROMEDIO.
           WRITE LINEA FROM LINEA4 AFTER ADVANCING 5 LINES.
      *CIERRA ARCHIVOS
           CLOSE SALIDA.
           STOP RUN.
       SAL SECTION.
       LEE.
      *PROCEDIMIENTO DE SALIDA
      *LEE EL ARCHIVO SORTEADO, TRADUCE LAS CALIFICACIONES A FORMA
```

```
12300         *NUMERICA
12400              RETURN SORTED AT END GO TO FIN.
12500              IF CALIF IN ORDENA EQUAL "43"
12600                ADD 10 TO TCALIF
12700              ELSE
12800              IF CALIF IN ORDENA EQUAL " 3"
12900              ADD 9 TO TCALIF
13000              ELSE
13100              IF CALIF IN ORDENA EQUAL " 5"
13200                ADD 5 TO TCALIF.
13300              IF CALIF IN ORDENA NOT EQUAL "NA"
13400         *ACUMULA CREDITOS
13500              ADD CREDITOS IN ORDENA TO TCRED
13600         *CUENTA EL NUMERO DE MATERIAS QUE HAY PARA PODER SACAR EL PROMEDIO
13700              ADD 1 TO MAT.
13800              MOVE CORRESPONDING ORDENA TO LINEASAL.
13900         *IMPRIME CADA REGISTRO
14000              WRITE LINEA FROM LINEASAL AFTER ADVANCING 2 LINES.
14100              GO TO LEE.
14200          FIN.
```

ALUMNO : AMPARO LOPEZ GAONA

CLAVE	CRED	NOMBRE DE MATERIA	NOMBRE DEL PROFESOR	CALIF
[illegible]	15	CALCULO I	MASSER	MB
[illegible]	10	ESTRUCTURA DE DATOS	NICKLAUS WIRTH	S
[illegible]	10	LENGUAJES DE PROGRAMACION I	TERRENCE W. PRATT	MB
[illegible]	10	LENGUAJES DE PROGRAMACION II	J. SAMMET	B
[illegible]	14	PROGRAMACION DE SISTEMAS	JEFFREY ULLMAN	MB
[illegible]	10	PROGRAMACION ESTRUCTURADA	DONALD KNUTH	B

TOTAL DE CREDITOS : 064

P R O M E D I O : 08.66

A.3 ALGOL

El alfabeto de ALGOL está constituido por los siguientes símbolos:

a) Caracteres alfabéticos:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

b) Caracteres numéricos:

0 1 2 3 4 5 6 7 8 9

c) Caracteres especiales:

* / + - = < > & . , ; : " [ ] ( ) % #

espacio en blanco.

Un identificador, es una combinación de letras y dígitos comenzando con una letra y su longitud no debe exceder a 63 caracteres.

Los comentarios, en ALGOL, pueden escribirse de cualquiera de las tres formas siguientes:

1. Empezando con la palabra reservada COMMENT;
2. Escribiendo un signo % y a continuación el comentario, ó
3. Entre la palabra reservada END y un punto y coma.

La sintaxis de ALGOL presentó dos inovaciones importantes, éstas fueron el uso de una gramática formal (BNF) para definir con precisión su sintaxis, y su estructura de bloque.

Un programa en ALGOL está compuesto por un conjunto de bloques anidados. Los subprogramas aparecen como declaraciones dentro de un bloque y a su vez tienen estructura de bloque.

Un bloque está compuesto de un conjunto de declaraciones seguida de una serie de proposiciones ejecutables, todo esto delimitado por las palabras BEGIN y END.

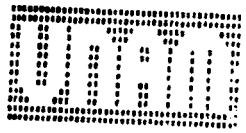
El programa principal es simplemente un bloque. Cada subprograma está compuesto de un encabezado, en el cual se especffica el nombre del subprograma y el nombre y tipo de cada uno de los parámetros formales; y un cuerpo que generalmente es un bloque. Además, cualquier serie de proposiciones puede ser agrupada en una proposición compuesta, simplemente encerrando la serie entre un BEGIN y un END.

Esta organización permite que la estructura del programa refleje directamente la estructura natural de muchos algoritmos. Los programas se escriben libremente dentro de las primeras 72 columnas de la línea. La simplicidad y claridad de las proposiciones y expresiones en ALGOL hacen que su estructura sea muy elegante.

A continuación se muestra un programa en ALGOL, que calcula el determinante de cualquier matriz, utilizando un procedimiento recursivo:

```
                                                                        00000100
{ FILE LST:=(KIND=PRINTER);                                             00000200
  OUTPUT:=LST;                                                          00000300
                                                                        00000400
{ PROCEDIMIENTO RECURSIVO PARA CALCULAR EL DETERMINANTE DE UNA MATRIZ   00000500
{ "MAT" DE N*N. TIENE DOS PARAMETROS: LA MATRIZ "MAT" Y LA DIMENSION "N" DE   00000600
{ LA MISMA (ESTA ES PASADA POR VALOR).                                  00000700
                                                                        00000800
REAL PROCEDURE DET(A,N);                                                00000900
  VALUE N;                                                              00001000
  INTEGER N; ARRAY A[*,*];                                              00001100
BEGIN                                                                   00001200
  REAL I,RES,P,Q,C;                                                     00001300
  IF N = 2 THEN DET:= A[0,0]*A[1,1] - A[0,1]*A[1,0]                     00001400
  { SI N ES 2 ENTONCES YA SE PUEDE CALCULAR EL DETERMINANTE Y TERMINA   00001500
  { EL PROCEDIMIENTO                                                    00001600
    ELSE                                                                00001700
    BEGIN                                                               00001800
      ARRAY MENORES[0:N-2,0:N-2];                                       00001900
  { EN EL ARREGLO "MENORES" SE VA A ALMACENAR LA MATRIZ QUE TIENE UNA   00002000
  { COLUMNA Y UN RENGLON MENOS.                                         00002100
      P := 1;                                                           00002200
      FOR I:=0 STEP 1 UNTIL N-1 DO                                      00002300
      BEGIN                                                             00002400
      FOR Q:=1 STEP 1 UNTIL N-1 DO                                      00002500
      BEGIN                                                             00002600
        FOR D:=0 STEP 1 UNTIL I-1 DO MENORES[Q-1,D]:=A[Q,D];            00002700
        FOR D:=I+1 STEP 1 UNTIL N-1 DO MENORES[Q-1,D-1]:=A[Q,D];        00002800
      END;                                                              00002900
      RES:= RES + A[0,I] * P * DET (MENORES,N-1);                       00003000
```

```
                    END;                                                    00003100
                    END;                                                    00003200
                  DETERM=B ;                                                00003300
                END;                                                        00003400
              END DET;                                                      00003500
              %                                                             00003600
              % PROGRAMA PRINCIPAL                                          00003700
              %                                                             00003800
               WHILE NOT NEND(LEER,7,N) DO                                  00003900
               BEGIN                                                        00004000
                 WRITE(OUT,'(/,'A MATRIZ A LA QUE SE ACUMULARA LA MATRIZ A LA QUE SE LE  00004100
                                    A VA A CALCULAR SU DETERMINANTE.')');   00004200
              %                                                             00004300
              % LECTURA DE LA MATRIZ                                        00004400
                 FOR I:=1 STEP 1 UNTIL N^1 DO                               00004500
                 READ(LEER,'(/,F8.3 STEP 1 UNTIL N^1 DO A[I,J])');          00004600
              % CALCULO DE LA MATRIZ                                        00004700
                 WRITE(OUT,'(/,X55,"LA MATRIZ ES :")');                     00004800
                 FOR I:=1 STEP 1 UNTIL N^1 DO                               00004900
                 WRITE(OUT,'(/,X44,I2,":",X5,(13,F8.3,X)/2,X44,           00005000
                       #)K I:=1 STEP 1 UNTIL N^1 DO A[I,J])');              00005100
                 WRITE(OUT,'(/,X53,"EL DETERMINANTE DE LA MATRIZ ES = ",  00005200
                                 /,X62,F8.3,")",DET(A,N))');    LAUJI LLAMA A CALCULAR EL 00005300
                                                          X DETERMINANTE    00005400
               END WHILE;                                                   00005500
             END.                                                           00005600
```



LA MATRIZ ES :

1 2 3

4 5 6

3 7 9

EL DETERMINANTE DE LA MATRIZ ES = :
-3.

A.4 PASCAL

El alfabeto de PASCAL está constituido por los siguientes símbolos:

a) Caracteres alfabéticos:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

b) Caracteres numéricos:

0 1 2 3 4 5 6 7 8 9

c) Caracteres especiales:

* / + - = < > : . , ; ' " ( ) [ ] { } @

espacios en blanco.

Un identificador en PASCAL, es una letra seguida de cualquier cantidad de letras o números. Una etiqueta es cualquier entero positivo, sin signo, declarado en la sección de etiquetas.

Los comentarios pueden ir en cualquier parte del programa y se especifican, escribiéndolos entre llaves ({ }) , o bien limitados por las siguientes parejas de símbolos: (* y *)

Un programa en PASCAL tiene un encabezado y un cuerpo llamado un bloque. En el encabezado va el nombre y la lista de archivos del programa. El bloque consta de seis secciones, donde cualquiera, excepto la última puede estar vacía. El orden en que deben aparecer las secciones es el siguiente:

1. Sección para declaración de etiquetas
2. Sección para definición de constantes
3. Sección para declaración de tipos de datos definidos por el programador.
4. Sección para declaración de variables
5. Sección para declaración de procedimientos y funciones
6. Sección de proposiciones.

Para escribir cualquier declaración o proposición se pueden utilizar las primeras 72 columnas de la línea, si se escribe más de una proposición se deben separar con un punto y coma.

A continuación se muestra un programa en PASCAL que efectúa la suma y la resta de dos polinomios dados, utilizando listas ligadas, es decir, utilizando el procedimiento NEW para crear dinámicamente los nodos de las listas.

```
100    (*   ESTE PROGRAMA CALCULA LA SUMA Y RESTA DE DOS POLINOMIOS DE CUAL        00000100
200         QUIER GRADO EN UNA VARIABLE "X".                                      00000200
300         AL LEER LOS POLINOMIOS LOS ORDENA POR EL GRADO DEL EXPONENTE          00000300
400         DE MENOR A MAYOR    *)                                                00000400
500    PROGRAM POLINOMIOS (INPUT,OUTPUT);                                         00000500
600    TYPE                                                                       00000600
700      APUNTA = ^NODO;                                                          00000700
800      NODO = RECORD                                                            00000800
900                COEF,EXP  : INTEGER;                                          00000900
1000               LIGA      : APUNTA                                            00001000
1100             END;                                                           00001100
1200                                                                            00001200
1300   VAR                                                                      00001300
1400     POLI,POL2,POL,BUFFER   : APUNTA;                                       00001400
1500     SIGNO                  : INTEGER;                                      00001500
1600                                                                            00001600
1700                                                                            00001700
1800   PROCEDURE ORDENA (SIGNO,COEF,EXP: INTEGER; VAR POLI: APUNTA);            00001800
1900                                                                            00001900
2000   (*   ESTE PROCEDIMIENTO SIRVE PARA ORDENAR LOS POLINOMIOS Y ADEMAS       00002000
2100        PARA EFECTUAR LAS OPERACIONES DE SUMA Y RESTA. SI "SIGNO" ES        00002100
2200        IGUAL A 1 ENTONCES EFECTUA LA SUMA Y SI ES -1 EFECTUA LA RESTA      00002200
2300        *)                                                                  00002300
2400                                                                            00002400
2500   VAR                                                                      00002500
2600     N1,N2,N3    :APUNTA;                                                   00002600
2700   BEGIN                                                                    00002700
2800     N2:= POLI;                                                             00002800
2900     N1 := N2^.LIGA;                                                        00002900
```

```
3400      BUFFER^.EXP := EXP1;                                          00003000
3500      WHILE V1^.EXP < EXP1 DO                                       00003100
3600      BEGIN                                                         00003200
3700        V2 := V1;                                                   00003300
3800        V1 := V2^.LIGA                                              00003400
3900      END;                                                          00003500
4000      IF (V1^.EXP = EXP1) AND (V1 <> BUFFER) THEN                   00003600
4100          V1^.COEF := V1^.COEF + SIGNO*COEF1                        00003700
4200      ELSE                                                          00003800
4300       BEGIN                                                        00003900
4400         NEW (V3);                                                  00004000
4500         WITH V3^ DO                                                00004100
4600         BEGIN                                                      00004200
4700           EXP := EXP1;                                             00004300
4800           COEF := COEF1;                                           00004400
4900           LIGA := V1;                                              00004500
5000         END;                                                       00004600
5100         V2^.LIGA := V3                                             00004700
5200       END                                                          00004800
5300    END;  (*  FIN DEL PROCEDIMIENTO ORDENA   *)                    00004900
5400                                                                   00005000
5500  PROCEDURE DEC(VAR POLI : APUNTA);                                00005100
5600                                                                   00005200
5700  VAR                                                              00005300
5800     COEF1,EXP1 : INTEGER;                                         00005400
5900  BEGIN                                                            00005500
6000     NEW(POLI);                                                    00005600
6100     NEW(BUFFER);                                                  00005700
6200     POLI^.LIGA:=BUFFER;                                           00005800
6300     READLN (COEF1,EXP1);                                          00005900
6400     WHILE COEF1 <> 0 DO                                           00006000
```

```
5400      BEGIN                                                          00005100
5500        ORDENA(P,COEF1,EXP1,POL1);                                   00005200
5600        READ (COEF1,EXP1)                                            00005300
5700      END;                                                           00005400
5800  END;  (* FIN DEL PROCEDIMIENTO LEE  *)                            00005500
5900                                                                     00005600
6000  PROCEDURE OPERACION(SIGNO : INTEGER ; P1,P2 : APUNTA);             00005700
6100                                                                     00005800
6200   (* AL EFECTUAR LA OPERACION DE SUMA Y RESTA ALMACENA EL RESULTADO 00005900
7000      DE LA MISMA EN EL SEGUNDO POLINOMIO   *)                       00007000
7100  BEGIN                                                              00007100
7200    WHILE P1 <> NIL DO                                               00007200
7300    BEGIN                                                            00007300
7400      ORDENA(SIGNO,P1^.COEF,P1^.EXP,P2);                             00007400
7500      P1 := P1^.LIGA;                                                00007500
7600    END                                                              00007600
7700  END;                                                               00007700
7800          (* FIN DEL PROCEDIMIENTO OPERACION  *)                    00007800
7900                                                                     00007900
8000  PROCEDURE IMPRIME ( POL : APUNTA);                                 00008000
8100  BEGIN                                                              00008100
8200    POL1:=POL;                                                       00008200
8300    WRITELN;                                                         00008300
8400    WRITELN;                                                         00008400
8500    WRITELN;                                                         00008500
8600    WHILE POL <> NIL DO                                              00008600
8700    BEGIN                                                            00008700
8800      IF POL^.COEF <> 0 THEN                                         00008800
8900      BEGIN                                                          00008900
9000        IF POL^.COEF < 0 THEN WRITE('  ') ELSE WRITE (' + ');        00009000
9100        WRITE (POL^.COEF:3, ' X ',POL^.EXP:3)                        00009100
```

```
9200        END;
9300        POLI:= POLIRESTA;
9400      END;
9500      WRITELN;
9600      WRITELN;
9700        WRITELN
9800    END; (*IMPRIME*)
9900
10000   (*   PROGRAMA  PRINCIPAL   *)
10100   BEGIN
10200     SEE (POLI1);
10300     SEE (POLI2);
10400      WRITELN(' POLINOMIO P1');
10500     IMPRIME (POLI1);
10600     WRITELN;
10700      WRITELN(' POLINOMIO P2');
10800     IMPRIME (POLI2);
10900      WRITELN(' POLINOMIO P*Q ');
11000      WRITELN;
11100     OPERACION (1,POLI2,POLI1);
11200     IMPRIME  (POLI1);
11300      WRITELN(' POLINOMIO P-Q ');
11400     WRITELN ;
11500
11600     (* CUANDO EL RESULTADO DE LA SUMA SE ALMACENO EN EL POLI1 SE
11700        EFECTUA LA OPERACION DE RESTA DOS VECES, UNA PARA QUE SE
11800        OBTENGAN LOS POLINOMIOS ORIGINALES Y LA OTRA PARA EFECTUAR
11900        LA RESTA DESEADA   *)
12000
12100     OPERACION (-1,POLI2,POLI1);
12200     OPERACION (-1,POLI2,POLI1);
```

```
00009200
00009300
00009400
00009500
00009600
00009700
00009800
00009900
00010000
00010100
00010200
00010300
00010400
00010500
00010600
00010700
00010800
00010900
00011000
00011100
00011200
00011300
00011400
00011500
00011600
00011700
00011800
00011900
00012000
00012100
00012200
```

```
1230      INFRAME (PS-11)
1240      END
```

```
00312300
00312400
```

A.5 SNOBOL

El alfabeto de SNOBOL4, está constituido de los siguientes símbolos:

a) Caracteres alfabéticos:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

b) Caracteres numéricos:

0 1 2 3 4 5 6 7 8 9

c) Caracteres especiales:

* / + - . _ ; $ & ' : ( ) ? # % < >

espacios en blanco

Un identificador en SNOBOL4, es una letra seguida de cualquier cantidad de caracteres alfanuméricos, además, puede contener puntos (".") o guiones bajos ("_").

Un programa en SNOBOL4, consta de una serie de proposiciones terminadas con la palabra END. EL comentario en una línea se indica escribiendo un asterisco ("*") en la columna uno, la línea de continuación se específica con un punto o con un signo de más ("+"). Todas las proposiciones deben empezar en la columna 2 y terminar en la 72.

El aspecto más llamativo de la sintaxis de SNOBOL4 es la falta de cualquier distinción sintáctica entre el cuerpo de los subprogramas y el pro-

grama principal. Un programa consta básicamente, de una lista de proposiciones, y no existe una forma de determinar sintácticamente cuando una proposición particular forma parte del programa principal o de una o varias subrutinas.

Esta estructura caótica permite que, sean fácilmente, integrados al programa nuevos segmentos del programa o subprogramas que son traducidos a código ejecutable al momento de ejecución.

Las proposiciones individuales tienen una base sintáctica común. La sintaxis básica es como sigue:

⟨etiqueta⟩ ⟨cuerda⟩ ⟨patrón⟩ = ⟨cuerda reemplazante⟩ : ⟨GO TO⟩

La ⟨etiqueta⟩ es un identificador, que empieza en la columna uno de la línea. La ⟨cuerda⟩ y ⟨cuerda reemplazante⟩ son definidas por expresiones, cuyo valor es una cuerda. El ⟨patrón⟩ es definido por una expresión, cuyo valor es un patrón. El ⟨GO TO⟩ puede especificar un salto incondicional o condicional. Los campos de ⟨etiqueta⟩ y : ⟨GO TO⟩ son opcionales en todas las proposiciones.

Las proposiciones de asignación son aquellas en donde el ⟨patrón⟩ se omite. El patrón de igualdad simple sin reemplazamiento es especificado por la omisión de = ⟨cuerda reemplazante⟩ . Para un GO TO incondicional se omite todo, excepto el campo ⟨GO TO⟩ . SNOBOL4 no proporciona proposiciones estructuradas de ninguna forma.

A continuación se muestra un programa en SNOBOL4 que ordena una lista de N elementos utilizando el Sort de la Burbuja.

```
BURBUJA  (10/13/81)                                                    12:45 PM TUESDAY, OCTOBER 13, 1981

      *
      * SORT DE LA BURBUJA
      *
              DEFINE ('ORDENA(V)I')
              DEFINE ('INTERCAMBIA(I)TEMP')
              DEFINE ('BURBUJA(J)')
      *
      *       LA EL NUMERO DE DATOS QUE SERAN ORDENADOS
      *
              N   = TRIM(INPUT)               :F(ERROR)
              A   = ARRAY(N)
      *
      *       LEE LOS DATOS
      *
      *
      LEER    I   = I + 1
              A<I> = TRIM(INPUT)              :F(CONTINUA)S(LEER)
      *
      *        ORDENA LA LISTA
      *
      CONTINUA  ORDENA(N)
      *
      *        IMPRIME LA LISTA ORDENADA
      *
              N   = 1
      IMPRIME  OUTPUT = A<N>                  :F(END)
              N   = N + 1                     :S(IMPRIME)
      *
      *       FUNCIONES
```

```
3100    *
3200    ORDENA  I   = LT(I+N - 1) I + 1          :F(RETURN)
3300            LGT(A<I>,A<I + 1>)          :F(ORDENA)
3400            INTERCAMBIO(I)
3500            BURBUJA (I)                 :(ORDENA)
3600    *
3700    INTERCAMBIO   TEMP = A<I>
3800            A<I> = A<I + 1>
3900            A<I + 1> = TEMP             :(RETURN)
4000    *
4100    BURBUJA   J = GT(J,I) J - 1         :F(ORDENA)
4200            LGT(A<J>,A<J + 1>)          :F(RETURN)
4300            INTERCAMBIO (J)                 :(BURBUJA)
4400    *
4500    END
4600    IS
4700    ANPARO
4800    MYRIAM
4900    DURANGUAY
5000    JAVIER
5100    FRANCISCO
5200    JUAN PABLO
5300    MARINA
5400    ROBERTO
5500    LUIS ANGEL
5600    JOSE LUNADIO
```

Bibliografía

1. AMMAN, U.
"On Code generation in PASCAL compiler"
Software: Practice and Experience, 7 (1977)

2. BRENDER, F RONALD and NASSI, ISAAC R
"ADA programming in the 80's"
Computer, June 1981

3. DONOVAN, JOHN
"Systems Programming"
International Student Edition
Mc. Graw-Hill, Kogakusha, L+d. (1972)

4. GREGORY, DONALD J
"ALGOL on the B6700: A complete Primer"
Volume I
Copyright C Donald J. Gregory (1976)

5. GRISWOLD. R, J. POAGE and POLONSKY
"The SNOBOL4 Programming Language"
Prentice-Hall, Englewood Cliffs, N.J. (1971)

6. GROGONO, PETER
"Programming in PASCAL"
Addison-Wesley (1979)

7. HOPCROFT, JOHN E and ULLMAN, JEFFREY D.
"Formal Languages and their relation to Automata "
Addison-Wesley Publishing Company (1969)

8. JAMES JOHN
"What is FORTH? A introduction"
BYTE: August 1980, Volume 5 Number 8.

9. JENSEN, KATHLEEN; WIRTH NIKLAUSS
"PASCAL user manual and report"
Springer-Verlag, (1978)

10. KRULL, FRED N.
"Experience with ILIAD. A High-Level Process Control Language"
Communications ACM
February 1981, Volume 24 Number 2

11. MAURER, W.D.
"An introduction to BNF"
BYTE: January 1979

12. NOLAN, RICHARD
"FORTRAN IV Computing and Applications"
Addison-Wesley Publishing Company (1971)

13. ORGANICK, E.I; FORSYTHE, A.I; PLUMMER R. P
"Programming Language Structures"
Academic Press, Inc. (1975)

14. PETERSON, GILES and BUDGOR, AARON
"The computer Language MATHSY and
Aplications to Solid State Physics"
Communications ACM
August 1980, Volume 23, Number 8

15. PRATT, TERRENCE W.
"Programming Languages: Design and Implementation"
Prentice-Hall, Inc. Englewood Ciffs, N.J. (1975)

16. SAMMET, JEAN
"Programming Languages: History and Future"
Communications ACM
25th Anniversary Issue
July 1972, Volume 15, Number 7

17. STERN, NANCY B; and STERN, ROBERT A
"COBOL Programming"
Second Edition
John Wiley & Sons, Ins (1977)

18. WIRTH, N.
"Algorithms + Data Structures = Programs"
Prentice-Hall (1976).