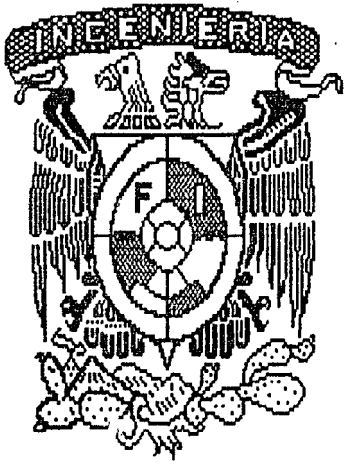


FACULTAD DE INGENIERIA

DESARROLLO DE UNA HERRAMIENTA DE SOFTWARE PARA LA OBTENCION DE LOS PARAMETROS DE HALSTEAD PARA PROGRAMAS ESCRITOS EN MODULA-2

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

I N D I C E

APENDICE: DESCRIPCION DEL LENGUAJE MODULA-2

BIBLIOGRAFIA

INTRODUCCION

Un aspecto al que hasta hace relativamente poco tiempo no se le había dado la suficiente importancia es la medición de programas, esto es, la medición de las características de la representación de algoritmos en un lenguaje de programación. Debemos considerar que tales representaciones poseen propiedades que vale la pena investigar, ya que la expresión de algoritmos en un lenguaje no merece ser medida únicamente en, por ejemplo, las líneas de código empleadas.

Uno de los pioneros en este campo fue Maurice Halstead, quien después de investigar sobre el particular propuso a finales de los años setentas la teoría que denominó "ciencia de la programación" (software science) en la que argumenta que los programas tienen propiedades que pueden ser cuantificadas; la principal ventaja de esta teoría es que a diferencia de otras características cualitativas, permite la obtención de los medidas o estimadores de dichas propiedades por medios automáticos y de esta manera también podemos corroborar la validez de sus hipótesis.

De esta manera, al contar con indicadores de las características de programas tendremos mejores elementos para su evaluación y control durante su desarrollo y mantenimiento. El poder medir las propiedades de la implementación de un algoritmo también nos permitirá obtener programas más confiables y eficientes.

El objeto de este trabajo es entonces implementar un sistema de software que nos permita obtener los principales indicadores de la teoría de Halstead -longitud, volumen, nivel, contenido intelectual y esfuerzo de programación- para programas escritos en Modula-2, para así contar con resultados que nos permitan comprobar las hipótesis de Halstead y al mismo tiempo presentar un ejemplo del desarrollo de un sistema en el más reciente lenguaje de programación creado por Niklaus Wirth, y que al incorporar las mejores características de sus predecesores (Pascal y Modula-1) es uno de los lenguajes más poderosos para el desarrollo de programas de aplicación general, junto a una metodología de diseño que aprovecha eficazmente los atributos de este lenguaje: el diseño modular orientado a objetos.

La elección del lenguaje de programación Modula-2 obedece a dos razones. En primer lugar, la motivación que todo ingeniero en computación debe tener por buscar y utilizar las herramientas más adecuadas para la realización eficaz y eficiente de su trabajo. No podemos permitir que la inercia nos haga utilizar herramientas obsoletas, o que sigamos recurriendo a metodologías que no satisfacen adecuadamente la calidad que el diseño y construcción de software que nuestro medio y nuestra sociedad requieren en la

actualidad. La programación es un arte constructivo; a mejores métodos y herramientas, mejores y más elegantes soluciones.
En segundo lugar, consideramos que Modula-2 será en la década de los 8Ø lo que Pascal fue en la de los 7Ø: un lenguaje de tipo imperativo eficaz y elegante, que permite el desarrollo de programas de manera más sencilla y confiable.

En el capítulo 1 describimos lo que en Ingeniería de software se denomina "ciclo de vida" y que servirá para ubicar el momento más adecuado para la medición de programas y en consecuencia saber cómo y cuándo puede reportar su mayor beneficio.
En el capítulo 2 describimos algunos métodos de medición propuestos por varios autores y presentamos las hipótesis de Halstead con las que desarrolló una serie de estimadores de las propiedades de programas y que comprenden el fundamento de la teoría conocida como "software science".
En el capítulo 3 se presenta el diseño e implementación de una herramienta de software que servirá para la obtención de la longitud, volumen, nivel, contenido intelectual y esfuerzo de programación propuestas por Halstead y de una medida de longitud alterna propuesta por Jensen, para programas escritos en Modula 2.
En el capítulo 4 se presentan las conclusiones a las que llegamos y sugerimos las aplicaciones que la medición de programas en general y la teoría de Halstead en particular pueden tener.
Finalmente incluímos un apéndice que describe el lenguaje Modula 2, considerando que es un lenguaje relativamente nuevo y por ello no ha tenido una gran difusión en nuestro medio.

Para la formación del texto y la elaboración de las ilustraciones se empleó un computador Apple II+, a excepción de los diagramas del capitulo 3 y la ilustración del apèndice realizadas con MacPaint en una Apple Macintosh amablemente facilitada por Eric Kelly.

Agradecimientos

A nuestros padres.

A nuestros maestros de la Facultad de Ingeniería y de la ENEP Aragón, donde uno de nosotros comenzó la carrera de ingeniería.

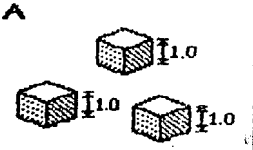
A nuestro director de tesis, Gustavo Origel Coutiño, a Luis Cordero Borboa y a José Miguel Martínez Alcaráz por su gran apoyo para la realización de este trabajo. A todos ellos nuestra gratitud y agradecimiento más profundo.

MAPC
SFRP

Noviembre de 1985

CAPITULO 1:

LA MEDICION DE SOFTWARE EN EL CICLO DE VIDA



En este capitulo se describen brevemente las fases principales del proceso de desarrollo de proyectos que en la Ingeniería de Software se denomina **ciclo de vida**, y que servirá como marco de referencia para situar el lugar y la importancia de la medición de software. El ciclo de vida lo podríamos representar de manera muy simple mediante un diagrama de bloques como lo muestra la fig. 1.1.

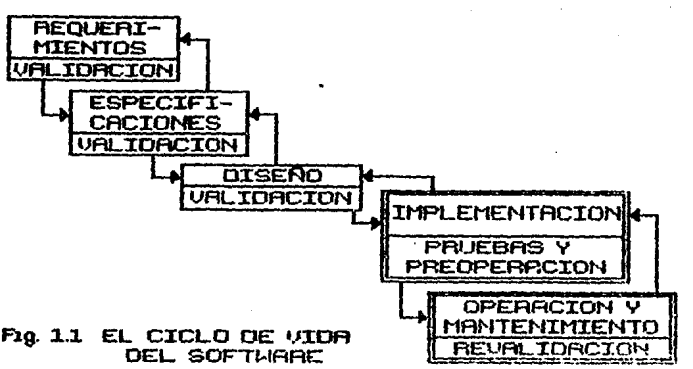


Fig. 1.1 EL CICLO DE VIDA DEL SOFTWARE

El concepto de ciclo de vida surge de la naturaleza evolutiva de los sistemas de software y nos facilita un marco de referencia uniforme para la solución de problemas mediante metodologías y herramientas que están en evolución continua. Su empleo permite el desarrollo de sistemas confiables y readaptables a un costo razonable.

En la fig. 1.1 hemos subrayado las etapas en las que estimamos que la medición de programas puede aportar su máximo beneficio: si podemos contar con instrumentos de evaluación más poderosos podremos vigilar más eficazmente las propiedades que hacen confiables y readaptables a los programas que conforman un sistema.

Además el proceso de desarrollo es iterativo: durante la etapa de diseño puede hacerse necesario modificar las especificaciones formales en el caso de encontrar dificultades que imposibilitaran cumplir alguna restricción operacional impuesta en la etapa anterior. Del mismo modo, puede ser necesario replantear parte del diseño durante la fase de implementación por limitaciones de hardware que no hubieran podido preverse. En resumen, la definición y diseño de un sistema se obtiene de manera iterativa y gradual.

ETAPAS DEL CICLO DE VIDA

Requerimientos y especificaciones

Los errores mas costosos en el proceso de desarrollo, ocurren generalmente durante esta etapa. En ocasiones (principalmente en grupos de programación sin experiencia) hay precipitación por comenzar la implementación sin un entendimiento ni una definición

clara de lo que el usuario realmente requiere y lo que el grupo implementa.

El primer paso en esta etapa es establecer un conjunto de requerimientos que resultan de extensas pláticas entre quien solicita la construcción del sistema (el cliente o usuario) y el grupo de desarrollo. El lenguaje empleado para expresar los requerimientos debe ser claro y conciso, de manera que refleje la comunicación precisa entre usuario y grupo de desarrollo; tanto las limitaciones del sistema, como su rango de aplicación deben ser delimitadas, así como la máquina o máquinas en las que se implementará. El método para la entrada de datos al sistema, así como las salidas (reportes, gráficas, etc.) y las restricciones de operación deben acordarse sin ambiguedad. En suma, los requerimientos y especificaciones indicarán lo que el sistema debe hacer, y no cómo lo deberá hacer. Además a partir de estas especificaciones también se conocerán los recursos humanos, materiales y por supuesto económicos que se requerirán, lo que constituye la base del estudio de viabilidad del sistema y que finalmente determina si el sistema podrá construirse.

Por otra parte, la definición de requerimientos conforma la base para un contrato inicial entre el usuario y el grupo de desarrollo; este último traduce los requerimientos informales en un documento formal de especificaciones, que forma la parte fundamental del contrato final. Este documento establece un modelo conceptual general del sistema; dado que el cliente puede tener un conocimiento técnico limitado, se debe tratar de alcanzar la máxima claridad y legibilidad sin recurrir a términos técnicos.

(En el futuro se dispondrà de herramientas que ayudaràn a expresar con mayor rigidez las especificaciones de un sistema: se han desarrollado lenguajes de especificaciòn con lo que se automatiza la generaciòn del documento final. Sin embargo estas herramientas aùn se encuentran en investigaciòn.)

En la actualidad el software es el componente màs costoso de un sistema de còmputo y esta tendencia persistirá por mucho tiempo. La estimaciòn del costo, no es una ciencia exacta, dado que hay que confrontar variables tècnicas, humanas, ambientales, y aun polìticas.

El mejor momento para establecer el costo del software es al final del proyecto, donde evidentemente el costo estimado serìa exactamente el costo real; desde luego, esto es totalmente impracticable, dado que el cliente necesita una estimaciòn confiable al inicio del proyecto.

Antes de que podamos estimar el costo es esencial tener un entendimiento de los factores que afectan la productividad. En lo concerniente al hardware, esta es generalmente medida con respecto al nùmero de unidades que se producen en un intervalo de tiempo predefinido. Los costos del hardware asociados con la planeaciòn, anàlisis y diseño, no son incluìdos en la productividad del hardware, sino amortizados sobre el tiempo total de fabricaciòn. Sin embargo este esquema no funciona para medir la productividad de software, dado que el desarrollo de tales sistemas es un evento que sòlo ocurre una vez. Còmo debemos medir la planeaciòn del software, el anàlisis o la productividad del diseño?

La productividad de software puede ser determinada mediante la medición de ciertas características de los programas. La medida màs simple y controversial es determinar el nùmero promedio de lìneas de còdigo por persona por unidad de tiempo, lo que no puede ser llevado a cabo hasta que el proyecto se haya terminado. Sin embargo los costos de nuevos proyectos pueden ser estimados basàndose en costos històricos para proyectos similares; la parte compleja consiste en la estimación del nùmero de lìneas de còdigo que se tendràn que escribir.

Por ejemplo, supongamos que acabamos de terminar un proyecto en el que se entregaron 4600 lìneas de còdigo fuente, de las cuales, 900 de ellas sirvieron para pruebas y simulación. Un desglose del proyecto indicarìa lo siguiente:

Requerimientos	1.0 personas-mes.
Especificaciones	1.0 persona-mes.
Diseño	2.0 personas-mes.
Implementación	4.5 personas-mes.
Pruebas y preoperación	4.0 personas-mes.
TOTAL	12.5 personas-mes.

La productividad del grupo de desarrollo que trabajò en este proyecto es 3700/12.5 = 296 lìneas de còdigo por persona-mes. Si este grupo decidiera trabajar en un proyecto de alcance y dificultad similar, donde el nùmero de lìneas a entregar sea estimado en 6200, podremos calcular que llevarà 6200/296 = 21 personas-mes por proyecto.

Ademàs existen otros factores que afectan la productividad de software. El personal asignado al proyecto constituye un factor principal así como tamaño del grupo, su experiencia, y aùn la

personalidad de cada individuo. Otro factor es la complejidad del problema. Si hay cambios en los requerimientos o en el diseño, la productividad se verá afectada negativamente. El lenguaje de programación elegido y las herramientas con que se cuente también afectan a la productividad, así como la disponibilidad de equipo y facilidades para el trabajo, cuestiones no solo difíciles sino a veces casi imposibles de cuantificar.

Valdría considerar la bondad del método de estimación basado en costos históricos, pero tomando parámetros que no se basen simplemente en el número de líneas de código fuente, sino en cantidades más significativas como las que ofrece la ciencia del software propuesta por Maurice Halstead y que se describen en el capítulo 2.

Diseño

El propósito principal de esta fase es cómo construir el sistema propuesto. Dado que el proceso de desarrollo es iterativo, el grupo de diseño puede recomendar que algunas especificaciones sean modificadas, lo que suele ocurrir cuando se detecta alguna inconsistencia o restricción funcional. Las decisiones que se lleven a cabo en esta fase y las técnicas que se empleen afectarán profundamente el costo del mantenimiento posterior. Además la confiabilidad del sistema depende fuertemente de la calidad de su diseño. Con el advenimiento de lenguajes mucho mas poderosos para el desarrollo de sistemas, como Modula-2, se ha desarrollado un nuevo método de diseño orientado a objetos; el diseño de software conleva un proceso de abstracción. Los

objetos y las operaciones que actuan sobre ellos en la vida real deben ser transladados a los correspondientes objetos y operaciones en el sistema de software.

El diseño modular orientado a objetos hace innecesario que el diseñador del sistema tenga que mapear los objetos del dominio del problema a las estructuras de datos y de control presentes en el lenguaje de implementación. En su lugar, el diseñador puede crear sus propios tipos abstractos y abstracciones funcionales para mapear eficazmente el problema real con estas abstracciones creadas por el propio diseñador. Una ventaja adicional es el desacoplamiento entre los detalles de la representación de los datos y objetos empleados; estos detalles pueden ser modificados cuantas veces sea necesario sin el peligro de inducir efectos "en cascada" en el sistema. La estructura de módulo de Modula-2 con sus tipos opacos y la completa separación entre la especificación e implementación de estos módulos hacen posible el diseño modular y orientado a objetos. En el capítulo 3 describiremos con mayor detalle esta metodología al desarrollar una herramienta de software para la obtención de los parámetros de Halstead.

Implementación

Tal vez la etapa mas sencilla del ciclo de vida sea aquella en la que se traducen las especificaciones de diseño a los programas que contitituyen el sistema, siempre y cuando la metodología de diseño aproveche las ventajas que ofrezca el lenguaje empleado. Antes de la existencia de Modula-2, cuando un proyecto se dividía para su realización entre un equipo de varias personas, existía un problema muy peculiar debido a la compilación separada de los

elementos que conformaban el sistema (desarrollados por personas diferentes) con frecuencia no se respetaban las interfaces establecidas durante la etapa de diseño y a través de las cuales se comunicaban estos elementos, problema que debía ser resuelto en una fase denominada "integración de sistemas" y que afectaba en mayor o menor medida el grado de acoplamiento del sistema, por no mencionar su confiabilidad. El lenguaje Modula-2 remedia esta situación puesto que el compilador verifica la consistencia de las interfaces desde el momento mismo de la compilación de un módulo. En el apéndice 1 se describe con mayor amplitud esta característica.

Pruebas y preoperación

Podemos afirmar que en todo sistema no trivial de software tendrá errores. Tales sistemas consisten típicamente de una compleja interconexión lógica y física de sus subsistemas; aunque cada uno de ellos pueda haber sido probado individualmente, el sistema integrado puede aún tener imperfecciones.

La validación de un sistema es un proceso continuo. El sistema y sus componentes deben ser probados exhaustivamente antes de liberarlo al usuario. No obstante, las pruebas no demuestran que un programa sea absolutamente correcto, sino que se demostrará que funciona para un determinado conjunto de datos. Aún así, puede haber errores despues de completar numerosas pruebas: únicamente puede detectarse la presencia de errores mediante la prueba de programas, nunca la ausencia total de los mismos, lo que no significa que esta sea una tarea siempre insatisfecha, sino que debe ser planeada y ejecutada con gran cuidado.

Implantación

Una vez terminadas las pruebas por parte del equipo que construyó el sistema, este se entrega al usuario, junto con los manuales de operación y referencia, que describirán de manera clara los procedimientos que se deben seguir para operarlo (además de un adiestramiento al usuario, que dependerá de la extensión o complejidad del sistema). De esta manera el usuario comprobará que el sistema realiza las funciones que fueron acordadas durante la especificación del mismo. Conjuntamente, el grupo de desarrollo observará el comportamiento y la eficiencia del sistema, cuando este es sometido a las cargas reales de trabajo, lo que permite observar las partes que pudierán (o debieran) modificarse para optimizar su funcionamiento.

Operación y mantenimiento

Una vez que el sistema se libera, comienza la fase de mantenimiento. Las reparaciones o los cambios a los programas se consideran para nuevas versiones del sistema. Con estas modificaciones, gradualmente se obtiene un sistema más eficiente y con menos errores, todo esto condicionado a la calidad del diseño y de la programación: en ocasiones, una mejora puede resultar imposible o de costo prohibitivo, o la ineficacia de la implementación tan grande que resulte necesario construirlo nuevamente desde el principio. Es en estos casos, cuando los errores u omisiones en la aplicación de un método se hacen dolorosamente patentes.

LA MEDICION DE SOFTWARE EN EL CICLO DE VIDA

La medición de características del software puede proporcionar información valiosa en el ciclo de vida. Se puede emplear un sistema de medición del código fuente de los programas, para predecir el número de errores que pueden encontrarse en pruebas subsecuentes o la dificultad involucrada en la modificación de un módulo. Dado el valor potencial de predicción, la métrica del software puede ser empleada en tres formas:

- Como herramienta de control. La métrica provèe varios tipos de información: puede ser empleada para predecir fallas, como ya se mencionó anteriormente; puede usarse para estimar el costo y el tamaño de un sistema o incluso para calcular la productividad. Tales mediciones permiten estimar requerimiento de recursos, problemas futuros o el grado de progreso de un sistema.
- Como indicador de la calidad del software: el poder contar con estos criterios permitiría fijar normas para adquisición de software o como una guía a problemas potenciales durante la validación y verificación del software.
- Como realimentación para el grupo de desarrollo: cuando alguna parte del sistema resulta tener un grado alto de complejidad, el grupo de desarrollo debe reescribirlo hasta hacer que se encuentre dentro de límites adecuados.

La medición de la complejidad del software que ha recibido más atención es la denominada Ciencia del Software, propuesta por Maurice Halstead, que argumenta que los algoritmos tienen características medibles análogas a las leyes físicas, obtenibles mediante la cuantificación de operadores y operandos y sus

frecuencias totales. A partir de estas cuatro cantidades, Halstead desarrolló medidas para la longitud total de un programa, volumen potencial mínimo de un algoritmo, volumen real de un algoritmo expresado en un lenguaje determinado, nivel de programa (la dificultad para entender un programa), nivel del lenguaje (una constante para un lenguaje determinado), esfuerzo de programación (el número de discriminaciones mentales requeridas para escribir un programa), tiempo de programación, y el número de fallas que puede existir en un sistema.

Esta teoría ha sido objeto de considerables investigaciónes y evaluaciónes. Coeficientes de correlación mayores a Ø.9 se han reportado entre las predicciones de Halstead y las mediciones reales para varios indicadores, y ha probado su utilidad en la práctica, principalmente en la realimentación a los responsables de la progamación durante las fases de implementación y mantenimiento de un sistema para indicarles la complejidad de su código, en la forma mencionada anteriormente. Bell y Sullivan sugieren que un límite razonable para la medida de longitud de Halstead es 260, dado que han encontrado que algoritmos con valores superiores a esta cifra contienen típicamente un error. El desarrollo de esta teoría se explica en el siguiente capítulo, y el desarrollo de una herramienta que permita la comprobación de sus resultados para programas escritos en Modula-2 es el núcleo del capítulo 3.


CAPITULO 2:
METODOS DE MEDICION
DE SOFTWARE

En este capítulo se describen algunos métodos de medición propuestos por varios autores para la medición de software, específicamente aquellos que más se prestan a su obtención automática y que como ya mencionamos, pueden ser de gran utilidad en el ciclo de vida del software, puesto que con ellos tendremos elementos para el control y evaluación de propiedades y características de los programas que conforman un sistema.

Así pues, describiremos la medida de complejidad de McCabe, el concepto de banda propuesto por Belady, la medida de intervalo propuesta por Basili y los principales indicadores de la ciencia del software propuesto por Maurice Halstead. Así mismo, haremos algunas consideraciones sobre cohesión y acoplamiento.

MEDIDA DE COMPLEJIDAD DE McCABE

La medida de complejidad propuesta por Thomas McCabe (que denotaremos como MC) se define como el número ciclomático de la gráfica del flujo de control de un programa e indica el número de trayectorias de control linealmente independientes. Esto permite estimar la cantidad de pruebas que pueden ser necesarias para validar dicho programa (o un módulo que sea parte de un programa). McCabe ha observado que este indicador puede ser obtenido a partir de la ecuación MC = D + 1, donde D es número de decisiones en el programa, esto es, la cuenta de las proposiciones condicionales, ciclos iterativos y operadores booleanos tales como AND, OR, NOT, etc. En otras palabras este indicador permite estimar la dificultad para la verificación de un programa y finalmente un estimador de su confiabilidad. McCabe propone que un valor menor o igual a 7 para MC es un límite razonable dentro del cual la complejidad de un programa es manejable.

BANDA

B. L. A. Belady propuso la medida denominada banda, basándose en el argumento de que es más difícil desarrollar y mantener programas que emplean estructuras de control profundamente anidadas, que en aquellos en que se emplean a menor profundidad. De esta manera la banda B proporciona un parámetro que indica el nivel promedio de anidamiento de las estructuras de control, y se define como

$$B = \frac{(\text{SUMATORIA } i \; L(i))}{(\text{número de nodos en la gráfica de control})}$$

donde L(i) denota el número de nodos de nivel i. De esta manera, un programa sin estructuras anidadas tendría B = 1, y programas con estructuras anidadas valores mayores que 1.

INTERVALO (SPAN)

Un factor importante que afecta la complejidad de un programa es la configuración de los datos en el mismo. La complejidad de un programa puede relacionarse con la forma en que los datos están organizados y distribuídos.

V. Basili describe la medida de intervalo como el número promedio de proposiciones que aparecen entre dos referencias a un mismo identificador; entre mayor sea el intervalo, mayor es la complejidad del programa, y mayor el esfuerzo que requerirá para su mantenimiento futuro.

CIENCIA DEL SOFTWARE (SOFTWARE SCIENCE)

La teoría de la ciencia del software, desarrollada por Maurice Halstead entre 1972 y 1979 sostiene que los algoritmos expresados en cualquier lenguaje de programación tienen propiedades medibles y propone una serie de medidas o indicadores obtenidos a partir de la cuenta de operadores y operandos que aparecen en un programa así como la frecuencia con la que estos aparecen. De esta manera, las propiedades de un programa se obtienen a partir de las siguientes definiciones:

Sean

n1 = el número de operadores distintos,

n2 = el número de operandos distintos,

N1 = el número total de operadores,

N2 = el número total de operandos,

$f_{1,j}$ = el número de apariciones del j-ésimo operador más frecuentemente empleado, donde j = 1,2,...,n1

$f_{2,j}$ = el número de apariciones del j-ésimo operando más frecuentemente empleado, donde j = 1,2,...,n2

a partir de las cuales, se define el vocabulario n como:

$$n = n1 + n2$$

y la longitud de implementación N como:

$$N = N1 + N2$$

De estas definiciones encontramos las siguientes relaciones:

$$N1 = \mathbf{SUM} (f_{1,j}),\ j = 1,n1 \qquad (2.1)$$

(Que se lee: "N1 es igual a la sumatoria de $f_{1,j}$ para j de 1 hasta n1".)

$$N2 = \mathbf{SUM} (f_{2,j}),\ j = 1,n2 \qquad (2.2)$$

$$N = \mathbf{SUM} (\mathbf{SUM} (f_{i,j}),j = 1,ni),\ i = 1,2 \qquad (2.3)$$

Como ejemplo, encontraremos n1, n2, N1 y N2 para el algoritmo de Euclides que encuentra el máximo divisor común de dos números escrito en Modula-2:

```
PROCEDURE MDC ( A,B: INTEGER ): INTEGER;
VAR R,G: INTEGER;

BEGIN
  IF A = 0 THEN RETURN B END;
  IF B = 0 THEN RETURN A END;
  LOOP
    G := A DIV B; R := A - B * G;
    IF R = 0 THEN RETURN B END;
    A := B; B := R
  END
END MDC;
```

La clasificaciòn de operadores y operandos se muestra en las tablas 1 y 2 respectivamente. Para esto, consideramos que el símbolo ";" es un operador dado que también determina el flujo del programa. Del mismo modo, las estructuras de control como IF..THEN, LOOP..END, REPEAT..UNTIL, etc. son consideradas operadores.

La clasificacion de operandos es intuitiva y no requiere de mayor explicaciòn.

OPERADOR	j	f 1,j
;	1	6
:=	2	4
IF..THEN	3	3
RETURN	4	3
=	5	3
DIV	6	1
-	7	1
*	8	1
LOOP..END	9	1
BEGIN..END	10	1
	n1=10	N1=24

TABLA 1: Clasificaciòn de los operadores del algoritmo MDC.

OPERANDOS	j	$f_{2,j}$
B	1	7
A	2	5
R	3	3
Ø	4	3
G	5	2
	n2=5	N2=2Ø

TABLA 2: Clasificación de los operandos del algoritmo MDC.

Longitud de un programa

Una cuerda de longitud N conformada por elementos de un vocabulario de n elementos diferentes, al seguir para su construcción un conjunto de reglas definidas por la sintaxis del lenguaje debe obedecer varias restricciones. En primer lugar, la condición de que cada uno de los n elementos deba aparecer al menos una vez garantiza que:

$$n <= N \qquad (2.4)$$

por lo que existe un límite inferior de N en términos de n.

Por otra parte, si se cumple una condición adicional, también existirá un limite superior, que puede encontrarse de la siguiente manera:

Dividamos la cuerda de longitud N en subcuerdas de longitud n; así dividido, un programa consistiría de N/n proposiciones de longitud n. Ahora, si la cuerda no contiene dos (o más) subcuerdas idénticas de longitud n, también existirá un limite superior para N (La condición de que no haya duplicaciones de

subcuerdas de longitud n es razonable, dado que la pràctica conduce a evitar que un programa se repitan expresiones).

El nûmero de combinaciones posibles de n objetos es n^n; consecuentemente, un programa consistirìa cuando màs de n^n subcuerdas de longitud n, dando asì la primera aproximación del lìmite superior:

$$N <= N^{n+1} \qquad (2.5)$$

Este lìmite puede ser mejorado si observamos que el vocabulario n consiste de n1 operadores y n2 operandos, y que su orden de aparición en una cuerda tiende a ser alternado. Por ejemplo para un vocabulario de n=4 con 2 operadores (A y B) y 2 operandos (a y b) tendrìamos las siguientes combinaciones posibles:

AaAa	AaAb	AbAa	AbAb
AaBa	AaBb	AbBa	AbBb
BaAa	BaBb	BbAa	BbAb
BaBa	BaBb	BbBa	BbBb

y en consecuencia la desigualdad 2.5 para el lìmite superior es reemplazada con:

$$N <= (n)\binom{n1}{n1}\binom{n2}{n2} \qquad (2.6)$$

Ahora debemos considerar todos los 2^N posibles subconjuntos que se pueden obtener a partir del conjunto ordenado de N elementos. De esta manera podremos igualar el nûmero de combinaciones de operadores y operandos con el nûmero de subconjuntos posibles y obtener la ecuación de longitud para la implementación de un algoritmo en tèrminos de su vocabulario. Tendrìamos entonces que:

$$2^N = \binom{n1}{n1}\binom{n2}{n2}$$

despejando N

$$N = \log_2 \left(\binom{n1}{n1}\binom{n2}{n2} \right)$$

que es también

$$N = \log_2 n1^{n1} + \log_2 n2^{n2}$$

dando

$$N^{\sim} = n1 \log_2 n1 + n2 \log_2 n2 \qquad (2.7)$$

Relación aplicable a la implementación no redundante y bien organizada de un algoritmo; el tilde "~" se emplea para diferenciar la longitud obtenida por observación directa de la obtenida por cálculo y que es una aproximación a la real.

Si aplicamos la ecuación 2.7 al algoritmo MDC expuesto anteriormente, para el que observamos:

$$n1 = 10, \quad n2 = 5 \quad y \quad N = N1 + N2 = 24 + 20 = 44$$

tendremos

$$N^{\sim} = 10 \log_2 10 + 5 \log_2 5$$
$$= 44.8$$

Evidentemente el ejemplo fue escrito de manera que $N^{\sim}$ fuese aproximado a N, ya que el algoritmo puede escribirse en un número infinito de formas para las que esta relación no se cumple. Por ejemplo, en la proposición de asignación B := R, el valor de B (y la validez del algoritmo) no se verían afectados si multiplicamos y dividimos por R cien veces alternativamente: esto incrementaría N en 400 sin cambiar n.

La validez de la expresión para obtener $N^{\sim}$ ha sido demostrada por experimentos llevados a cabo tanto con muestras pequeñas como grandes, esto es, obteniendo la longitud de implementación por observación y por la fórmula 2.7 en conjuntos de 12 a más de 400 programas, obteniendo coeficientes de correlación entre ambas longitudes superior a 0.95.

Halstead comenta la concordancia de la fòrmula para estimar la longitud de implementaciòn de un algoritmo, a pesar del número infinito de formas en la este puede ser escrito y sugiere que "el cerebro humano obedece un conjunto de reglas màs rigido de lo que suponemos, y los parámetros n1, n2, N1 y N2 pueden servir como elementos ùtiles para la obtenciòn de relaciones adicionales".

Por otra parte se ha demostrado que la fòrmula 2.7 tambièn es aplicable a las partes o mòdulos en los que se particiona lògicamente un programa; la demostraciòn consistiò en evaluar la longitud de un compilador, y evaluar la longitud de cada uno de los 14 mòdulos que lo conformaban. Para el primer caso se obtuvieròn los siguientes valores: n1 = 83, n2 = 96, N^ = 1161, N = 1334, lo que dà una variaciòn entre la longitud observada y la calculada de (N - N^) / N = Ø.1296 = 13 %.

Cuando se evaluò el compilador como 14 mòdulos separados, se obtuvo (Prom(x) denota el valor promedio de x):

N = 1334

Prom(n1) = 176/14 = 12.6

Prom(n2) = 193/14 = 13.8

$$N^{\wedge} = 14\ (Prom(n1) \log_2 Prom(n1) + Prom(n2) \log_2 Prom(n2))$$

$$= 14 \times 98.3 = 1376$$

(N - N^) / N = Ø.Ø31 = 3 %.

En consecuencia, es razonable asumir que la ecuaciòn de longitud es aplicable tanto a las partes de un programa particionado lògicamente, como al programa como un todo.

En un estudio presentado por Bell y Sullivan en el que analizan un conjunto de algoritmos publicados en "Communications of the Association for Computing Machinery", encontraron que la longitud

puede ser un excelente predictor de la ausencia de errores en un programa. Comprobaron que una medida de N menor o igual a 260 permite asumir que el programa puede estar libre de errores, o en otras palabras, que para valores mayores a 260 existe una considerable probabilidad de error en un programa.

Observando nuevamente la ecuación de longitud y recordando que la suma de dos tèrminos logarítmicos equivale a su multiplicación, podemos interpretar la longitud como una cantidad bi-dimensional. En consecuencia podemos esperar que N se comporte tambièn como una medida de àrea.

Longitud de un programa según Jensen

En un trabajo presentado por Howard Jensen (Ref. 2.1) se propone la siguiente ecuación para la estimación de la longitud de un programa:

$$N = \log_2 n1! + \log_2 n2!$$

la cual proporciona mejores resultados que la propuesta por Halstead y por tanto también la consideraremos en este trabajo.

Volumen de un programa

Otra característica importante de un algoritmo es su tamaño: dado que en algunos lenguajes la implementación de un algoritmo es màs corta que en otras, debemos contar con un parámetro que permita observar el tamaño de cada implementación; además, este indicador debe ser aplicable, sea cual fuere el lenguaje empleado, con objeto de conservar la generalidad (y objetividad) y en consecuencia debe ser independiente del conjunto de caracteres empleado para expresar un algoritmo, por lo que la medida de tamaño no debe reflejar el número de caracteres

empleado para representar identificadores de operandos u operadores. Esta condición se cumple si observamos que, para cualquier caso existe una longitud absoluta para la representación del identificador del operador u operando, si lo especificamos en bits. Además esta longitud depende sólo del número de elementos del vocabulario, esto es, n. Por ejemplo, en un vocabulario de ocho elementos diferentes podremos representar cualquier elemento del mismo con solo $\log_2 8 = 3$ bits. Así tenemos que $\log_2 n$ es la longitud mínima en bits de todos los elementos contenidos en un programa.

De esta forma, se define al volumen V de un programa como el tamaño de la implementación de un algoritmo con la siguiente expresión:

$$V = N \log_2 n \quad [\text{bits}] \qquad (2.8)$$

donde N es la longitud de implementación del algoritmo, n el vocabulario empleado y la cantidad resultante estará en bits.

Volumen potencial

La forma mas simple en la que un algoritmo pudiera ser expresado requeriría la existencia de un lenguaje en el que la operación o función requerida ya estuviera definida como una subrutina; en tal lenguaje, la implementación de ese algoritmo únicamente requeriría la especificación de los operandos, esto es, los parámetros de la subrutina.

Denotando con un asterisco a los parámetros de la forma más simple posible de un algoritmo, definimos a partir de la ecuación 2.8 la expresión para el volumen mínimo o volumen potencial V^* como:

$$V^* = (N1^* + N2^*) \log_2 (n1^* + n2^*) \qquad (2.9)$$

En esta forma minima, ningun operador u operando se repetirla, por lo que:

$$N1^* = n1^*$$

y

$$N2^* = n2^*$$

dando

$$V^* = (n1^* + n2^*) \log_2 (n1^* + n2^*) \qquad (2.1\emptyset)$$

Además, el número de operadores $n1^*$ para cualquier algoritmo consiste de: a) el operador para identificar la subrutina o función y b) el operador de asignación o de agrupamiento, por lo que:

$$n1^* = 2$$

lo que convierte a la ecuación 2.1Ø en:

$$V^* = (2 + n2^*) \log_2 (2 + n2^*) \qquad (2.11)$$

donde $n2^*$ representaría el número de parámetros de la subrutina y la cantidad resultante V^* estaría dada en bits. Una propiedad interesante de esta última ecuación es que el volumen potencial de un algoritmo es independiente del lenguaje en el que sea implementado.

Como ejemplo, para el programa que encuentra el màximo divisor común expuesto anteriormente, tendríamos que su volumen es:

$$V = (N1 + N2) \log_2 (n1 + n2)$$

$$= (24 + 2\emptyset) \log_2 (1\emptyset + 5)$$

$$= 171.9 \text{ bits}$$

Para determinar su volumen potencial, consideraremos que los parámetros de entrada y salida son A , B y R por lo que $n2^* = 3$ y entonces:

$$V^* = (2 + n2^*) \log_2 (2 + n2^*)$$

$$= 5 \log_2 5$$

$$= 11.6 \text{ bits}$$

Cuando veamos la definición de esfuerzo de programación, encontraremos una interpretación más profunda y significativa del volumen de un programa.

Nivel de un programa

Intuitivamente, el concepto del nivel en el que un programa podría estar escrito ha sido empleado desde que se utilizó por vez primera la frase "lenguajes de alto nivel".

El concepto de nivel de un programa se refiere al indicador que denota el nivel de implementación de un algoritmo, dado que este puede ser escrito en un número infinito de maneras, la habilidad de evaluar tales implementaciones nos permite preveer la claridad o propensión a errores que estas puedan tener.

La definición de nivel de un programa se obtiene a partir de las ecuaciones para volumen y volumen potencial de un programa:

$$L = V^* / V \qquad (2.12)$$

De donde se deduce que sólo la expresión más simple posible de un algoritmo puede tener un nivel igual a 1; implementaciones más voluminosas tendrían un valor de nivel menor, por lo que:

$$\emptyset < L <= 1$$

Otra forma de obtener el nivel de un programa sin recurrir al volumen potencial V^* es considerando los efectos por separado que operadores y operandos tienen sobre el nivel de un programa.

Con respecto a los primeros, es razonable asumir que a mayor nùmero de operadores distintos empleado en la implementaciòn, màs bajo serà el nivel de esa implementaciòn. En el otro extremo, el mìnimo nùmero posible de operadores empleados es 2, que serían el identificador de la funciòn y un operador de asignaciòn o de agrupamiento, por lo que $n1^* = 2$. A partir de esto, consideramos que L es proporcional a $n1^* / n1$.

Por otra parte, para los operandos podemos asumir que a mayor repeticiòn de un operando en la implementaciòn, menor serà el nivel de la misma. Esto puede ser expresado mediante la razòn n2/N2, y L serà en consecuencia proporcional a dicha razòn. Combinando estas dos ùltimas expresiones, y haciendo que la constante de proporcionalidad sea igual a uno, obtenemos la ecuaciòn de nivel

$$L^{\sim} = (n1^* / n1)\ (n2 / N2) \qquad (2.13)$$

donde se ha puesto el tilde "~" para denotar que es una aproximaciòn a la ecuaciòn 2.12 y que por evidencia experimental permite emplearla como una definiciòn alterna y vàlida para el nivel de un programa.

Cuantificaciòn del contenido intelectual

El volumen y el nivel de un programa tienen implicaciones que pueden ser sumamente interesantes; se ha notado que el producto de ambos parámetros para cualquier algoritmo tiende a ser invariante entre las implementaciones en lenguajes distintos de

ese algoritmo, por lo que esto debe representar una propiedad fundamental del mismo; podrìa ser un indicador de "cuanto" se està diciendo en una implementaciòn determinada. Esta medida de "que tanto" està diciendo un programa se le denomina contenido intelectual de un programa y se define como:

$$I = L^{\wedge} V \qquad (2.14)$$

que a partir de las ecuaciones 2.8 y 2.13 es:

$$I = (2 / n1)(n2 / N2)\ (N1 + N2)\ \log_2 (n1 + n2) \qquad (2.15)$$

donde todos los tèrminos necesarios para obtener I son directamente medibles en la implementaciòn de cualquier algoritmo. Este estimador tambièn està directamente relacionado con el volumen potencial V^* y como este es independiente del lenguaje, el contenido intelectual I tambièn debe ser independiente (y permanecer casi invariante entre las implementaciones de un algoritmo).

Esto proporciona una medida ùtil del contenido inherente de un programa; aunque por definiciòn el contenido intelectual y el volumen potencial representan cantidades diferentes, estàn fuertemente acopladas y pueden ser empleadas indistintamente.

Esfuerzo de programaciòn

Si consideramos al esfuerzo de programaciòn como la actividad mental requerida por una persona para expresar un algoritmo en un lenguaje de programaciòn, entonces los indicadores y conceptos que se han expresado anteriormente pueden proporcionar una estimaciòn del esfuerzo requerido en una implementaciòn y que serìa un paràmetro ùtil para la evaluaciòn y el control en el proceso de desarrollo de software.

Como puede comprenderse, la deducción de una ecuación para la obtención de este parámetro debe considerar aspectos psicológicos. No obstante, el intentar describir por medio de simples ecuaciones la estructura de los procesos mentales de un programador promedio sería ilusorio, por decir lo menos; más bien se hacen suposiciones que en un principio pueden parecer razonables o extrañas, pero que los resultados de experimentación posterior han validado. La programación es una actividad altamente intelectual como para no considerar aspectos psicológicos en su estudio.

Para la obtención de la ecuación de esfuerzo, primero consideramos que en la implementación de un algoritmo se harán N selecciones de un vocabulario de n elementos. Además, esta selección no se hace de forma aleatoria; con la posible excepción de las técnicas de dispersión (hash), la forma más eficiente de búsqueda en una lista ordenada es la búsqueda binaria, que requiere de $\log_2 n$ comparaciones para la selección de un elemento del vocabulario. De esta manera, la implementación de un algoritmo requerirá de $N \log_2 n$ comparaciones mentales, expresión que ya hemos considerado previamente como el volumen de un programa (ecuación 2.8); en consecuencia, podemos considerarlo como una estimación del número de comparaciones mentales requeridas para escribir un programa.

Por otra parte, cada comparación requiere una cierta cantidad de discriminaciones elementales, y esto es una buena medida de la dificultad de la implementación. A partir de los conceptos expuestos en la definición del nivel de un programa, podemos

considerarlo como el reciproco de la dificultad del programa. Por tanto, si el volumen V es una estimación de la comparaciones mentales, y el inverso del nivel es una medida del número promedio de discriminaciones requeridas por cada comparación, entonces el número total de discriminaciones elementales E estarà dado por:

$$E = V / L \qquad (2.16)$$

Una implicación interesante de esta ecuación es que si sustituimos la ecuación 2.12 en la 2.16 obtenemos:

$$E = V^2 / V^* \qquad (2.17)$$

Indicando que el esfuerzo mental requerido para implementar un algoritmo con un volumen potencial dado variarà con el cuadrado de su volumen. Además, dado que el cuadrado de una suma es mayor que la suma de los cuadrados, entonces una modularización adecuada reducirà el esfuerzo de programación: evidentemente al particionar un sistema en programas pequeños, además de hacerlos màs manejables tambien podemos verificarlos màs facilmente y obtener al mismo tiempo una mayor confiabilidad. (E. W. Dijkstra comenta en este sentido:

"Un programa o un sistema de software grande està finalmente compuesto de n programas pequeños o módulos. Supongamos que cada uno de estos n módulos independientes tiene una probabilidad p de ser correcto. Entonces la probabilidad P de que el sistema por entero sea correcto, con seguridad satisface $P <= p^n$. Dado que n es grande, con objeto de tener alguna confianza en la confiabilidad del sistema, p debe ser muy cercano a 1")

Debemos recalcar que el mètodo empleado para la estimaciòn del esfuerzo requerido puede ser muy simplista; la tarea de comprender y elaborar un programa no puede ser reducida a la bûsqueda de operadores en una lista ordenada. Sin embargo, el intento de combinar la ciencia de la computaciòn con la psicologia para intentar explicar ciertos fenòmenos involucrados en la producciòn de software es un enfoque novedoso que tal vez valiese la pena investigar màs a fondo.

CONSIDERACIONES SOBRE COHESION Y ACOPLAMIENTO

Si bien las caracterìsticas de acoplamiento (que es el grado de interconexiòn entre mòdulos en un sistema) y cohesiòn (que es una medida de la fuerza funcional de un mòdulo) inciden finalmente en la facilidad de mantenimiento y la confiabilidad de un sistema, estos son atributos fundamentalmente cualitativos, es decir, no se puede obtener en forma automàtica una cantidad que nos diga hasta que punto un mòdulo es cohesivo, o que grado de acoplamiento existe entre sus mòdulos.

No obstante, debemos tomar en cuenta que los beneficios de seguir los criterios de implementar un sistema con cohesiòn alta en sus mòdulos y con un acoplamiento mìnimo entre los mismos pueden obtenerse a travès de varias metodologìas de diseño propuestas y mejoradas por la investigaciòn y experiencia de varios autores. Con respecto a la metodologìa de diseño modular que seguimos en este trabajo es pertinente mencionar que los factores clave que contribuyen su aplicaciòn exitosa son la independencia de mòdulos y abstracciòn de datos. El primero implica que debe ser posible modificar los detalles de la implementaciòn de cualquier mòdulo sin dañar el resto del sistema. El segundo conlleva que los subprogramas cliente que importan tipos de datos u objetos de un mòdulo determinado no deben tener acceso a los detalles de la representaciòn de tales objetos, esto es, ùnicamente se pueden accesar y manipular esos objetos por medio del conjunto de operaciones especificadas para tales objetos. Un programa bien escrito debe ser tan independiente de las decisiones de implementaciòn como sea posible, esto es, debe de concentrarse en

los detalles del algoritmo. Esta tècnica propuesta desde hace varios años por David Parnas y denominada "ocultamiento de información" (information hiding) puede ser fàcilmente adoptada al utilizar el lenguaje Modula-2. En el siguiente capitulo emplearemos esta tècnica para el desarrollo de una herramienta de software para la obtención de los paràmetros de Halstead para programas escritos en Modula-2.

CAPITULO 3:

DESARROLLO DE UNA HERRAMIENTA DE SOFTWARE PARA LA OBTENCION DE LOS PARAMETROS DE HALSTEAD PARA PROGRAMAS ESCRITOS EN MODULA-2

En este capítulo se presenta el diseño e implementación de una herramienta de software para la medición de programas escritos en Modula-2 de acuerdo con la teoría propuesta por Maurice Halstead expuesta en el capítulo anterior y para lo cual empleamos una nueva metodología de diseño que aprovecha las características y las facilidades para la construcción de software con este lenguaje: el diseño modular orientado a objetos.

Con esta herramienta podremos evaluar y corroborar los resultados presentados por Halstead para confirmar la validez de su teoría en un lenguaje de programación moderno y al mismo tiempo contar con una herramienta que nos permita medir programas realizados en Modula-2 con las ventajas que se mencionaron en el capítulo 1.

De esta manera, además de presentar un sistema de medición poco conocido en nuestro medio, lo adecuamos para un lenguaje de programación que consideramos tendrá una amplia aceptación por sus características y que lo consideramos como uno de los mejores exponentes en la familia de lenguajes imperativos.

Por otra parte, el diseño de software conlleva un proceso de abstracción. Los objetos y las operaciones encontradas en la vida real deben ser transladadas a las correspondientes objetos y operaciones empleados en la solución de problemas mediante un sistema de software, esto con mayor o menor facilidad dependiendo de las características del lenguaje que se emplée y por supuesto de la experiencia y criterio que el diseñador posea.

Con el diseño modular orientado a objetos ya no es necesario que durante el diseño se tenga que mapear directamente los objetos reales en estructuras de datos y de control predefinidos en algún lenguaje de programación; en vez de ello el diseñador puede crear sus propios tipos y abstracciones funcionales que reflejan de una manera más natural el problema y hacen el mapeo a estas abstracciones mucho más sencillo, dado que puede ser creado un número virtualmente ilimitado de tipos abstractos. Además, el diseño del sistema se desacopla de los detalles de la representación interna de los objetos y que pueden ser modificados cuantas veces sea necesario sin inducir fallas "en cascada" al sistema. Con este método, y gracias a las facilidades de Modula-2, desarrollaremos la herramienta de software ya mencionada.

Requerimientos y especificaciones

Necesitamos una herramienta que nos permita obtener los indicadores de Halstead para programas escritos en Modula-2. La entrada de esta será un programa fuente residente en disco, sintácticamente correcto (no tiene sentido obtener indicadores de un programa con errores de sintaxis). Específicamente se deberán reportar los estimadores de Halstead de módulos de implementación y módulos de programa, dado que en los módulos de definición no existe la implementación de ningún algoritmo.

Requerimientos funcionales

Los requerimientos funcionales se refieren a las operaciones y transformaciones que el sistema debe llevar a cabo, así como los detalles concernientes a la interacción del usuario con el sistema.

1. El sistema requerirá del usuario el nombre del archivo en disco que contenga el programa en Modula-2, sintácticamente correcto que se desée analizar y deberá reportar la cuantificación de operadores y operandos para cada procedimiento y módulo del programa, así como las medidas de longitud, volumen, nivel (y su inverso, dificultad), contenido intelectual, esfuerzo de programación y longitud según la fórmula propuesta por Howard Jensen. La presentación de estos resultados deberá considerar el nivel de anidamiento del módulo o procedimiento correspondiente con objeto de mejorar la legibilidad de los resultados.

2. Para que el sistema pueda distinguir entre operadores y operandos en una lista de importación calificada, se le pedirá información al usuario. Por ejemplo, si el programa a analizar aparece la declaración

FROM INOUT IMPORT WriteString;

el programa preguntará al usuario si WriteString es un procedimiento para considerarlo como un operador. Con el mismo fin, cuando un módulo local exporte un objeto se le hará al usuario la misma pregunta.

3. Se deberá reconocer la declaración de procedimientos FORWARD, que si bien no existe en la definición propuesta por Wirth, si está presente en la implementación de Volition Systems.

4. Para efectos de cuantificación se reconocerán como operadores: los procedimientos importados de módulos de biblioteca, los procedimientos declarados dentro del módulo o procedimiento que se analize, los procedimientos exportados por módulos locales, los símbolos estándar reservados (a excepción de "!" que reemplaza a "↑" en virtud de que no puede ser generado en la mayoría de los teclados de las terminales) y las siguientes agrupaciones de símbolos y palabras reservadas que se contarán como un sólo operador:

```
!    #    &    *    +    ,    -    .    ..   []
:    :=   <    <=   <>   =    >    >=   ()
AND  BY   DIV  EXIT IN   MOD  NOT  OR
CASE ... END
FOR ... END
```

IF ... THEN ... END

ELSE

ELSIF

LOOP ... END

REPEAT ... UNTIL

WHILE ... END

WITH ... END

ABS	BITSET	BOOLEAN	CAP	CARDINAL	CHAR
CHR	DEC	DISPOSE	EXCL	FLOAT	HALT
HIGH	INC	INCL	INTEGER	NEW	ODD
ORD	REAL	TRUNC	VAL		

Requerimientos no funcionales

Los requerimientos no funcionales se refieren al tipo y características del computador en el que correrá el sistema, la compatibilidad que el sistema a implementar deba tener con otros sistemas existentes y en general las restricciones impuestas por hardware o software.

1. El sistema se ejecutará en un computador Apple II con 64 Kbytes de memoria principal, 2 unidades de disco flexible de 140 Kbytes cada una y operando bajo el sistema UCSD Pascal.

2. La nomenclatura empleada para especificar los archivos de entrada y salida debe ser la misma establecida en el sistema operativo mencionado.

3. El sistema deberá ser capaz de medir programas que contengan hasta 32 declaraciones de procedimientos o módulos locales.

4. El sistema deberà ocupar una sòla unidad de disco flexible.

5. Los mètodos de bùsqueda y cuantificaciòn de operadores y operandos deben ser eficientes y emplear un mìnimo de memoria.

Diseño

Dos factores clave que contribuyen al èxito del diseño modular son: independencia de mòdulos y abstracciòn de datos. La independencia de mòdulos significa que debe ser posible modificar los detalles de la implementaciòn de cualquier mòdulo sin 'dañar el resto del sistema. La abstracciòn de datos implica que los programas clientes que importan tipos de datos u objetos de un mòdulo determinado no tienen acceso a los detalles de representaciòn de los datos. Sòlo el conjunto de operaciones especificadas sobre los objetos de datos pueden ser empleados para tener acceso a tales datos.

Los elementos fundamentales del diseño modular son:

* Tipos abstractos de datos, con sus detalles de representaciòn ocultos.

* Subprogramas, donde cada uno satisface una especificaciòn funcional (lo que ademàs favorece una alta cohesiòn).

* Mòdulos, donde cada uno representa una agrupaciòn de subprogramas funcionalmente relacionados y que operan sobre tipos abstractos de datos (lo que ademàs permite un bajo acoplamiento entre mòdulos).

El sistema de software es entonces una interconexiòn de mòdulos controlados por un programa principal.

Comenzemos pues por bosquejar un algoritmo que cúmpla con las especificaciones y principios antes señalados para que podamos reconocer los "objetos" que necesitaremos.

Algoritmo para la obtención de los parámetros de Halstead para programas escritos en Modula-2

Obtener-archivo que contiene el programa a medir.
Definir-archivo de salida.
Mientras **obtenga-token** del archivo de entrada:
  Si el token es palabra o símbolo reservado entonces:
    Caso de que el token sea:
      MODULE,
      PROCEDURE: **Inicia-conteo** para un nuevo módulo o procedimiento, verificando la presencia de declaraciones IMPORT, EXPORT o FORWARD, o de otro módulo o procedimiento.

      BEGIN: nivel de anidamiento := 1; **Incrementa-N1**; **Verifica-n1** (esto es, verifica que BEGIN no se haya empleado antes para así poder incrementar n2 -número de operadores distintos-).

      CASE, FOR, IF,
      LOOP, WITH, WHILE: **Incrementa** nivel de anidamiento; **Incrementa-N1**; **Verifica-n1**.

      END: Si nivel de anidamiento > 0 entonces: **Decrementa** nivel de anidamiento; Si nivel de anidamiento=0 entonces: **Termina-conteo** del módulo o procedimiento (lo que conlleva emitir los resultados del conteo).

      ! ([&
      * + , -
      . .. : :=
      ; < <= >
      >= = <> #
      AND, BY, DIV,
      ELSE, ELSIF,
      EXIT, IN, MOD,
      NOT, OR, REPEAT,
      ABS, BITSET,
      BOOLEAN, CAP,
      CARDINAL, CHAR,
      CHR, DEC, DISPOSE,
      EXCL, FLOAT, HALT,
      HIGH, INCL, INC,
      INTEGER, NEW, ODD,

```
        ORD, REAL, TRUNC, VAL: Si nivel de anidamiento > Ø
                        entonces: Incrementa-N1; Verifica-n1.

        FALSE,
        TRUE,
        NIL: Si nivel de anidamiento > Ø entonces:
                Incrementa-N2; Verifica-n2.

    (* si no fue palabra reservada, entonces: *)
    Si (nivel de anidamiento > Ø) Y (el token es un
      Procedimiento-visible) entonces: Incrementa-N1;
      Verfica-n1.
    (* si no, entonces *)
    Si (nivel > Ø) Y (el token no es palabra reservada)
    entonces: (* es un operando *) Incrementa-N2; Verifica-n2.

  (* cuando no haya más tokens *)
  Libera-archivo de entrada
  Cierra-archivo de salida.
```

Al estudiar el algoritmo podemos observar que es necesario obtener los elementos básicos del lenguaje o "tokens", identificarlos como operadores u operandos y llevar apropiadamente la cuantificación de los mismos para el módulo o procedimiento que se esté analizando.

A partir de esto, podemos considerar que se requieren dos grupos de procedimientos: a) el primero que proporcione los procedimientos para obtener del usuario el nombre del archivo que contiene el programa y que obtenga los tokens de este; esta es la función característica de un analizador lexicográfico o "scanner". En consecuencia, el módulo Scanner exportará los procedimientos IniciaScan, Token y TerminaScan, así como el tipo enumerativo Clase que permitirá identificar el tipo de token que se obtiene (palabra no-reservada, símbolo reservado, etc.); y b) el segundo grupo proporciona los procedimientos para llevar la cuenta de operadores y operandos, a saber: IniciaConteo (que inicia la cuenta de operadores y operandos de un módulo o

procedimiento), IncN1 (que incrementa a N1 -número total de operadores empleados- en 1), VERIFICAn1 (que verifica que el operador en cuestión no haya sido empleado previamente dentro del módulo o procedimiento para así poder incrementar n1 -número de operadores distintos- en 1), IncN2 (que incrementa a N2 -número total de operandos empleados- en 1), VERIFICAn2 (para verificar que un operando no se haya utilizado antes para así poder incrementar n2 -número de operandos distintos- en 1), ImportDecl (para manejar los objetos importados por un módulo de manera no calificada), ExportDecl (para el manejo de objetos exportados por módulos locales), ForwardDecl (para el control de procedimientos declarados FORWARD), ProgramOp (que nos dice si un token es un procedimiento visible) y TerminaConteo (que reporta el conteo de operadores, operandos y las mediciones del módulo o procedimiento correspondiente). Estos procedimientos serán exportados por el módulo Conteo.

Estos recursos de software proporcionados por Scanner y Conteo se ponen a disposición del módulo Halstead (el cual contendrá el algoritmo definido informalmente líneas antes) a través de un canal de recursos o "software bus".

La figura 3.1 muestra esta organización en términos de Modula-2. Los 2 bloques superiores (DEF Scanner y DEF Conteo) representan los módulos de definición empleados para definir los objetos y la interface entre estos módulos y el programa cliente -en este caso, Halstead- (le llamamos interface ya que a través de ella se comunican dos o más componentes de un sistema); la habilidad para definir formalmente los objetos independientemente de los detalles de su implementación permite establecer las

características de cada objeto y visualizar los subcomponentes principales de un sistema durante la fase de diseño (esto es, cuando la implementación de los objetos aún no se ha llevado a cabo). Al compilar estos módulos de definición, el mismo compilador asegurará la consistencia en el uso de la interface (consulte el apéndice para mayor detalle).

Además con este esquema podemos modificar los detalles de la implementación cuantas veces sea necesario sin necesidad de recompilar los módulos clientes. Por otra parte, la división del trabajo de implementación entre dos o más personas se simplifica.

Una vez identificados los componentes principales del sistema podemos definir con mayor detalle los elementos que los conforman.

Después de revisar el diseño inicial, podemos hacer una revisión que dé como resultado un diseño "mejorado"; recordemos que la obtención de este es gradual, no instantánea (figura 3.2).

En primer lugar, puede definirse otro módulo que contenga las fórmulas para el cálculo de los indicadores de Halstead, ya que su función no pertenece precisamente a Scanner o a Conteo; por otra parte, se obtiene una mayor cohesión y es más fácil el mantenimiento al sistema si las fórmulas se agrupan en un módulo separado. De esta manera, el módulo Formulae exporta LongHalstead (que calcula la longitud de un programa según Halstead), LongJensen (que calcula la longitud según la fórmula de Jensen), Volumen, Nivel, Contenido intelectual y Esfuerzo de programación. Por lo que respecta a las funciones del módulo Conteo, para llevar a cabo el control de cada módulo y procedimiento sería

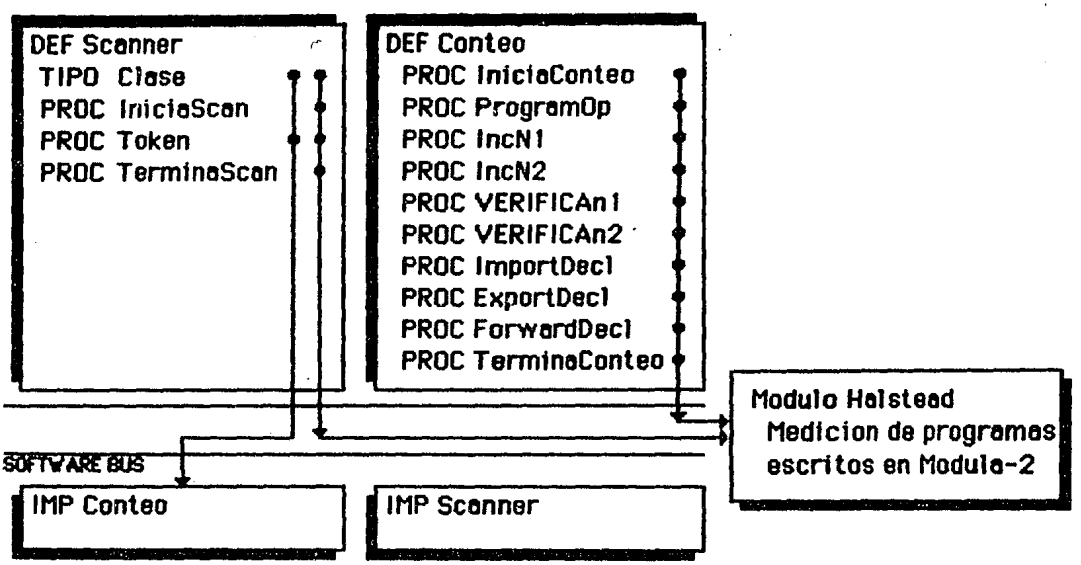


Fig. 3.1 Diseño inicial

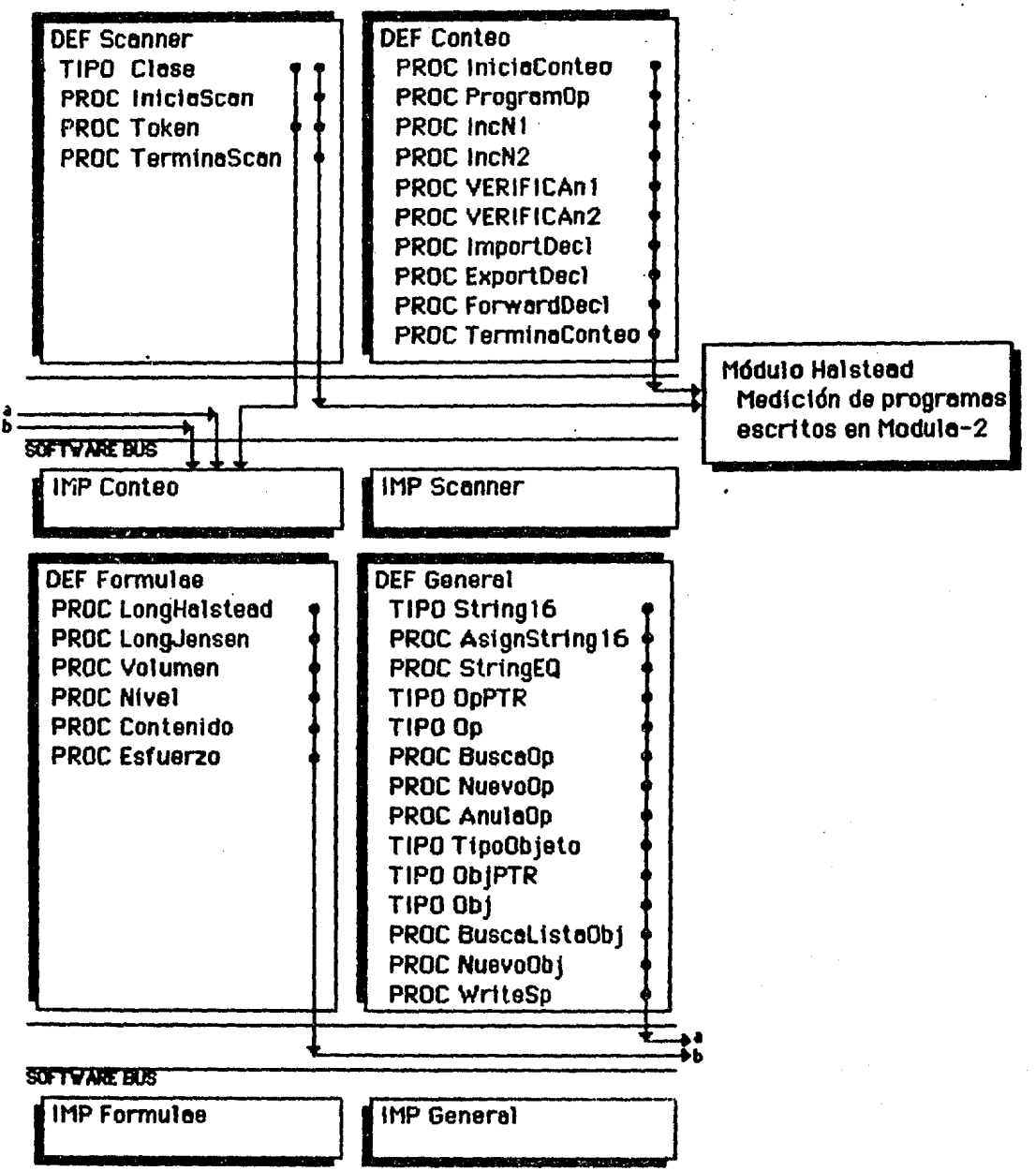


Fig. 3.2 Diseño final

conveniente tener un descriptor en el que se registraran las caracteristicas de cada módulo o procedimiento del programa que se analize. En este descriptor tendríamos el nombre del módulo o procedimiento, el nivel en el que ha sido declarado (para poder determinar su alcance), la lista de objetos que importa o exporta (si es un módulo), la lista de operadores que haya definido (esto es, los procedimientos que haya declarado dentro de él) y por supuesto su cuenta particular de operadores y operandos. Si las listas mencionadas se implementan como listas ligadas podremos utilizar exactamente la cantidad mínima de memoria (dado que no podemos saber con anticipación cuántos objetos puede importar, exportar o definir). Al tener un arreglo de n descriptores estaremos en condiciones de analizar programas que contengan hasta n módulos o procedimientos.

El módulo General se encarga de simplificar el desarrollo de Conteo, ya que proporciona los tipos de datos y los procedimientos para el manejo de las listas antes mencionadas. El tipo Op es un nodo para la lista ligada de operadores dependiente de un procedimiento o módulo y OpPTR es un apuntador a Op; los procedimientos BuscaOp (búsqueda de un operador en una lista ligada), NuevoOp (inserción de un nuevo nodo en la lista) y AnulaOp (liberación al sistema de la memoria ocupada por una lista) completan las funciones requeridas para el manejo de listas de operadores. Los tipos TipoObjeto y Obj sirven para la clasificación (en operadores u operandos) de los objetos exportados por un módulo local y ObjPTR es un apuntador a un nodo de tipo Obj; con el procedimiento NuevoObj insertamos un nodo de

tipo Obj en la lista de objetos exportados y con BuscaListaObj podemos determinar si un objeto determinado es exportado por un módulo local. Además este módulo provée rutinas de propósito general, como WriteSp que escribe un número determinado de espacios en blanco y el tipo String16 junto con los procedimientos AsignString16 y StringEQ para la asignación y comparación de cuerdas de hasta 16 caracteres, empleadas para la identificación de los procedimientos y variables.

Con este esquema mantenemos la facilidad de mantenimiento, podemos probar distintos métodos para la implementación y acceso de las listas ligadas sin recompilar innecesariamente las rutinas de conteo y seguimos manteniendo una baja cohesión entre módulos.

Implementación

A continuación se muestran los módulos de definición e implementación que corresponden a los bloques mostrados en la figura 3.2. Consideramos que con este método de diseño el problema se mantiene dentro de límites razonables de complejidad y se permite darle mantenimiento sin afectar otras partes del sistema, para adecuarse con facilidad a necesidades futuras.

```
DEFINITION MODULE Scanner;              (* $SEG := 29 *)

  (* Rutinas basicas para el analisis lexicografico
     de programas escritos en Modula-2.              *)

  EXPORT QUALIFIED  Clase,
                    IniciaScan,
                    Token,
                    TerminaScan;

  TYPE Clase = (VERTICAL,COMILLA,APOSTROFE,GATO,PARIZQ,PARDER,PARENCIZQ,
                PARENCDER,INICOM,FINCOM,AMPER,ASTER,MAS,COMA,MENOS,PUN,PUNPUN,
                DOSPUN,ASIGNA,PUNCOMA,MENOR,MENIGUAL,DISTINTO,IGUAL,MAYOR,
                MAYIGUAL,ANDSIM,BEGINSIM,BYSIM,CASESIM,DIVSIM,DOSIM,ELSESIM,
                ELSIFSIM,ENDSIM,EXITSIM,EXPORTSIM,FORSIM,FORWARDSIM,FROMSIM,
                IFSIM,IMPORTSIM,INSIM,MODSIM,LOOPSIM,MODULESIM,NOTSIM,OFSIM,
                ORSIM,PROCEDURESIM,QUALIFISIM,REPEATSIM,RETURNSIM,THENSIM,
                TOSIM,UNTILSIM,WHILESIM,WITHSIM,DECLARADOR,NORESERVADO,ABSST,
                BITSETST,BOOLEANST,CAPST,CARDINALST,CHARST,CHRST,DECST,DISPOSEST,
                EXCLST,FALSEST,FLOATST,HALTST,HIGHST,INCST,INCLST,INTEGERST,
                NEWST,NILST,ODDST,ORDST,PROCST,REALST,TRUEST,TRUNCST,VALST);

  PROCEDURE IniciaScan;
    (* Pide al usuario el nombre del archivo que contiene el programa
       del que se obtendran sus elementos basicos (tokens).        *)

  PROCEDURE Token ( VAR T: ARRAY OF CHAR; VAR C: Clase ): BOOLEAN;
    (* Regresa el valor booleano TRUE cuando pudo obtener del archivo
       que se analiza un token T de clase C y FALSE en caso contrario *)

  PROCEDURE TerminaScan;
    (* Cierra el archivo que fue analizado. Debe emplearse siempre al
       final del programa o antes de analizar otro archivo.        *)

END Scanner.
```

```
                (* $STANDARD := FALSE; $SPECIAL := TRUE *)

IMPLEMENTATION MODULE Scanner;

  FROM  InOut   IMPORT  OpenInput, CloseInput, EOL, Done, Read, WriteString;

  FROM  Strings IMPORT  Concat, Copy;

PROCEDURE IniciaScan;
BEGIN
  WriteString (' Programa a analizar: ');
  OpenInput ('TEXT');
  IF NOT Done THEN HALT END
END IniciaScan;


TYPE  Alfanumeric = SET OF CHAR;

PROCEDURE Salta ( S: ARRAY OF CHAR; CHARSET: Alfanumeric; VAR I: CARDINAL );
VAR  LIMIT: CARDINAL;
BEGIN         (* $RANGE := FALSE *)
   LIMIT := HIGH(S);
   WHILE (S[I] IN CHARSET) & (I < LIMIT) DO INC(I) END
END Salta;    (* $RANGE := TRUE *)

PROCEDURE LeeLinea ( VAR S: ARRAY OF CHAR; VAR LONG: CARDINAL ): BOOLEAN;
VAR  C: CHAR;
     I: CARDINAL;
    . EOF: BOOLEAN;
BEGIN           (* $RANGE := FALSE *)
  I := 0; LONG := 0;
  REPEAT
    Read (C); EOF := C = 0C;
    S[LONG] := C; INC(LONG)
  UNTIL (C = EOL) OR EOF; S[LONG] := 0C;
  RETURN (NOT EOF)
END LeeLinea;    (* $RANGE := TRUE *)


TYPE  String10 = PACKED ARRAY [0..9] OF CHAR;
      Elemento = RECORD
                   ID: String10;
                   TIPO: Clase
                 END;

PROCEDURE BuscaElemento ( T: ARRAY OF CHAR; A: ARRAY OF Elemento; VAR N: CARDINAL ): BOOLEAN;
VAR
   OK: BOOLEAN;
   L,R,I: CARDINAL;
BEGIN                 (* $RANGE := FALSE *)
  L := 1; R := HIGH(A);
  REPEAT
    N := (L + R) DIV 2; I := 0;
    WITH A[N] DO
      WHILE (T[I] = ID[I]) & (T[I] # " ") DO INC(I) END;
      IF T[I] <= ID[I] THEN R := N - 1 END;
```

```
      IF T[I] >= ID[I] THEN L := M + 1 END;
      OK := (ID[I] = " ") & (T[I] = " ")
    END;
  UNTIL OK OR (L > R); RETURN OK
END BuscaElemento;    (* $RANGE := TRUE *)


CONST  MaxReservadas = 71;

VAR  PR: ARRAY [0..MaxReservadas] OF Elemento;

PROCEDURE InitReservadas;
  PROCEDURE InitID;
  BEGIN       (* $RANGE := FALSE *)
    PR[1].ID := '! ';  PR[2].ID := '" ';   PR[3].ID := '# ';   PR[4].ID := '& ';
    PR[5].ID := "' ";  PR[6].ID := '( ';   PR[7].ID := '(* ';  PR[8].ID := ') ';
    PR[9].ID := '* ';  PR[10].ID := '*) '; PR[11].ID := '+ ';  PR[12].ID := ', ';
    PR[13].ID := '- '; PR[14].ID := '. ';  PR[15].ID := '.. '; PR[16].ID := ': ';
    PR[17].ID := ':= ';PR[18].ID := '; ';  PR[19].ID := '< ';  PR[20].ID := '<= ';
    PR[21].ID := '<> ';PR[22].ID := '= ';  PR[23].ID := '> ';  PR[24].ID := '>= ';

    PR[25].ID := 'AND ';      PR[26].ID := 'ARRAY ';     PR[27].ID := 'BEGIN ';
    PR[28].ID := 'BITSET ';   PR[29].ID := 'BOOLEAN ';   PR[30].ID := 'BY ';
    PR[31].ID := 'CASE ';     PR[32].ID := 'CARDINAL ';  PR[33].ID := 'CHAR ';
    PR[34].ID := 'CONST ';    PR[35].ID := 'DIV ';       PR[36].ID := 'DO ';
    PR[37].ID := 'ELSE ';     PR[38].ID := 'ELSIF ';     PR[39].ID := 'END ';
    PR[40].ID := 'EXIT ';     PR[41].ID := 'EXPORT ';    PR[42].ID := 'FOR ';
    PR[43].ID := 'FORWARD ';  PR[44].ID := 'FROM ';      PR[45].ID := 'IF ';
    PR[46].ID := 'IMPORT ';   PR[47].ID := 'IN ';        PR[48].ID := 'INTEGER ';
    PR[49].ID := 'LOOP ';     PR[50].ID := 'MOD ';       PR[51].ID := 'MODULE ';
    PR[52].ID := 'NOT ';      PR[53].ID := 'OF ';        PR[54].ID := 'OR ';
    PR[55].ID := 'PACKED ';   PR[56].ID := 'POINTER ';   PR[57].ID := 'PROCEDURE ';
    PR[58].ID := 'QUALIFIED ';PR[59].ID := 'RECORD ';    PR[60].ID := 'REPEAT ';
    PR[61].ID := 'RETURN ';   PR[62].ID := 'SET ';       PR[63].ID := 'THEN ';
    PR[64].ID := 'TO ';       PR[65].ID := 'TYPE ';      PR[66].ID := 'UNTIL ';
    PR[67].ID := 'VAR ';      PR[68].ID := 'WHILE ';     PR[69].ID := 'WITH ';
    PR[70].ID := '[ ';                                   PR[71].ID := '] ';
  END InitID;    (* $RANGE := TRUE *)

  PROCEDURE InitTipo;
  BEGIN       (* $RANGE := FALSE *)
    PR[1].TIPO := VERTICAL;  PR[2].TIPO := COMILLA;   PR[3].TIPO := GATO;
    PR[4].TIPO := AMPER;     PR[5].TIPO := APOSTROFE; PR[6].TIPO := PARIZQ;
    PR[7].TIPO := INICOM;    PR[8].TIPO := PARDER;    PR[9].TIPO := ASTER;
    PR[10].TIPO := FINCOM;   PR[11].TIPO := MAS;      PR[12].TIPO := COMA;
    PR[13].TIPO := MENOS;    PR[14].TIPO := PUN;      PR[15].TIPO := PUNPUN;
    PR[16].TIPO := DOSPUN;   PR[17].TIPO := ASIGNA;   PR[18].TIPO := PUNCOMA;
    PR[19].TIPO := MENOR;    PR[20].TIPO := MENIGUAL; PR[21].TIPO := DISTINTO;
    PR[22].TIPO := IGUAL;    PR[23].TIPO := MAYOR;    PR[24].TIPO := MAYIGUAL;

    PR[25].TIPO := ANDSIM;     PR[26].TIPO := DECLARADOR; PR[27].TIPO := BEGINSIM;
    PR[28].TIPO := DECLARADOR; PR[29].TIPO := DECLARADOR; PR[30].TIPO := BYSIM;
    PR[31].TIPO := CASESIM;    PR[32].TIPO := DECLARADOR; PR[33].TIPO := DECLARADOR;
    PR[34].TIPO := DECLARADOR; PR[35].TIPO := DIVSIM;     PR[36].TIPO := DOSIM;
    PR[37].TIPO := ELSESIM;    PR[38].TIPO := ELSIFSIM;   PR[39].TIPO := ENDSIM;
    PR[40].TIPO := EXITSIM;    PR[41].TIPO := EXPORTSIM;  PR[42].TIPO := FORSIM;
```

```
    PR[43].TIPO := FORWARDSIM; PR[44].TIPO := FROMSIM;    PR[45].TIPO := IFSIM;
    PR[46].TIPO := IMPORTSIM;  PR[47].TIPO := INSIM;      PR[48].TIPO := DECLARADOR;
    PR[49].TIPO := LOOPSIM;    PR[50].TIPO := MODSIM;     PR[51].TIPO := MODULESIM;
    PR[52].TIPO := NOTSIM;     PR[53].TIPO := OFSIM;      PR[54].TIPO := ORSIM;
    PR[55].TIPO := DECLARADOR; PR[56].TIPO := DECLARADOR; PR[57].TIPO := PROCEDURESIM;
    PR[58].TIPO := QUALIFISIM; PR[59].TIPO := DECLARADOR; PR[60].TIPO := REPEATSIM;
    PR[61].TIPO := RETURNSIM;  PR[62].TIPO := DECLARADOR; PR[63].TIPO := THENSIM;
    PR[64].TIPO := TOSIM;      PR[65].TIPO := DECLARADOR; PR[66].TIPO := UNTILSIM;
    PR[67].TIPO := DECLARADOR; PR[68].TIPO := WHILESIM;   PR[69].TIPO := WITHSIM;
    PR[70].TIPO := PARENCIZQ;  PR[71].TIPO := PARENCDER
  END InitTipo;    (* $RANGE := TRUE *)

BEGIN
  InitID; InitTipo
END InitReservadas;


CONST  MaxEstandar = 26;

VAR  ST: PACKED ARRAY [0..MaxEstandar] OF Elemento;

PROCEDURE InitEstandares;
  PROCEDURE InitID;
  BEGIN          (* $RANGE := FALSE *)
    ST[1].ID := 'ABS ';    ST[2].ID := 'BITSET ';   ST[3].ID := 'BOOLEAN ';
    ST[4].ID := 'CAP ';    ST[5].ID := 'CARDINAL '; ST[6].ID := 'CHAR ';
    ST[7].ID := 'CHR ';    ST[8].ID := 'DEC ';      ST[9].ID := 'DISPOSE ';
    ST[10].ID := 'EXCL ';  ST[11].ID := 'FALSE ';   ST[12].ID := 'FLOAT ';
    ST[13].ID := 'HALT ';  ST[14].ID := 'HIGH ';    ST[15].ID := 'INC ';
    ST[16].ID := 'INCL ';  ST[17].ID := 'INTEGER '; ST[18].ID := 'NEW ';
    ST[19].ID := 'NIL ';   ST[20].ID := 'ODD ';     ST[21].ID := 'ORD ';
    ST[22].ID := 'PROC ';  ST[23].ID := 'REAL ';    ST[24].ID := 'TRUE ';
    ST[25].ID := 'TRUNC '; ST[26].ID := 'VAL '
  END InitID;    (* $RANGE := TRUE *)

PROCEDURE InitTipo;
  BEGIN          (* $RANGE := FALSE *)
    ST[1].TIPO := ABSST;    ST[2].TIPO := BITSETST;   ST[3].TIPO := BOOLEANST;
    ST[4].TIPO := CAPST;    ST[5].TIPO := CARDINALST; ST[6].TIPO := CHARST;
    ST[7].TIPO := CHRST;    ST[8].TIPO := DECST;      ST[9].TIPO := DISPOSEST;
    ST[10].TIPO := EXCLST;  ST[11].TIPO := FALSEST;   ST[12].TIPO := FLOATST;
    ST[13].TIPO := HALTST;  ST[14].TIPO := HIGHST;    ST[15].TIPO := INCST;
    ST[16].TIPO := INCLST;  ST[17].TIPO := INTEGERST; ST[18].TIPO := NEWST;
    ST[19].TIPO := NILST;   ST[20].TIPO := ODDST;     ST[21].TIPO := ORDST;
    ST[22].TIPO := PROCST;  ST[23].TIPO := REALST;    ST[24].TIPO := TRUEST;
    ST[25].TIPO := TRUNCST; ST[26].TIPO := VALST
  END InitTipo;  (* $RANGE := TRUE *)

BEGIN
  InitID; InitTipo
END InitEstandares;
```

```
PROCEDURE ClaseToken ( T: ARRAY OF CHAR ): Clase;
VAR  M: CARDINAL;
BEGIN
  IF BuscaElemento(T,ST,M) THEN RETURN ST[M].TIPO
  ELSIF BuscaElemento(T,PR,M) THEN RETURN PR[M].TIPO
  ELSE RETURN NORESERVADO END;
END ClaseToken;


VAR  Linea: PACKED ARRAY [0..127] OF CHAR;
     I,J,L: CARDINAL;

PROCEDURE SaltaComent; FORWARD;

PROCEDURE Token ( VAR T: ARRAY OF CHAR; VAR C: Clase ): BOOLEAN;
BEGIN
  REPEAT
    IF I >= L THEN
      IF NOT LeeLinea(Linea,L) THEN RETURN FALSE ELSE I := 0 END
    END;
    Salta (Linea,Alfanumerico[' '],I); J := I;
    CASE Linea[I] OF
      'A'..'I',
      'a'..'z': Salta (Linea,Alfanumerico['A'..'Z','a'..'z','0'..'9','^'],I) !
      '0'..'9': Salta (Linea,Alfanumerico['0'..'9','C'],I);
                IF (Linea[I] = '.') & (Linea[I+1] IN Alfanumerico['0'..'9']) THEN
                  INC (I); Salta (Linea,Alfanumerico['0'..'9','E'],I) END  !
      '('     : IF Linea[I+1] = '0' THEN INC(I,2) ELSE INC(I) END     !
      ';'     : IF Linea[I+1] = '=' THEN INC(I,2) ELSE INC(I) END     !
      '<'     : IF Linea[I+1] = ')' THEN INC(I,2)
                ELSIF Linea[I+1] = '=' THEN INC(I,2) ELSE INC(I) END   !
      ')'     : IF Linea[I+1] = '=' THEN INC(I,2) ELSE INC(I) END     !
      '.'     : IF Linea[I+1] = '.' THEN INC(I,2) ELSE INC(I) END     !
      '0'     : IF Linea[I+1] = ')' THEN INC(I,2) ELSE INC(I) END     !
      "'"     : REPEAT INC(I) UNTIL Linea[I] = "'"; INC(I)            !
      '"'     : REPEAT INC(I) UNTIL Linea[I] = '"'; INC(I)
      ELSE INC(I)
    END;
    Copy(Linea,J,I-J,T); Concat(T,' ',T); C := ClaseToken(T);
    IF C = INICOM THEN SaltaComent
    ELSIF (C = COMILLA) OR (C = APOSTROFE) THEN C := NORESERVADO  END;
  UNTIL (C # INICOM) & (T[0] # EOL); RETURN TRUE
END Token;

PROCEDURE SaltaComent;
VAR  T: ARRAY [0..511] OF CHAR;
     I: CARDINAL;
     C: Clase;
BEGIN
  I := 1;
  REPEAT
    IF NOT Token(T,C) THEN HALT END;
    IF C = FINCOM THEN DEC(I)
    ELSIF C = INICOM THEN INC(I) END
  UNTIL I = 0
END SaltaComent;
```

```
PROCEDURE TerminaScan;
BEGIN  CloseInput
END TerminaScan;

BEGIN
  InitReservadas; InitEstandares; J := 0; I := 0; L := 0; Linea := ''
END Scanner.
```

```
DEFINITION MODULE Conteo;                    (* $SEG := 31 *)

  (* Rutinas para la cuantificacion de operadores y
     operandos de programas escritos en Modula-2.        *)

  FROM Scanner IMPORT Clase;

  EXPORT QUALIFIED  IniciaConteo,
                    ProgramaOP,
                    IncN1,
                    IncN2,
                    VERIFICAn1,
                    VERIFICAn2,
                    ImportDecl,
                    ExportDecl,
                    ForwardDecl,
                    TerminaConteo;

  PROCEDURE IniciaConteo ( ID: ARRAY OF CHAR; Cl: Clase );
    (* Inicia la cuantificacion de n1, N1, n2 y N2 para el modulo o
       procedimiento ID de clase Cl.                       *)

  PROCEDURE ProgramaOP ( T: ARRAY OF CHAR ): BOOLEAN;
    (* Regresa el valor TRUE si el identificador T ha sido declarado
       previamente como un procedimiento y FALSE en caso contrario.  *)

  PROCEDURE IncN1;
    (* Incrementa N1 en 1.                                 *)

  PROCEDURE IncN2;
    (* Incrementa N2 en 1.                                 *)

  PROCEDURE VERIFICAn1 ( T: ARRAY OF CHAR; C: Clase );
    (* Verifica la utilizacion del operador T de clase C en el modulo
       o procedimiento que se analiza.                     *)

  PROCEDURE VERIFICAn2 ( T: ARRAY OF CHAR );
    (* Verifica la utilizacion del operando T en el modulo o
       procedimiento que se analiza.                       *)

  PROCEDURE ImportDecl;
    (* Rutina para el manejo de objetos que se importan de un modulo
       de libreria de manera no-calificada.                *)

  PROCEDURE ExportDecl;
    (* Rutina para el manejo de objetos que exporta un modulo en un
       modulo-programa, de manera calificada y no-calificada.  *)

  PROCEDURE ForwardDecl;
    (* Rutina para el manejo de procedimientos declarados FORWARD.  *)

  PROCEDURE TerminaConteo;
    (* Reporta n1, N1, n2, N2 y los parametros de Halstead del modulo
       o procedimiento que acaba de analizarse.            *)

END Conteo.
```

```
                (* $STANDARD := FALSE; $SPECIAL := TRUE; *)

IMPLEMENTATION MODULE Conteo;

  FROM Formulae  IMPORT LongHalstead, LongJensen, Volumen,
                        Nivel, Contenido, Esfuerzo;

  FROM General   IMPORT String16, AsignString16, StringEQ,
                        OpPTR, Op, BuscaOp, NuevoOp, AnulaOp,
                        TipoObjeto, ObjPTR, Obj, BuscaListaObj, NuevoObj,
                        WriteSp;

  FROM InOut     IMPORT WriteString, WriteCard, WriteLn, Read;

  FROM RealInOut IMPORT WriteReal;

  FROM Scanner   IMPORT Clase, Token;

  FROM Storage   IMPORT ALLOCATE, DEALLOCATE;

  IMPORT Terminal;

TYPE  Descriptor = PACKED RECORD
                     IDENT: String16;
                     REV: BOOLEAN;
                     SEG: Clase;
                     NIVEL: CARDINAL;
                     n1, n2, N1, N2: CARDINAL;
                     ListaEXPORT,ListaIMPORT: ObjPTR;
                     ListaOP: OpPTR
                   END;

CONST  MaxModProc = 32;

VAR  ModProc: ARRAY [0..MaxModProc-1] OF Descriptor;
     MNP, IMP, NivLex: CARDINAL;

PROCEDURE ProgramOP ( T: ARRAY OF CHAR ): BOOLEAN;
VAR  I,J: CARDINAL;
     P: ObjPTR;
     S: String16;
BEGIN
  IF BuscaOp(ModProc[IMP].ListaOP,T) THEN RETURN TRUE END;

  I := IMP;
  WHILE (I > 1) & (ModProc[I].SEG # MODULESIM) DO DEC(I) END;
  IF BuscaListaObj(ModProc[I].ListaIMPORT,T,P) & (P^.TIPO=OPERADOR) THEN RETURN TRUE END;

  AsignString16 (T,S); I := I + 1; J := IMP;
  WHILE (J >= I) DO
    WITH ModProc[J] DO
      IF (StringEQ(IDENT,S) = #) & (NIVEL <= NivLex) THEN RETURN TRUE
      ELSE DEC(J) END
    END
  END;
  RETURN FALSE
END ProgramOP;
```

```
VAR  OpREFER: PACKED ARRAY [VERTICAL..WITHSIM] OF BOOLEAN;
     StREFER: PACKED ARRAY [ABSST..VALST] OF BOOLEAN;

PROCEDURE IniciaREFER;
VAR C: Clase;
BEGIN
  FOR C := VERTICAL TO WITHSIM DO OpREFER[C] := FALSE END;
  FOR C := ABSST TO VALST DO StREFER[C] := FALSE END;
END IniciaREFER;


TYPE  OpnPTR = POINTER TO Opn;
      Opn = RECORD
             ID: String16;
             IZQ,DER: OpnPTR
            END;

VAR  Raiz: OpnPTR;

PROCEDURE IniciaConteo ( ID: ARRAY OF CHAR; Cl: Clase );
VAR  I: CARDINAL;
     S: String16;
BEGIN
  AsignString16(ID,S); I := 2;
  LOOP
    WITH ModProc[I] DO
      IF (StringEQ(S,IDENT) = 0) & (SEG = FORWARDSIM) THEN IMP := I;
        Raiz := NIL; IniciaREFER; ModProc[I].SEG := PROCEDURESIM; RETURN
      ELSIF I < MMP THEN INC(I) ELSE EXIT END
    END
  END;
  IF MMP = MaxModProc - 1 THEN WriteString(' Demasiados procedimientos!'); HALT END;
  INC(MMP); IMP := MMP; INC(NivLex,1);
  WriteLn; WriteSp(NivLex*2);
  IF Cl = MODULESIM THEN WriteString ('MODULE ')
  ELSE WriteString ('PROCEDURE '); I := IMP - 1;
    LOOP
      WITH ModProc[I] DO
        IF NOT REV THEN NuevoOp(ModProc[I].ListaOP,S); EXIT
        ELSIF I>0 THEN DEC(I) END
      END
    END
  END;
  WITH ModProc[IMP] DO
    WriteString(ID); AsignString16(S,IDENT); SEG := Cl; REV := FALSE;
    NIVEL := NivLex; n1 := 0; n2 := 0; N1 := 0; N2 := 0;
    ListaIMPORT := NIL; ListaEXPORT := NIL; ListaOP := NIL
  END;
  Raiz := NIL; IniciaREFER;
END IniciaConteo;

PROCEDURE IncN1;
BEGIN
  INC (ModProc[IMP].N1)
END IncN1;
```

```
PROCEDURE IncN2;
BEGIN
  INC (ModProc[IMP].N2)
END IncN2;

VAR  OpList: OpPTR;

PROCEDURE VERIFICAn1 ( T: ARRAY OF CHAR; C: Clase );
BEGIN
  CASE C OF
    VERTICAL..WITHSIM: IF NOT OpREFER[C] THEN OpREFER[C] := TRUE ELSE RETURN END!
    ABSST..VALST: IF NOT StREFER[C] THEN StREFER[C] := TRUE ELSE RETURN END !
    NORESERVADO:  IF BuscaOp(OpList,T) THEN RETURN ELSE NuevaOp(OpList,T) END
    ELSE RETURN
  END;
  INC(ModProc[IMP].n1)
END VERIFICAn1;

PROCEDURE VERIFICAn2 ( T: ARRAY OF CHAR );

  PROCEDURE BUSCAOpn ( T: ARRAY OF CHAR; VAR P: OpnPTR );
  VAR  S: String16;
  BEGIN AsignString16(T,S);
    IF P = NIL THEN
      NEW (P);
      WITH P^ DO
        AsignString16 (S,ID); IZQ := NIL; DER := NIL; INC(ModProc[IMP].n2);
      END
    ELSIF StringEQ(S,P^.ID) < # THEN BUSCAOpn (S,P^.IZQ)
    ELSIF StringEQ(S,P^.ID) > # THEN BUSCAOpn (S,P^.DER)
    END
  END BUSCAOpn;

BEGIN
  BUSCAOpn(T,Raiz)
END VERIFICAn2;

PROCEDURE ForwardDecl;
BEGIN
  WriteString(': FORWARD'); WriteLn; DEC(NivLex,1);
  ModProc[NMP].SEG := FORWARDSIM
END ForwardDecl;

PROCEDURE ImportDecl;
VAR  R: CHAR;
     C: Clase;
     P,Q: ObjPTR;
     OK: BOOLEAN;
     I: CARDINAL;
     T,ID: String16;
BEGIN
  IF NOT Token(ID,C) THEN HALT END;  (* Identificador del modulo *)
  IF NOT Token(T,C) THEN HALT END;   (* IMPORT *)
  NEW(P); AsignString16(ID,T); P^.ID := T; P^.TIPO := OPERADOR; P^.LIGA := NIL;
  NuevoObj(ModProc[IMP].ListaIMPORT,P);
  I := 1; OK := FALSE;
```

```
  REPEAT
    WITH ModProc[I] DO
      OK := (StringEQ(IDENT,T) = 0) & (SEB = MODULESIM) & (NIVEL <= NivLex)
    END;
    IF NOT OK THEN INC(I) END
  UNTIL (I > NMP) OR OK;
  IF OK THEN
    LOOP
      IF NOT Token(T,C) THEN HALT END; (* Objeto importado *)
      IF BuscaListaObj(ModProc[I].ListaEXPORT,T,P) THEN
        NEW(Q); Q^ := P^; Q^.LIGA := NIL; NuevoObj(ModProc[IMP].ListaIMPORT,Q);
      END;
      IF NOT Token(T,C) THEN HALT END; (* ',' o ';' *)
      IF C = PUNCOMA THEN EXIT END
    END
  ELSE
    Terminal.WriteLn; Terminal.WriteString('Importacion de objetos del modulo ');
    Terminal.WriteString(ID); Terminal.WriteLn;
    LOOP
      IF NOT Token(T,C) THEN HALT END; (* Objeto importado *)
      Terminal.WriteString('  '); Terminal.WriteString(T);
      Terminal.WriteString(' es un Procedimiento ? (S) (N) ');
      Terminal.Read(R); R := CAP(R); Terminal.WriteLn;
      NEW(P); AsignString16(T,P^.ID); P^.LIGA := NIL;
      IF R = 'S' THEN P^.TIPO := OPERADOR ELSE P^.TIPO := OPERANDO END;
      NuevoObj (ModProc[IMP].ListaIMPORT,P);
      IF NOT Token(T,C) THEN HALT END; (* ',' o ';' *)
      IF C = PUNCOMA THEN EXIT END
    END
  END
END ImportDecl;

PROCEDURE ExportDecl;
VAR  T: String16;
     P: ObjPTR;
     C: Clase;
     R: CHAR;
BEGIN
  Terminal.WriteLn; Terminal.WriteString('Exportacion de objetos del modulo ');
  Terminal.WriteString(ModProc[IMP].IDENT); Terminal.WriteLn;
  LOOP
    IF NOT Token(T,C) THEN HALT END; (* Objeto importado *)
    IF C # QUALIFISIM THEN
      Terminal.WriteString('  '); Terminal.WriteString(T);
      Terminal.WriteString(' es un Procedimiento ? (S) (N) ');
      Terminal.Read(R); R := CAP(R); Terminal.WriteLn;
      NEW(P); AsignString16(T,P^.ID); P^.LIGA := NIL;
      IF R = 'S' THEN P^.TIPO := OPERADOR ELSE P^.TIPO := OPERANDO END;
      NuevoObj (ModProc[IMP].ListaEXPORT,P);
      IF NOT Token(T,C) THEN HALT END; (* ',' o ';' *)
      IF C = PUNCOMA THEN EXIT END
    END
  END
END ExportDecl;
```

```
PROCEDURE TerminaConteo;
  PROCEDURE ANULAOpn ( P: OpnPTR );
  BEGIN
    IF P = NIL THEN RETURN END;
    IF P^.IZQ # NIL THEN ANULAOpn (P^.IZQ) END;
    IF P^.DER # NIL THEN ANULAOpn (P^.DER) END;
    DISPOSE(P);
  END ANULAOpn;

VAR  TN,VOC: CARDINAL;
     VOL,NIV,CI,E,TI: REAL;
BEGIN
  WITH ModProc[IMP] DO REV := TRUE;
    WriteLn; WriteSp(NivLex*2);
    WriteString('Medicion de '); WriteString(IDENT); WriteLn;
    WriteSp (NivLex*2+1);
    WriteString ('n1:'); WriteCard (n1,2);
    WriteString ('  N1:'); WriteCard (N1,2);
    WriteString ('  n2:'); WriteCard (n2,2);
    WriteString ('  N2: '); WriteCard (N2,2); WriteLn;

    WriteSp (NivLex*2+1);
    TN := N1 + N2; VOC := n1 + n2;
    WriteString('Vocabulario:'); WriteCard (VOC,2);
    WriteString('  Longitud de implementacion: '); WriteCard(TN,2); WriteLn;

    WriteSp (NivLex*2+1);
    WriteString('Longitud calculada segun Halstead: ');
    WriteReal(LongHalstead(n1,n2),5); WriteLn;
    IF LongJensen(n1,n2) > 0.0 THEN
      WriteSp (NivLex*2+1);
      WriteString('Longitud calculada segun Jensen: ');
      WriteReal(LongJensen(n1,n2),5); WriteLn;
    END;

    VOL := Volumen(VOC,TN);
    WriteSp(NivLex*2+1);
    WriteString('Volumen: '); WriteReal(VOL,3); WriteString(' bits'); WriteLn;

    NIV := Nivel(n1,n2,N2);
    WriteSp(NivLex*2+1);
    WriteString('Nivel: '); WriteReal(NIV,3);
    IF NIV > 0.0 THEN WriteString (' => Dificultad: '); WriteReal(1.0/NIV,3) END;
    WriteLn;

    CI := Contenido(NIV,VOL);
    WriteSp(NivLex*2+1);
    WriteString('Contenido intelectual: '); WriteReal(CI,3); WriteLn;

    E := Esfuerzo(NIV,VOL);
    WriteSp(NivLex*2+1);
    WriteString('Esfuerzo de programacion: '); WriteReal(E,3); WriteLn
  END;
```

```
  IF Raiz # NIL THEN ANULAOpn(Raiz); Raiz := NIL END;
  IF OpList # NIL THEN AnulaOp(OpList) END;
  IniciaREFER; DEC(NivLex,1);
  REPEAT
    IF IMP > 0 THEN DEC(IMP) END
  UNTIL (NOT ModProc[IMP].REV) OR (ModProc[1].REV);
END TerminaConteo;

BEGIN
  NMP := 0; IMP := 0; NivLex := 0; Raiz := NIL; OpList := NIL
END Conteo.
```

```
DEFINITION MODULE General;              (* $SEG := 30 *)

   (* Rutinas de proposito general para el apoyo de
     la cuantificacion de operadores y operandos de
     programas escritos en Modula-2.                *)

   EXPORT QUALIFIED  String16,
                     AsignString16,
                     StringEQ,
                     OpPTR,
                     Op,
                     BuscaOp,
                     NuevoOp,
                     AnulaOp,
                     TipoObjeto,
                     ObjPTR,
                     Obj,
                     BuscaListaObj,
                     NuevoObj,
                     WriteSp;

TYPE  String16 = ARRAY [0..15] OF CHAR;

PROCEDURE AsignString16 ( ORIG: ARRAY OF CHAR; VAR DEST: String16 );
  (* Transfiere el arreglo de caracteres ORIG a la cuerda DEST de
     tipo String16, dejando en blanco los caracteres no ocupados.  *)

PROCEDURE StringEQ ( S1,S2: ARRAY OF CHAR ): INTEGER;
  (* Compara dos cuerdas regresando -1 si S1 es menor que S2,
     0 si S1 es igual a S2 y 1 si S1 es mayor que S2.              *)


TYPE  OpPTR = POINTER TO Op;
      Op = RECORD
             ID: String16;
             LIGA: OpPTR
           END;

PROCEDURE BuscaOp ( HEAD: OpPTR; S: ARRAY OF CHAR ): BOOLEAN;
  (* Busca en la lista apuntada por HEAD el nodo de tipo Op
     cuyo campo ID sea igual a S, regresando TRUE si lo encuentra
     y FALSE en caso contrario.                                    *)

PROCEDURE NuevoOp ( VAR HEAD: OpPTR; S: ARRAY OF CHAR );
  (* Agrega a la lista apuntada por HEAD un nuevo nodo de tipo Op
     cuyo campo ID es S.                                           *)

PROCEDURE AnulaOp ( VAR HEAD: OpPTR );
  (* Libera la memoria ocupada por la lista apuntada por HEAD      *)
```

```
TYPE   TipoObjeto = (OPERADOR,OPERANDO);
       ObjPTR = POINTER TO Obj;
       Obj = RECORD
               ID: String16;
               TIPO: TipoObjeto;
               LIGA: ObjPTR
             END;

PROCEDURE BuscaListaObj ( L: ObjPTR; T: ARRAY OF CHAR; VAR P: ObjPTR ): BOOLEAN;
  (* Busca en la lista de objetos apuntada por L la cuerda T
     regresando TRUE si es encontrada y FALSE en caso contrario;
     si el objeto es encontrado, P estara apuntando a el.          *)

PROCEDURE NuevoObj ( VAR HEAD: ObjPTR; Q: ObjPTR );
  (* Agrega a la lista apuntada por HEAD un nodo de tipo Obj apuntado
     por Q.                                                        *)


PROCEDURE WriteSp ( N: CARDINAL );
  (* Escribe N espacios en la unidad de salida asignada por default.  *)

END General.
```

```
IMPLEMENTATION MODULE General;

  FROM InOut   IMPORT Write;

  FROM Storage IMPORT ALLOCATE, DEALLOCATE;

PROCEDURE AsignString16 ( ORIG: ARRAY OF CHAR; VAR DEST: String16 );
VAR  I: CARDINAL;
BEGIN
                    (* $RANGE := FALSE *)
  DEST := '                ';  I := 0;
  WHILE (I <= 15) & (ORIG[I] # 0C) DO DEST[I] := ORIG[I]; INC(I) END
END AsignString16;    (* $RANGE := TRUE *)

PROCEDURE StringEQ ( S1,S2: ARRAY OF CHAR ): INTEGER;
VAR I,MAX: CARDINAL;
BEGIN           (* $RANGE := FALSE *)
  I := 0;
  MAX := HIGH(S1);
  IF HIGH(S2) < MAX THEN MAX := HIGH(S2) END;
  REPEAT
    IF S1[I] < S2[I] THEN RETURN -1 END;
    IF S1[I] > S2[I] THEN RETURN 1 END;
    INC(I)
  UNTIL I > MAX;
  IF HIGH(S2) > HIGH(S1) THEN RETURN -1
  ELSIF HIGH(S2) < HIGH(S1) THEN RETURN 1 ELSE RETURN 0 END
END StringEQ;   (* $RANGE := TRUE *)


PROCEDURE BuscaOp ( HEAD: OpPTR; S: ARRAY OF CHAR ): BOOLEAN;
VAR  T: String16;
     P: OpPTR;
BEGIN
  AsignString16(S,T); P := HEAD;
  WHILE P # NIL DO
    IF StringEQ(P^.ID,T) = 0 THEN RETURN TRUE
    ELSE P := P^.LIGA END
  END; RETURN FALSE
END BuscaOp;

PROCEDURE NuevoOp ( VAR HEAD: OpPTR; S: ARRAY OF CHAR );
VAR  P: OpPTR;
BEGIN
  IF HEAD # NIL THEN
    P := HEAD;
    WHILE P^.LIGA # NIL DO P := P^.LIGA END;
    NEW(P^.LIGA); P := P^.LIGA
  ELSE NEW(P); HEAD := P END;
  WITH P^ DO AsignString16(S,ID); LIGA := NIL END;
END NuevoOp;
```

```
PROCEDURE AnulaOp ( VAR HEAD: OpPTR );
VAR P: OpPTR;
BEGIN
  IF HEAD = NIL THEN RETURN END;
  REPEAT
    P := HEAD^.LIGA; DISPOSE(HEAD); HEAD := P
  UNTIL HEAD = NIL
END AnulaOp;


PROCEDURE BuscaListaObj ( L: ObjPTR; T: ARRAY OF CHAR; VAR P: ObjPTR ): BOOLEAN;
VAR  S: String16;
BEGIN
  P := L;
  IF L # NIL THEN
    AsignString16(T,S);
    REPEAT
      IF StringEQ(S,P^.ID) = # THEN RETURN TRUE
      ELSE P := P^.LIGA END
    UNTIL P = NIL;
  END; RETURN FALSE
END BuscaListaObj;

PROCEDURE NuevoObj ( VAR HEAD: ObjPTR; Q: ObjPTR );
VAR  P: ObjPTR;
BEGIN
  IF HEAD # NIL THEN
    P := HEAD;
    WHILE P^.LIGA # NIL DO P := P^.LIGA END;
    P^.LIGA := Q
  ELSE HEAD := Q END;
END NuevoObj;


PROCEDURE WriteSp ( N: CARDINAL );
BEGIN
  REPEAT IF N > # THEN Write(' '); DEC(N) END UNTIL N = #
END WriteSp;

END General.
```

```
DEFINITION MODULE Formulae;                                    (* $SEG := 35 *)

  (* Rutinas para el calculo de los indicadores de Halstead.                *)

  EXPORT QUALIFIED  LongHalstead,
                    LongJensen,
                    Volumen,
                    Nivel,
                    Contenido,
                    Esfuerzo;

PROCEDURE LongHalstead ( n1,n2: CARDINAL ): REAL;
  (* Calculo de la longitud de un procedimiento o modulo segun M. Halstead *)

PROCEDURE LongJensen ( n1,n2: CARDINAL ): REAL;
  (* Calculo de la longitud de un procedimiento o modulo segun Jensen      *)

PROCEDURE Volumen ( n,N: CARDINAL ): REAL;
  (* Calculo del volumen de un procedimiento o modulo.                     *)

PROCEDURE Nivel ( n1,n2,N2: CARDINAL ): REAL;
  (* Calculo del nivel de un procedimiento o modulo.                       *)

PROCEDURE Contenido ( L,V: REAL ): REAL;
  (* Calculo del contenido intelectual de un procedimiento o modulo.       *)

PROCEDURE Esfuerzo ( L,V: REAL ): REAL;
  (* Calculo del esfuerzo de programacion de un procedimiento o modulo.    *)

END Formulae.
```

```
IMPLEMENTATION MODULE Formulae;

  FROM MathLib0  IMPORT ln;


PROCEDURE Log2 ( R: REAL ): REAL;
CONST C = 1.442695;
BEGIN
  RETURN C * ln(R)
END Log2;

PROCEDURE LongHalstead ( n1,n2: CARDINAL ): REAL;
VAR  A,B: REAL;
BEGIN
  A := 0.0; B := 0.0;
  IF n1 > 0 THEN A := FLOAT(n1) * Log2(FLOAT(n1)) END;
  IF n2 > 0 THEN B := FLOAT(n2) * Log2(FLOAT(n2)) END;
  RETURN A + B
END LongHalstead;

PROCEDURE LongJensen ( n1,n2: CARDINAL ): REAL;
  (* Calcula la longitud de un programa segun la formula propuesta por
     H. Jensen. Si no puede calcularse la funcion regresa un -1.0     *)

  PROCEDURE Fac ( N: CARDINAL ): REAL;
    (* Calcula el factorial de N <=> N < 34.
       Si N >= 34 entonces la funcion regresa un 0.0                  *)
  BEGIN
    IF N > 34 THEN RETURN 0.0 END;
    IF N = 0 THEN RETURN 1.0 ELSE RETURN FLOAT(N) * Fac(N-1) END
  END Fac;

BEGIN
  IF (Fac(n1) = 0.0) OR (Fac(n2) = 0.0) THEN RETURN -1.0
  ELSE RETURN Log2(Fac(n1)) + Log2(Fac(n2)) END
END LongJensen;

PROCEDURE Volumen ( n,N: CARDINAL ): REAL;
BEGIN
  RETURN FLOAT (N) * Log2(FLOAT(n))
END Volumen;

PROCEDURE Nivel ( n1,n2,N2: CARDINAL ): REAL;
BEGIN
  IF (n1 = 0) & (N2 # 0) THEN RETURN FLOAT(n2) / FLOAT(N2)
  ELSIF (N2 = 0) AND (n1 # 0) THEN RETURN 2.0 / FLOAT(n1)
  ELSIF (n1 = 0) & (n2 = 0) THEN RETURN 0.0
  ELSE RETURN (2.0 / FLOAT(n1)) * (FLOAT(n2) / FLOAT(N2)) END
END Nivel;
```

```
PROCEDURE Contenido ( L,V: REAL ): REAL;
BEGIN
  RETURN L * V
END Contenido;

PROCEDURE Esfuerzo ( L,V: REAL ): REAL;
BEGIN
  IF L > 0.0 THEN RETURN V / L ELSE RETURN 0.0 END
END Esfuerzo;

END Formulae.
```

```
                        (* $SPECIAL := TRUE *)
MODULE Halstead;

  FROM Conteo  IMPORT IniciaConteo, ProgramaOP, IncN1, IncN2, VERIFICAn1, VERIFICAn2,
                      ForwardDecl, ImportDecl, ExportDecl, TerminaConteo;

  FROM InOut   IMPORT OpenOutput, CloseOutput, Done, WriteString, WriteLn;

  FROM Scanner IMPORT Clase, IniciaScan, TerminaScan, Token;

VAR  NIVEL: CARDINAL;
     T: ARRAY [0..79] OF CHAR;
     C: Clase;

PROCEDURE NuevoModProc ( C: Clase );
VAR  T,ID: ARRAY[0..31] OF CHAR;
     C1: Clase;
BEGIN
  IF NOT Token(ID,C1) THEN HALT END;
  IniciaConteo(ID,C);
  REPEAT
    IF NOT Token(T,C1) THEN HALT END;
    IF C1=FROMSIM THEN ImportDecl
    ELSIF C1 = EXPORTSIM THEN ExportDecl END
  UNTIL (C1=BEGINSIM) OR (C1=PROCEDURESIM) OR (C1=MODULESIM) OR (C1=FORWARDSIM);
  IF C1 = BEGINSIM THEN NIVEL := 1; IncN1; VERIFICAn1(T,C1)
  ELSIF C1 = FORWARDSIM THEN ForwardDecl
  ELSE NuevoModProc(C1) END
END NuevoModProc;

BEGIN
  IniciaScan;
  WriteString(' Archivo de salida: '); OpenOutput('');
  IF NOT Done THEN HALT END;

  WriteString('MEDICION DE PROGRAMAS ESCRITOS EN MODULA-2     Version 1.0');
  WriteLn; WriteString(' Reporte de los indicadores de Halstead'); WriteLn;

  WHILE Token(T,C) DO
    IF (C # NORESERVADO) THEN
      CASE C OF
        MODULESIM,
        PROCEDURESIM: NuevoModProc(C)

        BEGINSIM:     NIVEL := 1; IncN1; VERIFICAn1(T,C)

        CASESIM,
        FORSIM,
        IFSIM,LOOPSIM,
        WITHSIM,WHILESIM: INC(NIVEL); IncN1; VERIFICAn1(T,C) |

        ENDSIM: IF NIVEL # 0 THEN DEC(NIVEL);
                  IF NIVEL = 0 THEN TerminaConteo END
                END
```

```
      VERTICAL..PARIZQ,
      PARENCIZQ,
      AMPER..MAYIGUAL,
      ANDSIM,BYSIM,DIVSIM,
      ELSESIM,ELSIFSIM,
      EXITSIM,INSIM,
      MODSIM,NOTSIM,
      ORSIM,
      REPEATSIM,
      ABSST..EXCLST,
      FLOATST..NEWST,
      ODDST..REALST,
      TRUNCST..VALST,
      RETURNSIM:      IF NIVEL > 0 THEN IncN1; VERIFICAn1(T,C) END  !

      FALSEST,NILST,TRUEST: IF NIVEL > 0 THEN IncN2; VERIFICAn2(T) END

      ELSE
    END
  ELSIF (NIVEL > 0) & (ProgramOP(T)) THEN VERIFICAn1(T,C); IncN1
  ELSIF (NIVEL > 0) & (C = NORESERVADO) THEN IncN2; VERIFICAn2(T) END;
END;
TerminaScan; CloseOutput

END Halstead.
```

Ejemplos

A continuación presentamos la resultados de la medición de 3 programas. El primero es la implementación del algoritmo de Euclides para encontrar el máximo divisor común de 2 números cardinales (mencionado en el capítulo 1); el segundo es un ejemplo de la utilización en módulos locales y el tercero un ejemplo de cómo se accesa directamente la memoria principal desde un programa en Modula-2.

Finalmente presentamos en la tabla 1 los resultados que se obtuvieron al analizar los módulos que conforman el sistema de medición (esto es, medimos al medidor) y comentaremos su consistencia en el siguiente capítulo.

```
                        (* $UPCASE := TRUE *)
MODULE FirstTest;

FROM InOut IMPORT WRITESTRING, WRITECARD, WRITELN, READCARD, DONE;

VAR I,J: CARDINAL;

PROCEDURE MDC ( A,B: CARDINAL ): CARDINAL;
VAR R,G: CARDINAL;
BEGIN
  IF A = 0 THEN RETURN B END;
  IF B = 0 THEN RETURN A END;
  LOOP
    G := A DIV B; R := A - B * G;
    IF R = 0 THEN RETURN B END;
    A := B; B := R
  END
END MDC;

BEGIN
  REPEAT
    WRITESTRING ('Primer numero: '); READCARD(I);
    WRITESTRING ('Segundo numero: '); READCARD(J);
    WRITESTRING ('Maximo divisor comun: '); WRITECARD (MDC(I,J),5); WRITELN
  UNTIL (I=0) OR (J=0)
END FirstTest.
```

```
MEDICION DE PROGRAMAS ESCRITOS EN MODULA-2     Version 1.0
 Reporte de los indicadores de Halstead

  MODULE FirstTest
    PROCEDURE MDC
    Medicion de MDC
     n1:10  N1:24  n2: 5  N2: 20
     Vocabulario:15  Longitud de implementacion: 44
     Longitud calculada segun Halstead: 4.4828920E+01
     Longitud calculada segun Jensen: 2.8697948E+01
     Volumen: 1.7198317E+02 bits
     Nivel: 5.0000000E-02 => Dificultad: 2.0000000E+01
     Contenido intelectual: 8.5951585E+00
     Esfuerzo de programacion: 3.4380631E+03

  Medicion de FirstTest
   n1:12  N1:30  n2: 7  N2: 12
   Vocabulario:19  Longitud de implementacion: 42
   Longitud calculada segun Halstead: 6.2671027E+01
   Longitud calculada segun Jensen: 4.1134657E+01
   Volumen: 1.7841295E+02 bits
   Nivel: 9.7222223E-02 => Dificultad: 1.0285713E+01
   Contenido intelectual: 1.7345702E+01
   Esfuerzo de programacion: 1.8351045E+03
```

```
MODULE ModDemo;       (* (c) 1983 by Volition Systems.  All rights reserved. *)

FROM InOut IMPORT
  WriteString, WriteCard, WriteLn;

PROCEDURE WriteVal (val: CARDINAL);
BEGIN
  WriteString("Value is:");
  WriteCard(val,3);
  WriteLn;
END WriteVal;

MODULE NumberGenerator;
EXPORT NextVal;

  VAR CurVal: CARDINAL;

  PROCEDURE NextVal(): CARDINAL;
  BEGIN
    INC(CurVal);
    RETURN CurVal;
  END NextVal;

BEGIN
  CurVal := 0;
END NumberGenerator;

VAR value,count: CARDINAL;

BEGIN
  FOR count := 1 TO 10 DO
    value := NextVal();
    WriteVal(value);
  END;
  WriteString("That's all, folks!");
END ModDemo.

(* COMENTARIOS SOBRE ModDemo

ESTE PROGRAMA CONTIENE UN MODULO DENOMINADO NumberGenerator, EL CUAL ES
UN MODULO LOCAL PORQUE ESTA DECLARADO DENTRO DE OTRO MODULO.  LOS MODULOS
LOCALES SON EMPLEADOS PARA AGRUPAR DECLARACIONES RELACIONADAS EN UN SOLO LUGAR
Y TAMBIEN PARA PREVENIR QUE EL RESTO DEL PROGRAMA TENGA ACCESO A OBJETOS QUE
NO DEBE ALTERAR.

NumberGenerator PROPORCIONA LA FUNCION NextVal LA CUAL REGRESA UN VALOR
DIFERENTE CADA VEZ QUE ES LLAMADA.  EL MODULO EMPLEA LA VARIABLE CurVal PARA
MANTENER EL VALOR ACTUAL, PERO DADO QUE NO ES EXPORTADA, ESTA NO ES VISIBLE
FUERA DE NumberGenerator.  NOTE QUE EL MODULO LE ASIGNA A CurVal EL VALOR
INICIAL DE CERO.

NOTE TAMBIEN QUE MODULA-2 PERMITE DECLARAR TIPOS, VARIABLES, PROCEDIMIENTOS
Y MODULOS EN EL ORDEN QUE EL PROGRAMADOR CONSIDERE CONVENIENTE.          *)
```

```
MEDICION DE PROGRAMAS ESCRITOS EN MODULA-2     Version 1.0
 Reporte de los indicadores de Halstead

  MODULE ModDemo
    PROCEDURE WriteVal
    Medicion de WriteVal
     n1: 7  N1:10  n2: 3  N2:  3
     Vocabulario:10  Longitud de implementacion: 13
     Longitud calculada segun Halstead: 2.4406371E+01
     Longitud calculada segun Jensen: 1.4884169E+01
     Volumen: 4.3185062E+01 bits
     Nivel: 2.8571429E-01 => Dificultad: 3.4999997E+00
     Contenido intelectual: 1.2338589E+01
     Esfuerzo de programacion: 1.5114769E+02

    MODULE NumberGenerator
      PROCEDURE NextVal
      Medicion de NextVal
       n1: 5  N1: 6  n2: 1  N2:  2
       Vocabulario: 6  Longitud de implementacion:  8
       Longitud calculada segun Halstead: 1.1609640E+01
       Longitud calculada segun Jensen: 6.9868899E+00
       Volumen: 2.0679697E+01 bits
       Nivel: 2.0000000E-01 => Dificultad: 5.0000000E+00
       Contenido intelectual: 4.1359395E+00
       Esfuerzo de programacion: 1.0339847E+02

    Medicion de NumberGenerator
     n1: 3  N1: 3  n2: 2  N2:  2
     Vocabulario: 5  Longitud de implementacion:  5
     Longitud calculada segun Halstead: 6.7548866E+00
     Longitud calculada segun Jensen: 3.5849618E+00
     Volumen: 1.1609640E+01 bits
     Nivel: 6.6666669E-01 => Dificultad: 1.5000000E+00
     Contenido intelectual: 7.7397603E+00
     Esfuerzo de programacion: 1.7414459E+01

  Medicion de ModDemo
   n1: 7  N1:13  n2: 6  N2:  7
   Vocabulario:13  Longitud de implementacion: 20
   Longitud calculada segun Halstead: 3.5161254E+01
   Longitud calculada segun Jensen: 2.1791859E+01
   Volumen: 7.4008779E+01 bits
   Nivel: 2.4489798E-01 => Dificultad: 4.0833330E+00
   Contenido intelectual: 1.8124601E+01
   Esfuerzo de programacion: 3.0220251E+02
```

```
MODULE LowLevel3;     (* (c) 1983 by Volition Systems.  All rights reserved. *)

FROM SYSTEM IMPORT WORD, ADDRESS, ADR, SIZE, TSIZE;

FROM InOut IMPORT WriteString, WriteHex, WriteLn;

PROCEDURE OnesComplement (VAR arg: WORD);
BEGIN
  arg := WORD(BITSET(arg) / {0..15});
END OnesComplement;

PROCEDURE DumpMemory (memptr: ADDRESS; words: CARDINAL);
BEGIN
  WHILE words # 0 DO
    WriteHex(CARDINAL(memptr^), 6);
    INC(memptr, TSIZE(WORD));  (* next word in memory *)
    DEC(words);
  END;
  WriteLn;
END DumpMemory;

VAR a: ARRAY [1..5] OF CARDINAL;
    c, inx: CARDINAL;
    stkbase [0180H]: WORD;   (* variable resides at hex address 180 *)
    b: BOOLEAN;
    r: RECORD
         a,b,c: INTEGER;
         ch: CHAR;
       END;

BEGIN
  c := 0FF0H;
  WriteString(" complement of"); WriteHex(c,5);
  WriteString(" is"); OnesComplement(c); WriteHex(c,5);
  WriteLn;

  b := TRUE;
  WriteString(" complement of"); WriteHex(CARDINAL(b),5);
  WriteString(" is"); OnesComplement(b); WriteHex(CARDINAL(b),5);
  WriteLn;

  FOR inx := 1 TO 5 DO a[inx] := inx + 3 END;
  WITH r DO a := 1; b := 3; c := 5; ch := ' ' END;

  DumpMemory(ADR(a), SIZE(a));
  DumpMemory(ADR(r), SIZE(r));
  DumpMemory(ADR(stkbase), 5);
  DumpMemory(0A1F0H , 5);

END LowLevel3.

(* COMENTARIOS A LowLevel3

STE PROGRAMA DEMUESTRA EL USO DE LOS TIPOS DE DATOS WORD Y ADDRESS PARA
EXER ACCESO A NIVEL DE MAQUINA.  EL PROCEDIMIENTO  OnesComplement PUEDE
CEPTAR COMO ARGUMENTO CUALQUIER OBJETO QUE OCUPE UNA PALABRA DE MEMORIA;
) CONVIERTE A UN CONJUNTO DE UNA PALABRA Y REALIZA EN EL UN OR-EXCLUSIVO  *)
```

```
MEDICION DE PROGRAMAS ESCRITOS EN MODULA-2     Version 1.0
 Reporte de los indicadores de Halstead

  MODULE LowLevel3
    PROCEDURE OnesComplement
    Medicion de OnesComplement
     n1: 6  N1: 7  n2: 7  N2:  8
     Vocabulario:13  Longitud de implementacion: 15
     Longitud calculada segun Halstead: 3.5161254E+01
     Longitud calculada segun Jensen: 2.1791059E+01
     Volumen: 5.5506587E+01 bits
     Nivel: 2.9166669E-01 => Dificultad: 3.4285712E+00
     Contenido intelectual: 1.6189422E+01
     Esfuerzo de programacion: 1.9030828E+02

    PROCEDURE DumpMemory
    Medicion de DumpMemory
     n1:12  N1:21  n2: 6  N2:  7
     Vocabulario:18  Longitud de implementacion: 28
     Longitud calculada segun Halstead: 5.8529319E+01
     Longitud calculada segun Jensen: 3.8327304E+01
     Volumen: 1.1675789E+02 bits
     Nivel: 1.4285714E-01 => Dificultad: 6.9999995E+00
     Contenido intelectual: 1.6679699E+01
     Esfuerzo de programacion: 8.1730518E+02

  Medicion de LowLevel3
   n1:17  N1:88  n2:17  N2: 44
   Vocabulario:34  Longitud de implementacion: 132
   Longitud calculada segun Halstead: 1.3897372E+02
   Longitud calculada segun Jensen: 9.6675195E+01
   Volumen: 6.7154502E+02 bits
   Nivel: 4.5454545E-02 => Dificultad: 2.2000002E+01
   Contenido intelectual: 3.0524773E+01
   Esfuerzo de programacion: 1.4773992E+04
```

	Voc	Long	L H	D H	L J	D J	Vol [bits]	Niv	Dif	Cl	Esfuerzo
Scanner	11	20	27.1	.355	16.3	.210	69.1	.300	3.33	20.7	230.6
IniciaScan	11	13	28.7	1.211	17.8	.369	44.9	.250	4.0	11.2	179.8
Salta	15	22	46.8	1.093	29.8	.355	85.9	.090	11.0	7.8	945.4
LeeLinea	20	46	67.0	.456	44.1	.040	198.8	.070	14.2	13.9	2833.0
BuscaElemento	34	102	141.1	.383	98.7	.031	518.9	.024	41.2	12.5	21405.4
InitReservadas	4	4	8.0	1.0	4.5	.145	8.0	.500	2.0	4.0	16
InitID	149	569	1044.0	.834	--	--	4107.7	.202	4.93	833.1	20253.2
InitTIPO	137	568	941.4	.657	--	--	4031.6	.185	5.37	749.5	21685.5
InitEstandares	4	4	8.0	1.0	4.5	.145	8.0	.500	2.0	4.0	16
InitID	59	208	322.3	.549	--	--	1223.5	.207	4.81	254.1	5891.3
InitTIPO	59	208	322.3	.549	--	--	1223.5	.207	4.81	254.1	5891.3
ClaseToken	17	33	53.5	.623	34.7	.051	134.8	.083	11.9	11.3	1607.3
Token	69	352	353.0	.002	--	--	2150.2	.017	58.7	36.6	126295.9
SaltaComent	21	32	72.9	1.279	48.64	.520	140.5	.083	12.0	11.7	1686.6
TerminaScan	2	2	2	0.0	1.0	.5	2.0	1.0	1.0	2.0	2.0
General											
AsignString16	17	32	52.8	.652	34.09	.065	138.7	.10	10.0	13.0	13.0
StringEQ	21	78	74.1	.050	49.7	.362	342.6	.028	35.0	9.7	11991.0
BuscaOp	25	36	91.3	1.538	61.5	.711	167.1	.104	9.54	17.5	1595.7
NuevoOp	20	53	67.7	.278	44.8	.154	229.0	.046	21.3	10.7	4892.1
AnulaOp	15	24	44.8	.868	28.6	.195	93.7	.10	10.0	9.3	937.6
BuscaListaObj	25	41	91.3	1.228	61.5	.502	190.3	.092	10.0	17.5	2059.7
NuevoObj	14	31	39.5	.274	24.7	.20	118.0	.16	16.0	11.8	1180.2
WriteSp	12	16	33.2	1.000	21.0	.316	57.3	.111	9.0	6.3	516.2
Formulae											
Log2	7	7	13.6	.944	7.9	.129	19.6	.400	2.5	7.8	49.1
LongHalste	17	45	53.5	.190	34.7	.220	183.9	.060	14.6	12.5	2697.7
LongJensen	15	31	46.0	.480	29.8	.030	121.1	.103	9.6	12.5	1165.7
Fac	18	25	58.5	1.341	38.3	.533	104.2	.111	9.0	11.5	938.2
Volumen	8	11	17.5	.592	10.4	.046	33.0	.333	3.0	11.0	99.0
Nivel	19	64	62.6	.021	41.1	.357	271.8	.046	21.4	12.6	5825.7
Contenido	5	5	6.7	.350	3.5	.283	11.6	.666	1.5	7.7	17.4
Esfuerzo	9	12	19.6	.633	11.4	.043	38.0	.266	3.75	10.1	142.6
Halstead	84	218	454.6	1.003	341.4	.566	1393.5	.035	27.8	50.0	38819.6
NuevoModPr	33	80	135.2	.691	94.3	.179	403.5	.042	23.6	17.0	9533.9
				Prom:0.675		Prom:0.259				423.6	291413.7

TABLA 3.1 Indicadores de 4 modulos del sistema de medicion.

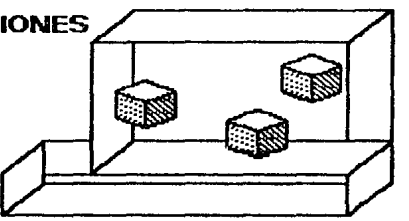
Voc: vocabulario (n1+n2) Long: longitud de implementacion (N1+N2)

L : longitud segun Halstead D : error normalizado en L
 H H H

L : longitud segun Jensen D : error normalizado en L
 J J J

Vol: volumen Niv: nivel Dif: dificultad

CAPITULO 4:
CONCLUSIONES



Si bien para el estimador de longitud Halstead reportò un error promedio normalizado (error normalizado = (ABS(N - N_H)) / N) de sòlo Ø.ØЗ, encontramos en la muestra utilizada un error promedio de Ø.675, discrepancia corroborada por otros investigadores en pruebas con otros lenguajes. En contraste, el estimador de longitud propuesto por Jensen resultò ser màs preciso (error promedio de Ø.259).

Por otra parte, fue interesante comprobar la consistencia de indicadores como el de dificultad (inverso del nivel) y esfuerzo de programaciòn. Por ejemplo, aunque los procedimientos que inicializan las tablas para el scanner (InitID e InitTIPO de InitReservadas e InitEstàndares) son los de longitud y volumen màs grande, su dificultad es efectivamente pequeña. En el otro extremo, el procedimiento Token del mismo mòdulo es el que tiene el indicador de dificultad màs grande y en concordancia con la realidad fue el procedimiento que màs se tardò en poner a punto. Caso anàlogo el del programa principal del mòdulo Halstead que tiene correctamente el segundo valor màs grande de dificultad.

Esto conlleva que aunque la recomendaciòn hecha por Bell y Sullivan en el sentido de que un procedimiento no debe superar el

límite de longitud de 260, se puede exceder siempre que la dificultad sea pequeña.
También es interesante hacer notar la diferencia entre contenido intelectual y dificultad: si bien los procedimientos InitReservadas e InitEstándares a los que se ha hecho referencia no son difíciles, si están expresando bastante al definir el contenido -sin redundancias- de una tabla.

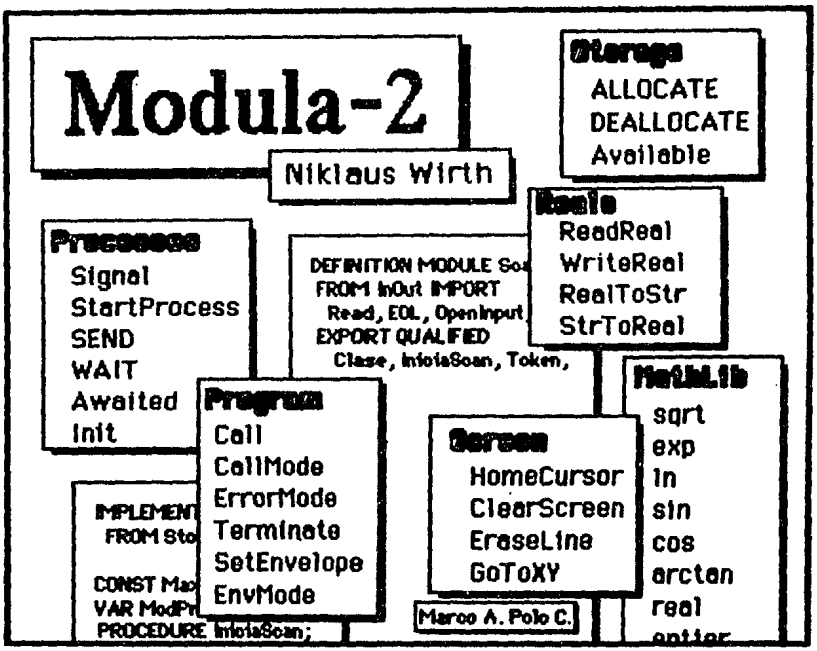
Las implicaciones de estos resultados son interesantes. En primer lugar proporcionan una perspectiva diferente y más formal de la manera en la que acostumbramos dimensionar un programa. El concepto de "líneas de código empleadas" resulta vago después de que al llevar a cabo la cuantificación de operadores y operandos podemos observar el valor de propiedades presentes en toda implementación de un algoritmo en un lenguaje de programación. Las hipótesis sobre las cuales se definen estas propiedades son al parecer consistentes y tenemos ahora mejores elementos para expresar las dimensiones de un programa.
Esto permite sugerir su aplicación en varios puntos. Por ejemplo, el estimador de dificultad junto con el de longitud nos permitirán mantener la complejidad de un programa dentro de límites razonables y también nos permite evaluar hasta que punto es efectiva la organización de un sistema: si entre los módulos que lo componen en uno sólo se concentra digamos el 90% de la dificultad total, entonces la división del programa no fue muy adecuada -por decir lo menos- sino que además tenemos un módulo muy propenso a convertirse en fuente de problemas.

Si aprovechamos la información de estos indicadores no sólo podemos conseguir programas más fáciles de mantener, sino también más confiables.

El estimador de esfuerzo de programación es una alternativa para valorar un programa. La estimación y asignación de recursos económicos no es tarea fácil en un proyecto de software y si mediante estos indicadores podemos evaluar una parte de los costos, el precio de desarrollar una herramienta que nos proporcione estos indicadores se justifica.

Por lo que respecta al ejercicio de haber desarrollado una herramienta de software empleando el diseño orientado a objetos y por supuesto Modula-2, puede afirmarse que bien vale la aparente "sobrecarga" de definir varios módulos de librería; esto se compensa con las ventajas que dá el poder organizar un proyecto con mayor sencillez y seguridad: una vez definidas las interfaces, cada integrante del grupo de implementación se hace cargo de los módulos de implementación que le corresponda desarrollar sin preocuparse por aquel trance conocido como "integración de sistemas" donde el grupo se reunía al término de su tarea para observar como "congeniaban" sus programas con los de los demás. Ahora, el compilador mismo asegura la adherencia a las interfaces establecidas.

Además la experiencia que se ganó al desarrollar este sistema fue invaluable. Estamos convencidos de que Modula-2, aunque no es perfecto, sí representa una de las mejores opciones cuando se trata de desarrollar sistemas con eficacia y elegancia, confiables y de fácil mantenimiento, lo que constituye finalmente una de las responsabilidades de todo ingeniero en computación. Un sistema imperfecto tiene poco valor. Sin la motivación para buscar la perfección y la búsqueda de soluciones óptimas como otra responsabilidad, estaremos perdiendo el tiempo.



DEBEMOS RECONOCER LA FUERTE E INNEGABLE INFLUENCIA QUE NUESTRO LENGUAJE EJERCE EN NUESTRA MANERA DE PENSAR, YA QUE DE HECHO DEFINE Y DELIMITA EL ESPACIO ABSTRACTO EN EL CUAL PODEMOS FORMULAR -DAR FORMA A- NUESTROS PENSAMIENTOS

N. WIRTH

INTRODUCCION

Con la invención del lenguaje de programación Pascal a finales de los años sesentas, Niklaus Wirth ofreció un instrumento eficaz y elegante para la enseñanza de la programación estructurada. La capacidad del lenguaje para describir claramente algoritmos y estructuras de datos le hizo alcanzar una popularidad sin precedentes, lo que al mismo tiempo provocó que surgieran una serie de "dialectos" (extensiones para resolver carencias en la versión original) que naturalmente hicieron imposible su estandarización, con la consiguiente falta de compatibilidad entre las múltiples implementaciones del mismo.

No obstante sus innegables ventajas, Pascal tenía ciertas carencias y defectos no por fallas en su diseño, sino por su utilización en aplicaciones originalmente no previstas. Sin embargo, en lugar de proponer extensiones a su lenguaje, Wirth comentó: "Si un lenguaje prueba ser sólo marginalmente adecuado para alguna aplicación que obviamente no fue prevista por su autor, debemos tener los ánimos para construir una nueva y verdaderamente adecuada herramienta, en vez de ponerle un parche a la existente".

Después de diseñar Pascal, Wirth realizó investigaciones sobre problemas de programación impuestos por multiprocesamiento y dispositivos de entrada/salida, para lo cual diseñó el lenguaje Modula que consistía en un subconjunto mínimo de Pascal, además de la estructura de módulo, una sintaxis mejorada y facilidades para multiprogramación y acceso a nivel de máquina.

Al regresar en 1977 del Centro de Investigación de Palo Alto de la compañía Xerox, donde pasó un año sabático experimentando con la computadora Alto y el lenguaje de programación Mesa, Wirth inició en el Instituto Federal de Tecnología de Zurich, un proyecto de investigación con el objetivo de diseñar íntegramente un sistema de cómputo (tanto hardware como software), para el cual se especificaron tres condiciones:

- todo el software debía escribirse en un sólo lenguaje
- el sistema operativo sería diseñado para un solo usuario y
- el computador debería tener un solo procesador, suficientemente poderoso para ejecutar programas y efectuar la operaciones necesarias para el despligue de imágenes y texto mapeado en memoria.

El lenguaje empleado fue Modula-2, resultado de cuidadosas deliberaciones de diseño, incluyendo los mejores elementos de sus predecesores: la estructura de las proposiciones y de los tipos de datos son un superconjunto de Pascal; la sintaxis, el concepto de módulo, el acceso a nivel de máquina y el multiprocesamiento son versiones mejoradas de las que se encuentran en Modula. Además, el diseño de Modula-2 resuelve los siguientes problemas y defectos de Pascal:

- Arreglos de tamaño fijo.
- Falta de variables propias ("OWN", como en Algol) que llevan al programador a declarar variables globales con un alcance mayor de lo adecuado.
- Falta de facilidades para compilación separada, que dificulta el desarrollo de programas grandes y hace imposible el uso de programas de biblioteca.

- El orden de declaración rígido impide agrupar objetos relacionados lògicamente.
- La expresiones booleanas no son evaluadas condicionalmente, lo que hace que se pierda claridad.
- La proposición CASE carece de un ELSE.
- Las facilidades de entrada/salida son muy limitadas.
- Falta de facilidades para acceso a nivel de màquina.

Con respecto al diseño de Modula-2 Wirth comenta:

"... se requería de un lenguaje igualmente adecuado para expresar algoritmos con un alto nivel de abstracción y para expresar operaciones que emplearan directamente facilidades de la màquina, igualmente adecuado para formular una base de datos como para escribir el controlador de una unidad de disco. Evidentemente, Pascal no era suficientemente poderoso, y yo no favorecía la salida común de embellecerlo con unas cuantas "extensiones deseables". Modula, un pequeño lenguaje que había diseñado años antes para experimentar con los conceptos de multiprogramación tampoco era suficiente, pero contenía una facilidad de particionar programas en mòdulos con interfases explícitamente especificadas. Esta era precisamente la facilidad necesitada para permitir la introducción de las llamadas facilidades de bajo nivel en el lenguaje de alto nivel, porque permitía encapsularlas y restringir su peligrosidad en partes claramente delineadas de un programa. De aquí, la elección para el nuevo lenguaje fue Pascal, aumentado por el mòdulo y otras cuantas facilidades, y regularizado por una sintaxis màs sistemàtica. Así nació Modula-2."

MODULOS

Un aspecto fundamental del lenguaje Pascal es su mètodo de estructuraciòn por bloques. El bloque ha probado su utilidad en la organizaciòn de programas: permite que los objetos que son declarados dentro de un bloque sean desconocidos fuera de èl; cuando una variable o un procedimiento se necesita en un solo lugar, se declara en el bloque local donde permanece oculto a los bloques exteriores.

El rango en el cual un objeto es conocido es denominado alcance (scope). Los bloques pueden estar anidados y el alcance del objeto es tambièn el bloque en el que està declarado, por tanto los alcances pueden anidarse de la misma forma y por ello decimos que el alcance de un objeto se extiende a partir del bloque en que se ha declarado hacia todos los bloques anidados. Otra forma de considerar los alcances es la regla de visibilidad: para un bloque dado, todos los objetos declarados en bloques anidados son invisibles, pero todos los objetos declarados en bloques exteriores son visibles.

La estructura de bloque no sòlo controla el alcance de un objeto, sino tambièn su existencia al tiempo de ejecuciòn. Los objetos declarados en un bloque son creados cuando se entra al bloque y eliminados cuando se sale del mismo. La regla de existencia implica que las variables locales no pueden mantener sus valores entre llamadas. La ùnica forma de lograrlo es declaràndolos en un bloque externo. De esta manera, la estructura de bloque une la existencia de una variable con su visibilidad.

En el diseño de programas grandes, el bloque es inadecuado por dos razones:

- Existe la necesidad de separar la visibilidad de la existencia. Debiera ser posible declarar variables que mantengan sus valores entre llamadas (como con la declaracion OWN de Algol), pero que sean visibles sólo en algunas partes del programa.
- Existe la necesidad de un mejor control de la visibilidad. Un procedimiento no debiera accesar cualquier objeto declarado fuera de èl, cuando sòlo necesita tener acceso a unos cuantos (si es el caso) de ellos.

Modula-2 ofrece la estructura de mòdulo para la soluciòn de estos problemas. Sintàcticamente un mòdulo se asemeja mucho a un procedimiento, pero tienen reglas distintas para la visibilidad y la existencia de los objetos que son declarados dentro de ellos. Por ejemplo, en el siguiente fragmento de programa:

```
PROCEDURE Externo;
 VAR x,w,z: INTEGER;

 MODULE Mod;
  IMPORT x;
  EXPORT a,P;
  VAR a,b,c: INTEGER;

  PROCEDURE P;
  BEGIN
    a := a + 1;
    x := a
  END P;

 END Mod;
...
END Externo;
```

Los objetos declarados dentro de Mod (a,b,c,P) existen en el mismo nivel de las variables x, w, z. En tèrminos de las variables esto significa que a, b y c son creadas al mismo tiempo

que x, w, z y existen mientras el procedimiento "Externo" está activo. Los objetos referidos en la declaración de importación del módulo Mod (esto es, la lista de identificadores que siguen a la palabra reservada IMPORT) son los únicos objetos declarados externamente que son visibles dentro de Mod; por tanto, Mod puede accesar la variable x, pero w y z no serán visibles. Los objetos referidos en la lista de exportación de Mod (aquellos identificadores que siguen a la palabra reservada EXPORT) son los únicos objetos declarados localmente que son visibles fuera de Mod; de esta manera, a y P son accesibles desde el procedimiento "Externo", pero b y c permanecen ocultos dentro de Mod.

Resumiendo, las reglas que controlan la visibilidad y existencia en módulos son:

- Los objetos declarados localmente, existen mientras el procedimiento que los contienen esté activo.
- Los objetos declarados localmente son visibles dentro del módulo. Si aparecen en la lista de exportación del módulo serán visibles fuera de él; y los objetos declarados fuera del módulo serán visibles sólo si están declarados en la lista de importación del módulo.

Como los procedimientos, los módulos pueden ser declarados en cualquier nivel; las reglas de visibilidad y existencia se mantienen para las declaraciones de módulos anidados.

Por ejemplo, supongamos que debemos escribir un programa de simulación en el que tenemos que implementar entre otras cosas, un generador de números aleatorios; el bosquejo del programa podría ser:

```
MODULE Simulador;

 ...

 MODULE NumAleatorios;
   IMPORT HoraDelDia;
   EXPORT Aleatorio;

 CONST Modulo = 3791;
       Delta  = 7227;

 VAR Semilla: INTEGER;

 PROCEDURE Aleatorio (): INTEGER;
 BEGIN
   Semilla := (Semilla + Delta) MOD Modulo;
   RETURN Semilla
 END Aleatorio;

 BEGIN
   Semilla := HoraDelDia
 END NumAleatorios;

 ...

BEGIN  (* Simulador *)
 ...
 WriteInt ( Aleatorio(),6);
 ...
END Simulador.
```

El procedimiento para generar números aleatorios emplea una semilla que debe mantener su valor entre llamadas para generar el siguiente número aleatorio y por tanto, se declara dentro del módulo "NumAleatorios" la variable "Semilla" que existirá mientras esté activo el módulo que la declara (en este caso el programa principal), pero que no podrá ser alterada fuera del módulo que la contiene; en resumen, todas las funciones necesarias para la generación de números aleatorios se encuentran en un sólo lugar. Para lograr esto en un lenguaje estructurado por bloques, tendríamos que declarar a "Semilla" como una variable global, inicializarla en el cuerpo principal del programa (o en algún procedimiento inicializador de variables

globales) y permitir que cualquier otra proposición (o procedimiento) tuviera acceso a tal variable, provocando con ello efectos laterales indeseables, por no mencionar la dificultad de depuración y mantenimiento del programa.

Este ejemplo presenta otra característica de los módulos: a diferencia del ejemplo anterior donde sólo incluía declaraciones, un módulo puede contener también proposiciones. El cuerpo del módulo es la parte opcional que contiene el conjunto de proposiciones del módulo, dedicadas principalmente a la inicialización de variables del mismo. Los cuerpos de los módulos son ejecutados automáticamente cuando el procedimiento que los incluye es invocado; si un procedimiento contiene varios módulos, los cuerpos son ejecutados en el orden en que aparecen dentro del procedimiento y al último es ejecutado el cuerpo (esto es, las proposiciones) del procedimiento. Por ejemplo (siguiente página):

```
PROCEDURE Ejem;

 MODULE M1;
   EXPORT x,y;

   VAR x: INTEGER;

   MODULE M2;
     EXPORT y;

     VAR y: INTEGER;
   BEGIN
     y := Ø;
   END M2;

 BEGIN
   x := -1;
 END M1;

 MODULE M3;
   EXPORT z;

   VAR z: INTEGER;
 BEGIN
   z := 1;
 END M3;

BEGIN    (* Ejem *)
   (* El cuerpo de M2 se invoca automàticamente aquí *)
   (* El cuerpo de M1 se invoca automàticamente aquí *)
   (* El cuerpo de M3 se invoca automàticamente aquí *)

   (* Comienza la ejecución del cuerpo de Ejem *)

   ...
END Ejem;
```

Las listas de importación y exportación deben estar inmediatamente después del encabezado del módulo. Ambas listas son opcionales: un módulo puede tener una lista de importación y puede carecer de una de exportación, o viceversa; puede haber varias listas de importación (esto es, varias ocurrencias de la palabra reservada IMPORT seguida por una lista de identificadores) pero sólo puede haber una lista de exportación. Además, las declaraciones de importación siempre deben preceder a las de exportación.

Cualquier tipo de objeto puede ser importado o exportado al referir su identificador en una lista de importación o exportación. Al exportar un registro se hacen visibles los identificadores de sus campos; al exportar un tipo enumerativo se hacen visibles todas sus constantes enumeradas; al exportar un módulo se hacen visibles todos sus identificadores exportados. Los procedimientos retienen la estructura de su lista de parámetros, pero no transfieren los identificadores del tipo de parámetro; para ello, los tipos deben exportarse por separado. Debe evitarse utilizar el mismo identificador para nombrar dos objetos declarados en el mismo nivel, excepto para los identificadores de los campos de un registro, los cuales son locales a su tipo y por tanto no puede existir duplicidad con otros identificadores.

Los identificadores obtenidos por importación o exportación son usados como identificadores declarados normalmente, esto es, como si no fuesen originarios de otro módulo. Sin embargo, también pueden ser referenciados como identificadores calificados: un identificador es calificado si está precedido por el nombre de su módulo y la sintaxis es igual a la utilizada para tener acceso a los campos de un registro en Pascal. Por ejemplo, un identificador denominado "disk" importado desde el módulo "Storage" puede ser referenciado indistintamente como "disk" o como "Storage.disk". (Sin embargo, un identificador calificado no se puede usar en listas de importación o exportación.)

Las referencias a identificadores exportados pueden ser calificadas o no-calificadas; pero si la palabra reservada EXPORT está seguida por QUALIFIED, las referencias a los identificadores fuera del módulo deberán ser calificadas. Esto se conoce como exportación calificada y evita la duplicidad de identificadores generada por la exportación del mismo identificador por otros módulos.

El preceder una lista de importación con la palabra reservada FROM seguida por el identificador de un módulo, tiene el efecto de "descalificar" los identificadores exportados por ese módulo, lo que se conoce como importación no-calificada. Ejemplo:

```
MODULE A;
 MODULE M1;
  EXPORT v1,v2;                    (* exportación no-calificada *)
  VAR v1,v2: INTEGER;
  ...
 END M1;

 MODULE M2;
  EXPORT QUALIFIED z1,z2;          (* exportación calificada *)
  VAR z1,z2: INTEGER;
  ...
 END M2;

 MODULE M3;
  IMPORT M1;
  EXPORT QUALIFIED t1,t2;
  VAR t1,t2: INTEGER;
 BEGIN
  t1 := v1;
  t2 := v2;
 END M3;

 MODULE Host;
  FROM M1 IMPORT v1,v2;            (* estas importaciones son *)
  FROM M2 IMPORT z1;               (* no-calificadas *)
  IMPORT M3;
 BEGIN
  z1 := v1 + v2;
  v1 := M3.t1 + M3.t2;
 END Host;

END A;
```

MODULOS COMPILADOS SEPARADAMENTE

La unidad bàsica de texto que acepta el compilador es llamada unidad de compilaciòn. Los programas en Modula-2 son construidos a partir de dos tipos de unidades de compilaciòn: mòdulos de programa y mòdulos de biblioteca.

Los mòdulos de programa estàn formados por una sola unidad que al compilarse constituyen programas ejecutables y pueden tener listas de importaciòn, pero no de exportaciòn. Una lista de importaciòn hace referencia a objetos definidos en mòdulos de biblioteca, especificamente en aquellos compilados por separado. La biblioteca (o librerìa de programas) es parte integral de Modula-2, ya que es el medio a nivel de sistema del cual se importan objetos (como rutinas del sistema operativo) a un programa. El siguiente listado es un ejemplo de un mòdulo de programa:

```
MODULE Saludo;

 FROM  InOut IMPORT WriteString;  (* WriteString se importa
                                     del mòdulo de librerìa InOut *)

BEGIN
   WriteString ('Hola !')
END Saludo.
```

Un mòdulo de programa puede importar mòdulos de biblioteca completos o solamente algunos objetos de tales mòdulos.

Los mòdulos de biblioteca se dividen siempre en dos unidades de compilaciòn: mòdulos de definiciòn y mòdulos de implementaciòn. Los primeros son similares a los mòdulos de programa, con la diferencia que el encabezado inicia con la palabra reservada DEFINITION; contienen declaraciones de los objetos que el mòdulo de biblioteca podrà exportar a otras unidades de compilaciòn,

esto es, contiene declaraciones de constantes, tipos, variables y los encabezados de los procedimientos que exporta. No contiene declaraciones de otros módulos, ni cuerpos de procedimiento o de módulo y puede incluir listas de importación.
En estos módulos se debe emplear únicamente exportación calificada. Un módulo de definición podría ser:

```
DEFINITION MODULE StringIO;
 FROM StringOps IMPORT String;
 EXPORT QUALIFIED ReadStr, WriteStr;
 PROCEDURE WriteStr ( S: String );
 PROCEDURE ReadStr ( VAR S: String );
END StringIO.
```

Los módulos de implementación contienen el código que implementa en sí el módulo de biblioteca; el encabezado principia con la palabra reservada IMPLEMENTATION y no pueden tener una lista de exportación. Todos los objetos declarados en un módulo de definición están automáticamente disponibles en el módulo de implementación correspondiente (lo cual implica que los módulos de definición se deben compilar antes que los de implementación). Sin embargo, los objetos que se hubiesen importado en la definición deberán ser importados nuevamente si fuesen requeridos. Por ejemplo, el módulo de implementación correspondiente al anterior podría ser:

```
IMPLEMENTATION MODULE StringIO;

 FROM CharIO IMPORT ReadCh, WriteCh;
 FROM StringOps IMPORT String, Length, MaxString;
 FROM ASCII IMPORT nul;

 PROCEDURE WriteStr ( S: String );
  VAR I: CARDINAL;
 BEGIN
   FOR I := 0 TO Length(S) - 1  DO WriteCh(S[I]) END
 END WriteStr;

 PROCEDURE ReadStr ( VAR S: String );
  VAR I: CARDINAL;
      ch: CHAR;
 BEGIN
   I := 0;
   REPEAT
     ReadCh(ch);
     S[I] := ch; INC(I)
   UNTIL (ch = nul) OR (I > MaxString)
 END ReadStr;

END StringIO;
```

Las ventajas que se obtienen al utilizar el esquema de módulo de definición separado del módulo de implementación se hacen evidentes en el desarrollo de programas no triviales. Considérese por ejemplo, el diseño y desarrollo de un sistema grande de software que involucre varios programadores. El primer paso en diseñar tal sistema es identificar los subsistemas principales y definir las interfases a través de las cuales se comunicarán dichos subsistemas; al término de esta fase, cada programador se hace responsable del desarrollo de uno o más de los subsistemas. Ahora considere el desarrollo del proyecto en términos de las facilidades de la compilación separada de Modula-2. Los subsistemas podrán ser conformados por una o más unidades de compilación. La definición y el mantenimiento de interfases consistentes es de importancia crucial para asegurar la comunicación libre de errores entre subsistemas (especialmente

cuando son desarrollados por varias personas). Sin embargo durante la fase de diseño, los subsistemas en sí no existen: sòlo se conocen sus interfases.

El concepto de interfase de un subsistema corresponde a la construcciòn del mòdulo de definiciòn; de esta manera, las interfases pueden ser definidos como un conjunto de mòdulos de definiciòn antes de iniciar el desarrollo de los subsistemas (esto es, antes del desarrollo de los mòdulos de implementaciòn). De esta manera la consistencia de las interfases es automàticamente verificada por el compilador. Otra ventaja de tener separados los mòdulos de definiciòn e implementaciòn es la habilidad de que dos o màs mòdulos de biblioteca puedan importar objetos entre sí; esto seria imposible si tales mòdulos fueran una sola unidad de compilaciòn.

Por otra parte, mediante los mòdulos de biblioteca es posible implementar dos clases de tipos de datos: transparentes y opacos. Las deçlaraciones de tipos son (por default) transparentes. El identificador de un tipo transparente està asociado con una estructura que define implícitamente las operaciones vàlidas sobre los objetos de ese tipo; en el caso de tipos estructurados, los componentes internos son accesibles. Por ejemplo, el tipo arreglo implica un tipo base conocido y define implícitamente la operaciòn de subíndicaciòn para accesar elementos individuales del arreglo.

Los tipos opacos son aquellos cuya estructura interna es conocida solamente en el mòdulo de implementaciòn. Los mòdulos que importen un tipo opaco pueden declarar y asignar objetos de ése tipo, pero no pueden ejecutar ninguna otra operaciòn mas que

aquellas previstas por los procedimientos exportados junto con el tipo opaco. En particular, los componentes internos de un tipo opaco son inaccesibles.

Los tipos opacos son declarados en un módulo de definición como identificadores carentes de una definición de tipo. Al igual que los procedimientos exportados, la declaración completa de un tipo opaco, está contenida en el módulo de implementación.

Un empleo clásico de los tipos opacos es la definición de archivos: éste tipo y sus operaciones (reset, rewrite, get, put) pueden ser expresadas en un módulo de biblioteca.

Modula-2 limita los tipos opacos a apuntadores y subrangos de tipos estándar.

BIBLIOTECA DE MODULOS

La biblioteca de módulos es una colección de módulos compilados por separado y es parte esencial de toda implementación de Modula-2. Contiene típicamente las siguientes clases de módulos:

* Módulos de "bajo nivel" que ofrecen acceso a recursos del sistema local.
* Módulos estándar de utilería que proveen de un ambiente de sistema consistente entre todas las implementaciones de Modula-2.
* Módulos de propósito general que permiten operaciones útiles a muchos programas.
* Módulos de propósito especial que forman parte de un programa.

La biblioteca es almacenada en uno o varios archivos que contienen el resultado de la compilación de los módulos de biblioteca. La compilación de un módulo de definición es llamado archivo de símbolos y la compilación de un módulo de implementación, archivo objeto.

La biblioteca es accesada tanto por el compilador como por el cargador de programas. El compilador lee los archivos de símbolos de la biblioteca cuando cuando compila programas que importan módulos de la misma. El cargador obtiene los archivos objeto de la biblioteca cuando se ejecutan programas que importan módulos de la bibioteca.

Los módulos son compilados separadamente, no independientemente (como ocurre en FORTRAN, por citar un ejemplo). La división de programas en módulos compilados por separado genera relaciones de dependencia entre módulos de biblioteca y sus clientes. Estas dependencias afectan la habilidad de recompilar un módulo independientemente del resto del sistema.

El ejemplo más simple de tal dependencia surge en la compilación de un módulo de biblioteca. El compilador debe referenciar el archivo de símbolos del módulo para poder compilar el módulo de implementación; por tanto, el módulo de definición debe compilarse primero. Una vez compilado el módulo de implementación, su archivo objeto está "ligado" al correspondiente archivo de símbolos, dado que el código objeto está basado en la información de direccionamiento para procedimientos y datos que obtuvo del archivo de símbolos.

En forma similar, los mòdulos clientes estàn "ligados" con los archivos de símbolos; los programas que importan un mòdulo de biblioteca deben asumir que la información de direccionamiento del archivo de símbolos sea un reflejo preciso del archivo objeto correspondiente.

Què ocurre si un mòdulo de definiciòn es modificado sin recompilar su mòdulo de implementaciòn ? La información de direccionamiento de procedimientos y variables en el archivo de símbolos puede ser distinta a la del archivo objeto y al tratar de ejecutar algún programa que haga referencia a èse mòdulo lo màs probable es que corriera sin control y degradara al sistema. Estos problemas pueden evitarse al seguir las siguientes reglas:

* Un mòdulo de definiciòn debe ser compilado antes que sus mòdulos clientes.

* Un mòdulo de implementaciòn puede ser compilado sin tener que recompilar ningún otro mòdulo del sistema.

* .Cuando los mòdulos de definiciòn e implementaciòn son recompilados, todos sus mòdulos clientes son invalidados y deben ser recompilados.

Modula-2 contiene mecanismos para hacer cumplir la ùltima regla: el compilador asigna un valor único al archivo de símbolos para cada mòdulo de definiciòn que compila; èstos valores son llamados llaves de mòdulo. Cuando una unidad de compilaciòn importa un mòdulo, el compilador graba la llave del mòdulo en el archivo objeto. Cuando un programa se ejecuta, el cargador verifica que las llaves del programa sean iguales a las llaves en los mòdulos importados; si alguna no concuerda, el cargador emite un mensaje de error y aborta la ejecuciòn del programa; de esta manera, el

sistema previene ejecuciones erróneas de programas causados por interfases inconsistentes de módulos.
La verificación de las llaves en los módulos es para el nivel del sistema lo que la verificación de tipos es para el nivel del compilador. Ambas son de igual importancia en Modula-2.

MODULOS ESTANDAR

Modula-2 no contiene procedimientos estándar para entrada/salida, manejo de memoria o sincronización de procesos; estas facilidades son provistas por módulos estándar almacenados en la biblioteca de módulos y deben estar presentes en toda implementación de Modula-2. De esta manera, al emplear solo módulos estándar, los programas en Modula-2 se vuelven portátiles entre todas las implementaciones.
Las ventajas de expresar rutinas comunmente usadas como módulos de biblioteca (en lugar de hacerlos parte del lenguaje) son: un compilador pequeño, un sistema para ejecución (run time system) más pequeño y la habilidad de definir métodos alternas cuando las facilidades estándar resulten ineficientes. Las desventajas son: la necesidad de importar y ligar explicitamente módulos de biblioteca y (ocasionalmente) una sintaxis menos flexible impuesta por tener que expresar rutinas estándar como módulos de biblioteca (en lugar de que fueran tratadas especialmente por el compilador). Aunque en varios casos esto pudiera parecer menos eficiente, el código generado no es más grande que el generado por un compilador que maneje las rutinas estándar (como el caso de READ y WRITE en Pascal).

Los mòdulos estàndar de Modula-2 son:

* InOut: para entrada/salida de tipos bàsicos.

* RealInOut: para entrada/salida de nûmeros reales.

* Texts: para el manejo de archivos de texto en la forma que lo hace Pascal.

* Files: implementa el acceso secuencial y aleatorio a archivos de datos.

* Terminal: proporciona las rutinas bàsicas de lectura de caracteres del teclado y desplegado de los mismos en pantalla.

* Reals: para entrada/salida y conversiòn (reales a string y viceversa) de nûmeros en punto flotante.

* MathLibØ: proporciona funciones matemàticas (seno, coseno, logaritmos, etc.).

* Storage: rutinas para el manejo dinàmico de memoria.

* Program: proporciona rutinas para realizar llamadas a subprogramas y tambièn ofrece facilidades para el manejo de excepciones, ademàs de un mecanismo que permite a mòdulos de biblioteca especificar procedimientos de inicializaciòn y terminaciòn que son automàticamente invocados en llamadas a subprogramas.

* Processes: proporciona las facilidades para la implementaciòn de procesos secuenciales.

* SYSTEM: ofrece las facilidades necesarias para la implementaciòn de corutinas; asî mismo provee tipos de datos para manipular objetos a nivel de màquina y procedimientos para determinar la direcciòn en memoria de variables y la representaciòn a nivel de màquina de variables y tipos.

Este módulo provèe los tipos WORD y ADDRESS. El tipo WORD es usado en rutinas de propósito general las cuales deben operar en argumentos de cualquier tipo. Los parámetros formales de este tipo son compatibles con cualquier tipo de parámetro real que ocupe una palabra de memoria. Sin embargo, fuera de listas de parámetros la única operación permitida en variables de tipo WORD es la asignación. Los parámetros que son arreglos "abiertos" de tipo base WORD son compatibles en tipo con todas las variables, en particular con registros y conjuntos.

El tipo ADDRESS es compatible con todos los apuntadores y también con el tipo CARDINAL, para permitir operaciones aritméticas sobre operandos de tipo ADDRESS. Su definición formal es:

```
TYPE ADDRESS = POINTER TO WORD;
```

La función ADR(x) regresa la dirección de memoria de la variable x siendo el resultado de tipo ADDRESS.

La función SIZE(x) regresa el número de unidades de almacenamiento (bytes o palabras, dependiendo de la implementación) que ocupa la variable x. SIZE no acepta arreglos abiertos como parámetros.

La función TSIZE(T) regresa el número de unidades de almacenamiento requeridas por un tipo T.

VARIABLES PROCEDIMIENTO

Modula-2 incluye un nuevo tipo de variable denominado tipo procedimiento y a las variables que se declaran de este tipo se les llama variables procedimiento. Estas son una generalizaciòn del concepto de paràmetros de procedimiento de Pascal; son anàlogas (pero no equivalentes) a variables de tipo apuntador y pueden interpretarse como "apuntadores a procedimientos".

Las ùnicas operaciones definidas para este tipo de variables son la asignaciòn y la invocaciòn.

Al llamar a una variable de procedimiento, se invoca el procedimiento que se le ha asignado. Las referencias a variables de procedimiento se diferencian de las llamadas a tales variables por la presencia de una lista de paràmetros (que puede ser vacìa). Por ejemplo, en las siguientes declaraciones:

```
TYPE Baile = (Disco,Rock,Break);
VAR G1, G2: PROCEDURE (Baile, Baile, Baile);
PROCEDURE Danza(i,j,k: Baile);

BEGIN
...
END Danza;
```

Una ocurrencia simple (esto es, sin lista de paràmetros) del identificador de procedimiento Danza o de las variables de procedimiento G1 y G2, denota al procedimiento como un objeto màs que como la llamada a un procedimiento. Por ejemplo:

```
G1 := Danza;
G2 := G1;
```

Por otra parte, una ocurrencia del identificador Danza o de G1 y G2 con una lista de parámetros denota la llamada al procedimiento:

```
Danza (Rock, Disco, Rock);
G1 (Rock, Disco, Rock);
G2 (Rock, Break, Disco);
```

Procedimientos de tipo función que carezcan de lista de parámetros deben ser declarados y llamados como se muestra para evitar ambiguedades:

```
PROCEDURE mx (): INTEGER;
...
I := mx();
```

Los procedimientos son asignados solamente si son declarados en el nivel más externo de una unidad de compilación. Los procedimientos estándar no son asignables.

DIFERENCIAS ENTRE MODULA-2 Y PASCAL

Vocabulario

El vocabulario comprende los identificadores, palabras reservadas, símbolos y comentarios.

Identificadores

En el reconocimiento de identificadores se diferencian mayúsculas y minúsculas, esto es, los identificadores N y n son distintos, como también lo serían InOut e inout.

A diferencia de Pascal, donde sólo los primeros ocho caracteres son significativos en la mayoría de sus implementaciones, todos los caracteres de los identificadores son válidos en Modula-2.

El guión de subrayado "_" no está permitido en los identificadores empleados en Modula-2.

Palabras y símbolos reservados

Las palabras reservadas siempre deben ser escritas con mayúsculas. Estas son:

AND	FOR	QUALIFIED
ARRAY	FROM	RECORD
BEGIN	IF	REPEAT
BY	IMPLEMENTATION	RETURN
CASE	IMPORT	SET
CONST	IN	THEN
DEFINITION	LOOP	TO
DIV	MOD	TYPE
DO	MODULE	UNTIL
ELSE	NOT	VAR
ELSIF	OF	WHILE
END	OR	WITH
EXIT	POINTER	
EXPORT	PROCEDURE	

Los símbolos reservados son:

+	-	*	/	:=	&
.	,	;	(	[	{
^	@	=	#	<	>
<>	<=	>=	..	:	)
]	}	|	"		

La barra vertical "|" sirve como delimitador en registros variantes y en proposiciones CASE. El símbolo "&" es sinónimo de de AND y el símbolo "#" es sinónimo de "<>".

Comentarios

Modula-2 sólo permite comentarios en la forma:

```
(* <aqui el comentario> *)
```

Las llaves "{" Y "}" se emplean para delimitar conjuntos constantes.

Los comentarios puede anidarse. Por ejemplo

```
(* este es el primer comentario
     (* y este es el segundo *)
aquí termina el primer comentario *)
```

Constantes

A diferencia de Pascal, permite expresiones constantes en declaraciones y tipos que dependen de otros valores constantes. Las expresiones constantes no pueden hacer referencia a variables o llamadas a funciones; en todo los demàs casos no hay restricción en su uso. Ejemplo:

```
CONST N = 4;
      MaxLength = 2*N;
      LastElement = MaxLength-1;
      SetExpression = {0,1,2} * {2..4};

TYPE  Elements = ARRAY [0..MaxLength-1] of INTEGER;
```

Constantes enteras

Las constantes enteras especifican valores constantes para tipos INTEGER y CARDINAL (enteros sin signo). Las constantes menores que cero son compatibles únicamente con el tipo INTEGER, y las constantes mayores o iguales a cero con ambos tipos.

Consulte el manual de usuario de su instalación particular para

conocer los rangos de validez de las constantes.

Las constantes enteras pueden ser especificadas en tres bases: decimal, hexadecimal y octal. Ejemplos de constantes decimales:

38	1985	29999
-8	0000	-32767

Las constantes hexadecimales siempre deben terminar con la letra H. Si una constante comienza con un digito alfabètico se le debe anteponer un cero. Ejemplos:

0H	3AEH	0BF33H

Las constantes octales siempre deben terminar con la letra B. Ejemplos:

0B	177B	7654B

Constantes reales

Estas constantes siempre deben llevar un punto decimal. Cuando se requiera elevar una constante a una potencia de 10, se usarà la letra E antes del exponente. Ejemplos:

1.0	2.71828	1.49E7

Constantes de tipo caracter

Estas constantes son compatibles con el tipo CHAR y pueden especificarse en dos formas: con valor caracter y con valor ordinal. Los primeros consisten en un caracter delimitado por apòstrofes o comillas. Las de valor ordinal consisten en un valor octal seguido de la letra C. Ejemplos:

'A'	'!'	"'"	15C

Cuerdas de caracteres (strings) constantes

La única diferencia sintáctica con Pascal es que se delimitan con comillas cuando contienen apòstrofes, y viceversa. Ejemplo:

" la barra vertical '|' es un simbolo reservado"

Las cuerdas deben tener más de un caracter para considerarse strings. Si sòlo tienen un caracter serán compatibles con el tipo CHAR.

Conjuntos constantes

Estas constantes se delimitan por corchetes "{" y "}" y sus elementos están limitados a expresiones constantes y subrangos. Se puede especificar explícitamente su tipo si se les precede con un identificador de tipo. Ejemplo:

```
PROCEDURE CheckChar;

   TYPE CharSet = SET OF CHAR;

   VAR  ch: CHAR;
        Valid: CharSet;

BEGIN
   ...
   IF ch IN CharSet{'a'..'c'} THEN
     INCL(Valid,ch);
   END;
   Valid := Valid + CharSet{'a','z'};
   ...

END CheckChar;
```

Los procedimientos INCL y EXCL para manejo de conjuntos se explican en la sección de Procedimientos estándar.

los conjuntos constantes que carezcan de un identificador de tipo serán por default del tipo BITSET, que se explica en la sección Conjuntos.

Tipos

Los nuevos tipos que ofrece Modula-2 son el tipo PROCEDURE y el CARDINAL (entero sin signo). A diferencia de Pascal, el tipo File se maneja a través de módulos estàndar.

Tipo procedimiento

A las variables de este tipo se les asigna procedimientos como valores y en sus declaraciones pueden contener listas de parámetros. Para que un procedimiento sea compatible con una variable de tipo procedimiento debe tener el mismo número y tipo de parámetros.

A las variables de este tipo no se les puede asignar procedimientos estándar o procedimientos locales a otro procedimiento.

Modula-2 incluye el tipo estándar PROC que denota un procedimiento sin parámetros. Ejemplos de tipo procedimiento:

```
TYPE ProcType = PROCEDURE (CARDINAL, VAR INTEGER, CHAR);
     FuncType = PROCEDURE (): CARDINAL;
     Shortype = PROC;
```

Tipo cardinal

El tipo CARDINAL es utilizado únicamente para operaciones enteras mayores o iguales a cero. Todas las operaciones para enteros pueden emplearse para cardinales.

Las variables de tipo cardinal son asignables a variables de tipo entero y viceversa; sin embargo, variables enteras y cardinales no se pueden mezclar en expresiones.

Tipo caracter

El tipo CHAR es el mismo que en Pascal. Se emplea el conjunto ASCII de caracteres.

Tipo subrango

La única diferencia sintáctica con Pascal es el empleo de parèntesis cuadrados para delimitar subrangos. Ejemplos:

```
TYPE
    Dias=(Lunes,Martes,Miercoles,Jueves,Viernes,Sabado,Domingo);
    Alfabeto = ['A'..'Z'];
    Semana = [Lunes..Domingo];
    Laborables = [Lunes..Viernes];
    Descansos = [Sabado..Lunes];
```

Tipo arreglo

Las declaraciones de arreglos tienen una diferencia: cuando el identificador de un subrango es empleado para especificar los limites del arreglo, se deben omitir los parèntesis cuadrados que normalmente lleva. Ejemplo. Considerando las declaraciones que hicimos en el ejemplo anterior:

```
VAR Alfabetico: ARRAY Alfabeto OF CHAR;
    Calendario: ARRAY [0..51] OF Semana;
```

Tipo registro

En Modula-2, los registros pueden tener una o màs partes variantes y cada una de ellas debe terminar con END; los selectores (o etiquetas) de los CASE pueden contener expresiones constantes y subrangos, y pueden tener un ELSE. Las declaraciones variantes deben separarse con una barra vertical '|' (que no debe aparecer antes del ELSE). Ejemplo:

```
TYPE Fecha = RECORD
               Dia: [1..31];
               Mes: [1..12];
               Año: [0..99]
             END;

     Varii = RECORD
               CASE selec1: CARDINAL OF
                 0..9: x,y: INTEGER |
                 11:    c:    CHAR
               ELSE i,j: CARDINAL
               END;
               Nacimiento: Fecha;
               Talla: [28..44];
               CASE selec2: BOOLEAN OF
                 TRUE: s: REAL    |
                 FALSE: r: INTEGER
               END
             END;

     Truco = RECORD
               CASE BOOLEAN OF
                 TRUE: I: INTEGER |
                 FALSE:C: CARDINAL
               END
             END;
```

Tipo conjunto

Como se mencionó antes, los conjuntos son delimitados por corchetes (en lugar de parèntesis cuadrados) y son tipificados explícitamente.

Modula-2 define el tipo estàndar BITSET como un conjunto que cabe en una palabra de memoria. Las operaciones de conjuntos son màs eficientes con bitsets que con conjuntos màs grandes. La definición formal de BITSET es:

```
TYPE BITSET = SET OF [0..WordSize-1];
```

Tipo apuntador

Las declaraciones de tipo apuntador ahora emplean las palabras reservadas "POINTER TO" en lugar de "^" y ya no estàn restringidas a identificadores de tipos: cualquier tipo o estructura puede ser utilizada. Ejemplo:

```
TYPE P1 = POINTER TO INTEGER;
     PR = POINTER TO RECORD
                       a,b,c: CHAR;
                       i,j,k: CARDINAL
                     END;
```

Expresiones

Variables-procedimiento

Las variables-procedimiento pueden ser refenciadas de dos maneras en una expresiòn: a) los identificadores de funciòn que lleven una lista de paràmetros (que puede ser vacia) denotan la invocaciòn de èse procedimiento; b) aquellos que carezcan de lista de paràmetros se refieren al procedimiento en si mismo, necesario para asignar valores a variables de tipo procedimiento. Ejemplo:

```
MODULE PF;

VAR i: INTEGER;
    pa: PROCEDURE (INTEGER): INTEGER;
    pb: PROCEDURE (): INTEGER;

PROCEDURE Func1 ( arg: INTEGER ): INTEGER;
BEGIN
  RETURN arg DIV 2
END Func1;

PROCEDURE Func2 (): INTEGER;
BEGIN
  RETURN 76
END Func2;

BEGIN
  i := Func1 (7);     (* invocaciòn de Func1              *)
  pa := Func1;        (* asigna Func1 a pa                *)
  i := pa (7);        (* llama a Func1 a través de pa     *)
  i := Func2 ();      (* llama a Func2                    *)
  pb := Func2;        (* asigna Func2 a pb                *)
  i := pb ();         (* llama a Func2 a través de pb     *)
  ...

END PF.
```

Operadores

Las operaciones '+', '-', '*', DIV y MOD se aplican a cardinales, enteros y subrangos de ellos. El '/' se aplica a la división real. Observe que MOD no está definido para argumentos negativos. Los operadores lógicos AND y OR son evaluados condicionalmente, esto es, "cortocircuitan" la evaluación de expresiones si el resultado de la expresión puede ser determinado por el valor del argumento izquierdo.

p AND q es equivalente a "IF p THEN q ELSE FALSE"

p OR q es equivalente a "IF p THEN TRUE ELSE q"

Esta definiciones de AND y OR permiten soluciones más simples y eficientes a muchos problemas de programación. En el siguiente ejemplo, la evaluación condicional previene la referencia a un apuntador inválido:

```
WHILE (Event # NIL) & (Event^.Time < Now) DO
  Event := Event^.Next
END;
```

En adición a los operadores para unión, diferéncia, intersección e inclusión de conjuntos, se ofrece el operador '/', que se define como la diferencia simétrica de conjuntos. Este operador efectúa una operación OR-exclusivo bit por bit.

Mezcla de tipos en expresiones

Los operandos de los tipos INTEGER, CARDINAL y REAL no pueden ser mezclados libremente en expresiones; a menos que sean usadas funciones de tranferencia de tipos, las expresiones deben consistir enteramente de enteros, cardinales y reales, respectivamente. Para conversiones de real a entero existe la función TRUNC(x). Para conversiones de entero a real se utiliza la función FLOAT(i). Modula-2 no ofrece la función ROUND.

Los operandos de tipo WORD no son compatibles con otros tipos de operando.

Los operandos de tipo ADDRESS son compatibles con cardinales y apuntadores en expresiones.

Finalmente, los operandos de conjuntos deben ser del mismo tipo para ser compatibles.

Proposiciones

La diferencia principal estriba en la reorganizaciòn de las proposiciones estructuradas mediante secuencias de proposiciones, en lugar de proposiciones compuestas.

En Modula-2 todas las proposiciones estructuradas terminan con un símbolo de terminaciòn explícito: UNTIL para la proposiciòn REPEAT, y END para las demàs. La proposiciòn compuesta de Pascal (una o màs instrucciones delimitadas por BEGIN y END) no existe en Modula-2; ha sido reemplazada por el concepto de secuencias de proposiciones, que son series de proposiciones separadas por punto y coma; las secuencias estàn delimitadas por la estructura que las contiene, màs que por símbolos delimitadores.

La instrucciòn GOTO no existe en Modula-2. En su lugar estàn las proposiciones LOOP/EXIT y RETURN, y el procedimiento estàndar HALT.

Proposiciòn de asignaciòn

La regla fundamental para la compatibilidad de asignaciòn (esto es, que los operandos sean asignables uno al otro) es que deben ser del mismo tipo.

Los operandos tambièn son compatibles si uno es declarado como subrango del otro, o si ambos operandos son declarados subrangos del mismo tipo.

El tipo WORD solo es compatible con sí mismo. Operandos de tipo ADDRESS son compatibles con apuntadores y variables del tipo CARDINAL. Los tipos INTEGER y CARDINAL también son compatibles. Las cuerdas constantes de caracteres son compatibles con arreglos de caracteres de longitud mayor; en este caso la asignación pone un caracter nulo (ØC) en el siguiente elemento del arreglo.

Llamadas a procedimientos

Las llamadas a procedimientos consisten de un identificador al que puede seguir una lista de paràmetros entre parèntesis. En Modula-2, tal identificador puede ser el de un procedimiento o el de una variable de tipo procedimiento. Vea la sección Variables procedimiento para mas detalles.

Proposición WHILE

La única diferencia consiste en que en Modula-2 requiere el símbolo de terminación END. Ejemplo:

```
WHILE A[J] <= Ø DO
  A[J] := A[J] + A[J-1];
  INC (J)
END;
```

Proposición IF

Su sintaxis requiere el símbolo de terminación END, y ofrece el símbolo ELSIF que permite expresar condiciones en cascada en una sola proposición. Las formas bàsicas son:

```
IF <condición> THEN <secuencia de proposiciones> END;

IF <condición>
THEN
  <secuencia de proposiciones>
ELSE
  <secuencia de proposiciones>
END;
```

Las condiciones en cascada tienen la forma:

```
IF <condición 1> THEN
  <secuencia de proposiciones>
ELSIF <condición 2> THEN
  <secuencia de proposiciones>
...
ELSIF <condición n> THEN
  <secuencia de operadores>
END;

IF <condición 1> THEN
  <secuencia de proposiciones>
ELSIF <condición 2> THEN
  <secuencia de proposiciones>
...
ELSE <secuencia de proposiciones> END;
```

Proposición FOR

Requiere el símbolo de terminación END. El valor del incremento por default es 1, aunque puede especificarse cualquier otro mediante el símbolo BY. En Modula-2 no existe el símbolo DOWNTO. La variable de control no puede ser parte de una variable estructurada, ni ser importada, ni ser un parámetro. Las formas de esta proposición son:

```
FOR <variable de control> := <expresión> DO
  <secuencia de proposiciones>
END;

FOR <variable de control> := <expresión> BY <incremento> DO
  <secuencia de proposiciones>
END;
```

Proposición WITH

Requiere el símbolo de terminación END y (a diferencia de Pascal) sólo acepta la referencia a un identificador.

Proposición CASE

En Modula-2 se permite en las etiquetas de CASE subrangos y expresiones constantes. Si el valor de prueba no iguala ninguna etiqueta, se seleccionará la secuencia de proposiciones que siga al símbolo ELSE (si éste fuera el caso y no hubiera ELSE ocurrirá

un error de ejecución).

Las secuencias de proposiciones se separan por una barra vertical "|", la cual no debe aparecer antes del ELSE

Proposición LOOP/EXIT

Esta proposición especifica la ejecución cíclica de una secuencia de proposiciónes. En particular, aborda dos situaciones de programación que no son bien manejadas en Pascal. Las proposiciones LOOP sin EXIT expresan la repetición infinita de un grupo de proposiciones; los LOOP con EXIT son empleadas para expresar secuencias repetitivas de expresiones con requerimientos de terminación especial, esto es, uno o varios puntos de salida enmedio de la secuencia de proposiciones. Ejemplos:

```
LOOP
  GeneraRuido;
  JuegaOtelo;
  WriteString (" Ganè !!!")
END;

LOOP
  WITH Nodo^ DO
    IF Nombre = ID THEN EXIT
    ELSIF Nombre < ID THEN EXIT
      IF LigaI = NIL THEN EXIT ELSE Nodo := LigaI END
    ELSE
      IF LigaD = NIL THEN EXIT ELSE Nodo := LigaD END
    END
  END
END;
```

Proposición RETURN

La proposición RETURN tiene dos formas y sirve para dos propòsitos. En procedimientos y mòdulos sirve para terminar el procedimiento o mòdulo que lo contiene (aunque èstos lo tienen implícitamente al final). En procedimientos-función, el símbolo RETURN siempre está seguido de una expresión, la cual es el valor que regresa la función A y hace que el procedimiento termine.

Procedimientos y funciones

Los únicos cambios a procedimientos y funciones son la nueva sintaxis para la declaración de funciones y la adición de arreglos de dimensión variable como parámetros.

En Modula-2 las funciones son llamadas procedimientos función; su encabezado es similar al de un procedimiento, con la diferencia de que se denota el identificador del tipo del resultado que la función regresa después de la (posiblemente vacía) lista de parámetros. Dentro de un procedimiento función, el resultado que regresa siempre está precedido del símbolo RETURN. Ejemplo:

```
PROCEDURE AreaCirc ( r: REAL ): REAL;
CONST PI = 3.1415926;
BEGIN
  RETURN r * r * PI
END AreaCirc;
```

En la declaración de procedimientos (al igual que en módulos) se debe repetir el identificador del procedimiento (o módulo) después del símbolo de terminación END.

Modula-2 permite parámetros formales de la forma:

```
ARRAY OF T
```

donde T es cualquier tipo base. Note que los límites del arreglo se omiten (o sea, el arreglo está abierto) y por tanto es compatible con todos los arreglos (unidimensionales) que tengan como tipo base a T. Mediante la función HIGH(a) se obtiene el índice del último elemento del arreglo a; se asume que el límite inferior de los arreglos abiertos es cero.

Si un parámetro formal tiene la forma ARRAY OF WORD, el correspondiente parámetro real puede ser de cualquier tipo.

Procedimientos estàndar

Los procedimientos estàndar son importados automàticamente en todos los mòdulos. (Esto implica que los procedimientos estàndar pueden ser redefinidos dentro de procedimientos, pero no dentro de mòdulos.) Estos son:

ABS(x)	valor absoluto; x puede ser entero o real.
ODD(x)	regresa el valor booleano TRUE si la expresiòn x (cardinal o entera) es non.
ORD(x)	valor ordinal de x, que puede ser enumerativo, caracter, cardinal o entero. El resultado es de tipo cardinal.
CHR(x)	regresa el caracter cuyo ordinal es x (se emplea el còdigo ASCII; x es cardinal.
VAL(T,x)	regresa el valor que corresponde al ordinal x del tipo T; x es cardinal, T es enumerativo, caracter, cardinal o entero.
TRUNC(x)	regresa la parte entera del nùmero real x.
FLOAT(x)	regresa el valor real del cardinal x.

Las funciones PRED y SUCC de Pascal han sido reemplazadas por los procedimientos INC y DEC, que tienen dos formas: INC(x) y DEC(x) reemplazan x con su sucesor o predecesor inmediato respectivamente. INC(x,n) y DEC(x,n) reemplazan x con su n-èsimo sucesor o predecesor, respectivamente; n puede ser cualquier expresiòn compatible con x.

INC y DEC aceptan caracteres, enteros y direcciones (ADDRESS) asi como subrangos y enumeraciones.

Modula-2 provèe los procedimientos INCL y EXCL para manipulaciòn de conjuntos. INCL(s,e) añade el elemento e al conjunto s;

EXCL(s,e) remueve el elemento e de s. Note que "e" puede ser cualquier expresiòn compatible con el tipo base de "s".

Los procedimientos NEW(p) y DISPOSE(p) funcionan de la misma manera que en Pascal, si bien estos son traducidos en llamadas a los procedimientos ALLOCATE y DEALLOCATE del mòdulo estàndar Storage.

El procedimiento HALT hace terminar la ejecuciòn de un programa.

La funciòn HIGH(a) regresa un cardinal que indica el límite superior del arreglo unidimensional a. Este procedimiento es empleado normalmente para determinar el último elemento de un arreglo abierto que fue pasado como argumento a un procedimiento.

Finalmente, el procedimiento-funciòn CAP(c) regresa el equivalente en mayúscula del caracter en minúscula c.

Orden de declaraciones

El lenguaje Pascal impone un orden estricto de la declaraciòn de objetos: primero las etiquetas, seguidas de las constantes, tipos, variables, procedimientos, funciones y finalmente el cuerpo del programa. Modula-2 elimina esta restricciòn: las declaraciones pueden aparecer en cualquier orden. De esta manera, aquellos programas que contengan un gran número de declaraciones son màs faciles de comprender al tener agrupadas declaraciones relacionadas lògicamente.

Cada declaraciòn de mòdulo y de procedimiento debe terminar con su identificador, lo que tambièn contribuye a la legibilidad de programas grandes.

B I B L I O G R A F I A

CAPITULO 1

1.1 Software engineering with Modula-2 and Ada
Wiener - Sincovec
John Wiley 1984

1.2 Software engineering: a practitioner's approach
Pressman
McGraw Hill 1982

1.3 Elements of Software Science
Halstead
Elsevier 1977

CAPITULO 2

2.1 An experimental study of software metrics for real-time software
Jensen - Vaivaran
IEEE Proceedings on Software Engineering
vol. 11, no. 2, febrero 1985

2.2 Software engineering with Modula-2 and Ada
Wiener - Sincovec
John Wiley 1984

2.3 Software engineering: a practitioner's approach
Pressman
McGraw Hill 1982

2.4 Elements of Software Science
Halstead
Elsevier 1977

2.5 Research directions in software technology
Wegner (editor)
MIT Press 1980

CAPITULO 3

3.1 Software engineering with Modula-2 and Ada
Wiener - Sincovec
John Wiley 1984

3.2 Programming in Modula-2
Niklaus Wirth
Springer-Verlag 1982

3.3 The science of programming
Gries
Springer-Verlag 1981

APENDICE

A.1 Modula-2 user's manual
Richard Gleaves
Volition Systems
Release 0.3 26 de agosto de 1983