Anales PANEL'81/12 JAIIO
Sociedad Argentina de informática
e Investigación Operativa. Buenos Aires, 1981

# CONSIDERACIONES INICIALES PARA EL DISEÑO DE UN NUEVO LENGUAJE DE PROGRAMACION

**Javier De La Cuba Bravo**
Central de Procesamiento de Datos
Ministerio de Marina
Av. La Marina Cuadra 36 s/n., Lima PERU.

El desarrollo de software en general, constituye una desafiante disciplina de las ciencias de computación que, desafortunadamente y pese a su gran trascendencia, es normalmente considerada como de poca importancia, permitiendo este hecho, que el desarrollo de proyectos de software se realize de manera bastante especial y por lo general, sin tener en cuenta experiencias anteriores ni recientes avances teóricos de la disciplina.

De igual manera, existe poca cooperación entre distintos grupos de desarrollo de software, para la realización del esfuerzo de diseño funcional del proyecto, dejándose esta importante etapa prácticamente en la imaginación, experiencia y "buen juicio" del equipo de programadores.

Reconsideremos por un momento los pasos básicos necesarios para lograr un desarrollo satisfactorio de software en general. Además de simplemente RESOLVER EL PROBLEMA (cualquiera sea la definición de este objetivo), se deben también considerar los siguientes aspectos:

1) Costos mínimos de desarrollo - vale decir, la mano de obra necesaria, tiempo de computador y otros recursos necesarios para detallar y escribir todo el software.

2) Costos mínimos de pruebas - tiempo de computación, esfuerzo, y otros recursos necesarios para probar "satisfactoriamente" dicho software.

3) Portabilidad - la habilidad para poder variar las condiciones ambientales de ejecución de dicho software, sin que este sufra grandes variaciones, o se vea afectado en su ejecución.

4) Mantenimiento - la habilidad para poder sostener dicho software en forma operativa durante un tiempo razonable mediante pequeñas modificaciones y sin encontrar mayores problemas.

5) Confiabilidad - La ausencia de errores graves y/o leves

6) Eficiencia - el permitir un uso eficiente de CPU, memoria, canales y otros recursos del sistema.

Es interesante anotar que es quizás la última característica, eficiencia, la que más atrae la atención del programador común.

Asumiendo que los objetivos son bien conocidos y aceptables para todo proyecto de desarrollo de software, es también significativo que no se haya desarrollado hasta la fecha, algún método de diseño que garantice la obtención final de todos estos requerimientos "simples".

Se puede observar asimismo, la importancia tan grande que representa la presencia de los lenguajes de programación de alto nivel en el proceso de desarrollo de software. Estos constituyen en sus diversas formas, importantes herramientas usadas en la proyección final del diseño de una aplicación, es decir, constituir un producto útil.

Pese a que el desarrollo de dichos lenguajes de programación ha sido bastante acelerado, y existe en la actualidad una gran diversidad y variedad de estos, se puede afirmar que quizás la única propiedad común y fundamental que se ha propagado a lo largo de toda la historia de computación, es la de que todos los lenguajes de programación permiten de alguna manera la representación de procedimientos y conceptos de tal forma que sean entendibles tanto por seres humanos como por un computador, luego de ser "traducidos" a un conjunto de instrucciones de máquina que al ser ejecutadas reflejen las intenciones de su creador.

Aunque hubiese sido quizás lo mas natural, que cada nueva generación de lenguajes de programación aprendiese de las anteriores de manera tal que se adaptasen a su correspondiente problemática de desarrollo, pareciera ser que esto no es lo que ha sucedido en la realidad. Aparentemente, no se tiene en cuenta el desarrollo de disciplinas tales como Ingeniería de Software, que han experimentado un acelerado y espectacular desarrollo en los últimos tiempos, pese a que como dice el Profesor Richard W. Hamming:

> "...existe una gran diferencia entre la Ingeniería como ciencia y la Ingeniería de Software, en que mien-

tras la primera se basa principalmente en reglas del mundo real, la segunda se desenvuelve siempre dentro de un ambiente producido por el hombre (MUNDO SINTETICO), el mismo que por definición, se halla sujeto a constantes variaciones..."

Esta misma problemática de desarrollo de software, permite considerar como factible el hecho de que se puede lograr un lenguaje de programación que considere como natural y propio de su estructura, el desarrollo de software utilizando para ello las mejores y más reconocidas técnicas de diseño y desarrollo, así como que permita la consecución final de los objetivos planteados anteriormente.

De igual manera, dicho lenguaje debiera constituir una imagen del proceso total del diseño utilizado, de tal forma que se convierta en la base común de los usuarios, analistas y programadores, así como que permita una reducción significativa de la dificultad existente actualmente tanto en la etapa de chequeo del producto, como de los problemas de inter-comunicación típicos de un grupo de desarrollo de software.

Es por consiguiente el propósito de este artículo, el presentar las consideraciones iniciales mínimas para el diseño de un lenguaje de programación que posea las características antes descritas, y que sirva de base para su desarrollo futuro; dicho lenguaje deberá tener una base sintáctica mínima que le permita una fácil y natural adaptación al ambiente en que se use, así como representar y constituir una real imagen de la aplicación desarrollada; esto implica necesariamente que dicho lenguaje deberá permitir la descripción funcional de un problema de tal manera, que facilite el desarrollo de software, a través de la utilización natural de los conceptos y principios de la Ingeniería de Software y abstracción funcional.

## ANTECEDENTES Y BASE COMUN DE DISEÑO

Dado que no es del todo simple el poder separar de una manera completa los alcances, beneficios, desventajas e implicancias del binomio hardware-software en lo que a importancia y trascendencia se refiere, es necesario sin embargo, el establecer de una manera formal el rol histórico que dicho binomio ha jugado en el desarrollo de las ciencias de computación en general.

Históricamente, y sólo con el propósito de enfocar aquellos aspectos que tienen trascendencia en el desarrollo de este artículo, es necesario destacar la gran importancia que han tenido las características de hardware en general, en el desarrollo del software correspondiente, asi como señalar que han constituído un factor muy importante en la evolución de los distintos lenguajes de programación, al igual que en su tendencia futura.

Estos aspectos van desde el desarrollo propio de nuevos

elementos electrónicos y técnicas de miniaturización de circuitos, hasta el giro económico que éstas mismas han ocasionado. A nadie debe escapar que la tendencia actual de crecimiento de costos del software, ya sea por diseño o mantenimiento, continuará en una vertiginosa escalada, mientras que el costo del hardware, se verá cada vez más y más reducido. Esto ha ocasionado que el deseo de disminuir los costos del software, haya sido dirigido hacia el desarrollo de nuevos métodos y técnicas que faciliten su desarrollo y/o mantenimiento.

Ya que el objetivo es el conseguir que el software realice la tarea para la cual fue concebido, es necesario señalar entonces, que este objetivo implicará que dicha tarea sea realizable sin errores graves y por consiguiente, con una confiabilidad casi total.

Myers (2) nos presenta dos definiciones interesantes:

- Error de software:

  "Un error de software se halla presente cuando éste no hace lo que el usuario espera razonablemente que haga. Una falla de software es una ocurrencia de un error de software".

- Confiabilidad de software:

  "Es la probabilidad de que el software ejecutará durante un período particular de tiempo sin fallar, multiplicado por el factor del costo al usuario, ocasionado por cada falla (error de software) encontrada".

Del estudio realizado sobre fallas de software y de la gran complejidad que significa el chequeo consistente del funcionamiento de programas y sistemas, se puede inferir que tales fallas no son normalmente causadas por "desgaste", sino por errores en el diseño básico inicial, en la programación en sí o en posteriores modificaciones debidas a la obsolescencia del producto, o mejor dicho, a su incapacidad para adaptarse naturalmente a las necesidades impuestas por un ambiente dinámicamente cambiante.

Esta situación se presenta de manera gráfica en la Figura N° 1 en la que se puede apreciar claramente, cual es el comportamiento típico de un sistema a través del tiempo, en relación al número de fallas de software.

Con el propósito de reducir estos inconvenientes asi como conseguir un software generalmente más barato de diseñar y mentener, se han desarrollado una serie de técnicas nuevas tales como:

- Programación estructurada
- Diseño de arriba hacia abajo (TOP-DOWN)

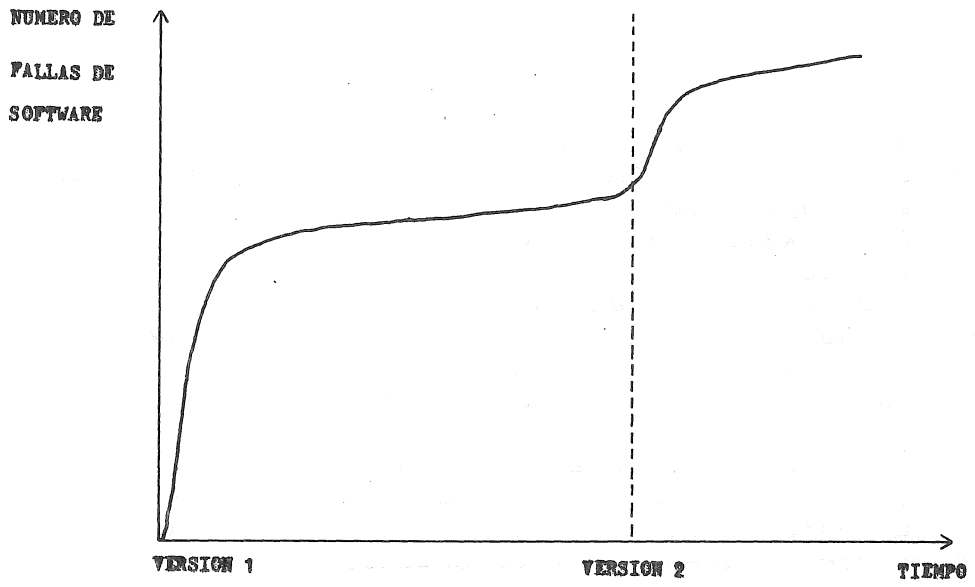


FIGURA Nº 1

- Diseño modular
- Desarrollo jerárquico
- etc.

Todas ellas tienen en común el hecho de qué concentran su atención en la determinación de límites acerca de la manera en la que se debe diseñar y programar software mediante la determinación de estructuras adecuadas; esto implica que en general, se debe diseñar y programar utilizando aspectos y métodos disponibles en el lenguaje escogido. Al realizar ésto, es posible que la integridad del diseño sea perdida, por lo que se añadiría - cierto grado de complejidad al problema.

Para obviar dicha complejidad, se ha desarrollado una técnica muy poderosa que consiste en utilizar una abstracción del problema; ésto implica que en lugar de tratar directamente con el problema, lo que se hace es tratar con un modelo idealizado del mismo, que sí sea manejable. Dicha abstracción puede ser realizada al nivel de diseño, o en los niveles de representación de datos o de las estructuras de control necesarias.

El inconveniente ocurre cuando, debido a las limitaciones naturales del lenguaje escogido para poder representar convenientemente dicha abstracción, se producen efectos laterales desconocidos y que pueden influenciar indirectamente el comportamiento de un programa, ocasionando la presencia de fallas que son muy difíciles de detectar.

Estas fallas sin embargo, no son producto del diseño original, sino mas bien, un resultado directo de la incapacidad del lenguaje de programación de permitir una transformación directa

del diseño, en programas que funcionen adecuadamente; el problema básico se halla entonces, en que en lugar de integrar los lenguajes en los nuevos métodos de diseño, lo que se ha hecho es "ajustar" estos lenguajes a dichas técnicas de diseño (i.e. programación estructurada usando FORTRAN!!)

Dos aspectos más acerca de abstracción. No se desea eliminar con la abstracción, todas aquellas construcciones oscuras y confusas que resultan inadecuadas para el programador por trabajar a un nivel muy cercano al del hardware, sino por el contrario, se trata de hacer que no "aparezcan" el nivel de programa fuente, que sí es el nivel de trabajo del Programador. Por otro lado, tal como señala W.A. Wulf (3), se debe notar que la noción de abstracción puede aparecer en un lenguaje de programación, en cualquiera de dos maneras: implícita o explícitamente. Como abstracciones implícitas, se entiende todas aquellas impuestas por el lenguaje, tales como: estructuras de datos predefinidas (variables simples, registros, arreglos, etc.), estructuras de control predefinidas (DO WHILE, DO UNTIL, FOR, etc.) y estrategias de definición de la realización de dichas estructuras de datos y de control (código generado, administración, de memoria, etc). Como abstracciones explícitas, se entiende todas aquellas producidas e introducidas por el Programador, y que son soportadas por el lenguaje en un nivel casi metalingüístico, en el sentido de que provee de los mecanismos necesarios para la definición de dichas abstracciones (PROCEDURE, FUNCTION, nuevas definiciones de tipos de datos, etc.).

Pareciéra ser sin embargo, que el problema se halla precisamente en que los lenguajes contienen demasiadas abstracciones realizadas de manera implícita, lo que motiva que gran parte del esfuerzo de diseño se vea dirigido al "encapsulamiento" del problema en el esquema más o menos rígido que es planteado por el lenguaje; ésto es especialmente cierto si se trata de utilizar construcciones, o mejor dicho abstracciones que se hallen por debajo del ya "alto nivel" proporcionado por el lenguaje(i.e. trabajar con variables tipo puntero en COBOL).

Lo contrario es también cierto, en el sentido de que normalmente no se cuenta con suficientes mecanismos como para poder definir y utilizar abstracciones del tipo explícito, lo que realmente establece el límite superior del ambiente de desarrollo de software, al utilizar dichos lenguajes.

Un aspecto muy importante de la abstracción, la constituye el uso de datos abstractos. Bárbara Liskov y Stephen Zilles (4) nos proveen de una acertada definición de data abstracta:

"Un grupo de funciones u operaciones relacionadas que actúan sobre una clase particular de objetos, con la restricción de que el comportamiento de los objetos puede ser observado sólo mediante la aplicación de dichas operaciones".

La forma más común de data abstracta, la constituye la definición de tipo de datos, cuyas propiedades pueden ser descritas formalmente, facilitando la construcción de pruebas formales

o informales de la bondad de un programa; cuando un programa se escribe usando sólo construcciones primitivas de un tipo de dato abstracto, su realización al nivel más bajo puede ser variada sustancialmente, sin afectar la operación del programa.

En adición al concepto de data abstracta, se debe tener en consideración otros dos principios muy importantes, siendo el primero de ellos el de ocultamiento de la información, el mismo que consiste básicamente en no sólo hacer visibles ciertas propiedades esenciales de un módulo de software, sino en hacer inaccesibles al Programador aquellos detalles que no tengan importancia. El segundo principio es el de localización, que puede ser aplicado tanto a datos como a estructuras de control resultando en la conocida programación estructurada.

En este punto estamos ya en condiciones de presentar las consideraciones mínimas para el diseño de un lenguaje de programación basado en los aspectos discutidos; es necesario indicar que no se pretende presentar detalles de implementación. Todo lo que debe hacerse, es suponer que dicho lenguaje puede usarse para desarrollar programas "abstractos de alto nivel" que representarán una imagen del proceso que pretenden ejecutar, así como que existe algún medio (compilador) para poder traducirlo a una versión ejecutable en cierto hardware. La representación gráfica utilizada, así como el esquema general se hallan basados en el trabajo de J.Paccassi y C.Wick (1).

DISEÑO INICIAL

El componente principal de este lenguaje es lo que denominaremos PROCESO. Un PROCESO es la menor unidad sintáctica capaz de ser traducida a código ejecutable, por lo que debe ser lo suficientemente completa como para desarrollar una acción. Es lo que más se parece a un programa tradicional. La definición de PROCESO es totalmente recursiva, por lo que un PROCESO puede estar compuesto por sí sólo, o ser utilizado como una subunidad sintáctica de otros PROCESOS exteriores a él; en todo caso, un PROCESO es capaz de existir por sí sólo, requeriendo únicamente del ambiente operativo en que se ejecute. Esto no quiere decir que necesariamente se resuelva todas sus referencias externas a tiempo de traducción, sino que simplemente al momento de ser compilado es transformado a un nivel tal que le sea permitido ejecutar en el ambiente de operación seleccionado; estos detalles corresponden al modo de implementación que se decida utilizar, por lo que deben permanecer ignorados por el momento.

La correcta definición de un PROCESO en sus aspectos sintácticos y semánticos, se basa en la adecuada secuencia de aquellos elementos lexicográficos que contenga; dichos elementos pueden clasificarse en general en dos categorías: TOKENS y SEPARADORES.

Un TOKEN representa una unidad lexicográfica, y constituye la esencia de los elementos del lenguaje. Ejemplos típicos

de TOKEN son: nombres, palabras reservadas, literales, símbolos especiales, etc.; asimismo, un TOKEN puede estar compuesto por uno o más caracteres, que dependerán del tipo de código que se utilice (EBCDIC, ASCII, etc.).

Un SEPARADOR que como su nombre lo indica, cumple la función de proveer cierta "distancia" entre TOKENS, la misma que es necesaria cuando la yuxtaposición de dos TOKENS pudiera hacer que aparezcan como uno sólo para el traductor. Ejemplos pueden ser: espacios en blanco, comentarios, etc.; de igual manera, un SEPARADOR puede estar compuesto de uno o más caracteres.

Un PROCESO realiza sus funciones a través de operaciones en datos que pueden ser externos o internos al mismo; la figura número 2 muestra una representación abstracta de esta estructura.

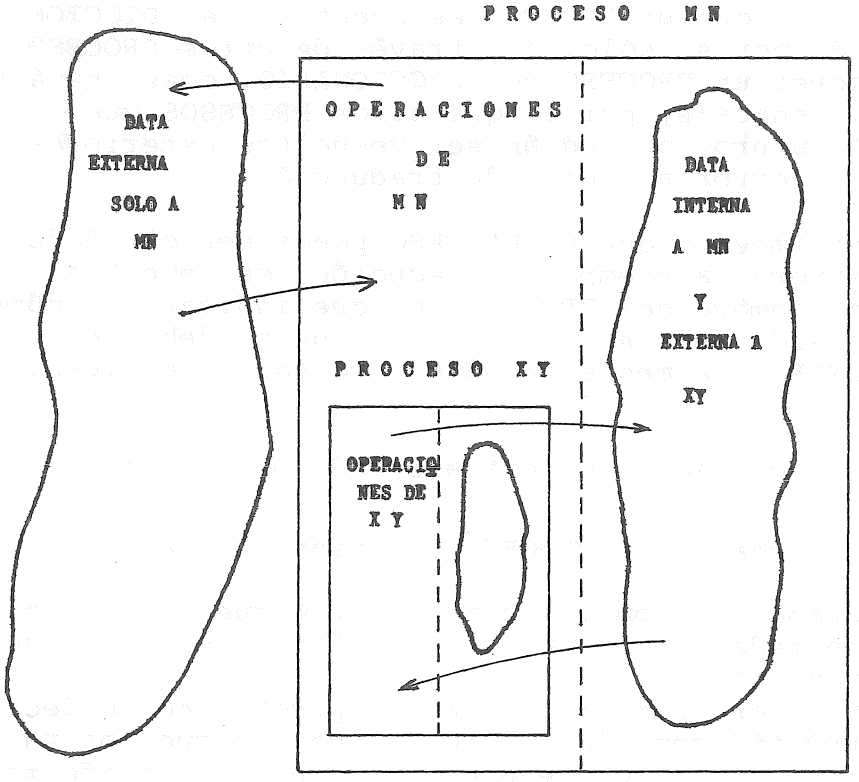


FIGURA Nº 2

Como se puede apreciar, el PROCESO XY existe por sí solo, pero a la vez forma parte del PROCESO MN; la data interna de MN, es a la vez externa para XY, mientras que la data externa de MN, "no existe" para XY, por no haberse definido dentro del alcance del mismo.

Todo PROCESO asimismo, tiene un nombre que lo hace único; sus componentes son:

- Interface de entrada

- Definición de datos
- Operaciones
- Interface de salida.

El único requisito de un PROCESO, deberá ser su nombre, pudiendo por lo tanto, contener ningún componente; esto es lo que denominaremos como PROCESO NULO.

El lenguaje es establecido a través del DICCIONARIO, que es alguna estructura de datos (posiblemente un archivo secuencial por índices, o una base de datos), cuya función es la de almacenar todos los PROCESOS que hayan sido definidos como "recordables"; dicho DICCIONARIO deberá proveer a su vez, de algún método para "olvidar" PROCESOS, asi como posiblemente algún tipo de operación entre PROCESOS, que facilite su ejecución en una forma pseudo-secuencial.

Una vez que un PROCESO es puesto en el DICCIONARIO, puede ser usado por sí sólo, o a través de otros PROCESOS; asimismo, el remover un PROCESO del DICCIONARIO, ocasionará su pérdida para el lenguaje, por lo que otros PROCESOS que lo referencien en el futuro, no podrán ser resueltos exteriormente, y se producirá un error a tiempo de traducción.

Para impedir que un PROCESO pueda ser olvidado y que esto cause errores a tiempo de ejecución, se deberá asociar un contador al nombre del PROCESO, el que indicará el número de veces que ha sido referenciado por lo que no debe ser removido del DICCIONARIO, a menos que dicho contador sea igual a zero binario.

Dicho contador se verá afectado por las siguientes acciones:

- Al crearse un PROCESO, contiene un valor de zero binario.
- Cuando se compila otro PROCESO que lo utiliza, es aumentado en uno, únicamente la primera vez que es referenciado.
- Si dicha compilación es de prueba, no se declara el PROCESO como "aprendible", por lo que los valores de los contadores referenciados no se ven afectados; al"olvidarse" un PROCESO, los contadores respectivos se verán disminuídos en uno.

Como se puede apreciar, el mantenimiento del DICCIONARIO es realizado únicamente por el traductor respectivo; asimismo, si se considera la propiedad de que cada DICCIONARIO es único, y que contiene a su vez datos sobre la frecuencia de utilización de todos los PROCESOS definidos como "recordables", entonces es posible extraer de él, estadísticas de uso de los diferentes PROCESOS, así como otros datos útiles que pudieran considerarse durante la etapa de implementación del lenguaje.

INTERFACES DE ENTRADA Y DE SALIDA

El propósito de definir interfaces de entrada y salida, es el de proveer un medio o canal de transmisión de información entre procesos. La idea, es evitar que pudiesen producirse efectos indirectos no deseados debido a errores en el manejo de los datos internos y que afecten a datos externos no definidos para el PROCESO que causó el error. Las interfaces de entrada y salida son los únicos puntos a través de los cuales un PROCESO puede comunicarse con datos externos, al permitir la definición de todos los parámetros léxicos necesarios para hacer que un dato externo sea accesible al PROCESO a través de la interface de entrada o de permitir que datos externos puedan ser alterados mediante la manipulación adecuada de datos internos, a través de la interface de salida.

La interface de entrada debe constituir el único punto a través del cual se puede recibir o definir datos externos al PROCESO; si es que el PROCESO no requiriese de datos externos, entonces la interface de entrada será nula y podrá ser omitida.

La interface de salida constituirá el único punto a través del cual se podrá hacer que datos internos del PROCESO puedan ser compartidos. Se utiliza para definir el canal de comunicación de salida entre PROCESOS, lo que significa que aquellos datos internos que se vean transformados en datos externos a través de su definición en la interface de salida, podrían a su vez ser usados como datos externos por otros PROCESOS. De igual manera pudiera no ser necesario definir datos externos, en cuyo caso la interface de salida será nula y podrá ser omitida.

Esta posibilidad de compartir dinámica y simultáneamente datos externos a tiempo de ejecución, podría ocasionar serios problemas de transmisión de valores así como de alcance, por lo que se considera que debe ser más estudiada, antes de proceder con su implantación.

DESCRIPCION DE DATOS

Esta sección corresponde a la definición de todos los datos internos de un proceso; dado que deseamos brindar las mayores facilidades en cuanto al manejo de datos se refiere, es necesario distinguir entre dos aspectos muy importantes y que son frecuentemente confundidos: el concepto abstracto de una estructura de datos y una posible realización del mismo; mientras el primero tiene que ver con la definición formal del dato, el segundo se refiere a la manera en que se implementa, y a los posibles estados que puede asumir. El lenguaje propuesto debe proveer facilidades para ambos, y especialmente para producir diferentes realizaciones del mismo concepto. Nótese asimismo, que parte muy importante de la definición, es el posible uso que se le puede dar mediante la aplicación de ciertos Operadores tales como adición, multiplicación, intersección, etc.

Veamos, por claridad, un ejemplo de esta situación: asumamos que nuestro lenguaje permite la definición de un tipo de variables que se "comportan" como stacks; lo único que se nece-

sitaría para poder usarlos sería una expresión tal como:

DECLARE A,B: STACK

En donde se está declarando dos variables del tipo stack, llamadas A y B; a su vez, al definirse el tipo STACK, se deberá tener en cuenta las operaciones que se pueden realizar con él, su alcance, su nivel de protección, etc. Cuando un Programador define una variable como de tipo STACK, automáticamente asume todas sus propiedades, por lo que él sólo tiene que ver con operaciones primitivas tales como PUSH, POP, chequear si el stack está vacío, límites en su uso, etc. Asimismo, la realización de un stack en una máquina determinada, debe ser totalmente transparente al programador.

Esto sugiere que es deseable tener un cierto número de alternativas de implementación para todos los tipos de estructuras de datos que se definan, y que se pueda escojer aquella alternativa que sea más conveniente para determinado programa; sin embargo, es necesario incidir en el hecho de que el Programador no debe asumir ningún tipo de representación, por lo que él sólo debe utilizar aquellos primitivos que se definan para cada tipo de variable. Nuevamente, el problema de decidir cuál es el mejor método, estará a cargo del compilador respectivo, el mismo que seleccionará la mejor alternativa de entre un grupo de diferentes modos de implementación, que también podrían encontrarse definidos en el DICCIONARIO; esto implica que un PROCESO podría contener únicamente el componente de declaración de datos, y ser introducido en el DICCIONARIO para su posterior utilización.

Algunos aspectos que consideramos deben ser evaluados en la definición de datos, son:

- Nombre
- Propiedades de la estructura, las mismas que pueden ser "Importadas" (es decir, definidas con anterioridad en el DICCIONARIO).
- Elementos básicos de datos, (Ejm. bytes, apuntadores, bits, etc.).
- Tamaño de los datos (su representación en hardware, que podría ser dinámica).
- Información de valores iniciales
- Manejo de condiciones de error (interrupciones), por violación de las características de la estructura(OVERFLOW, UNDERFLOW, etc.).

OPERACIONES

El componente de operaciones constituye la parte "activa" de un PROCESO, en el que se realiza alguna tarea, ya sea utilizando las construcciones básicas (estructuras de control) del lenguaje, o mediante el uso de otros PROCESOS que se hallan definidos en el DICCIONARIO.

En este componente asimismo, se crea, modifica o destruye

datos locales, así como se utiliza las interfaces de entrada y salida. Un PROCESO asimismo, puede contener únicamente su componente de operaciones, en cuyo caso se trataría de la definición de algún tipo nuevo de estructura de control, que pueda ser utilizado por otros PROCESOS; este caso determina lo que denominaremos un PROCESO PURO.

Asi como se ha insistido mucho acerca de la importancia de la abstracción en la definición de datos, es necesario indicar que dicha abstracción es también posible en cuanto se refiere a estructuras de control, de asignación, y condicionales. Y esto se deduce fácilmente al coincidir en que si es posible abstraer datos y operaciones, entonces, es posible la abstracción en cuanto a la forma en que deben ser procesados.

Esto se debe precisamente a que al referirse de alguna manera a algún dato, se asocian su representación en el hardware, con la manera en que se comportan las estructuras de control que lo utilizan.

Así por ejemplo, todas aquellas estructuras de control que se relacionan con arreglos, deben necesariamente considerar como es que se hallan dispuesto en memoria (i.e. la suma de matrices en BASIC); sin embargo, dependiendo del Hardware que se utilice, el modo de secuenciamiento que se use será muy diferente si es que se trata de una máquina con paginamiento, o si el arreglo se ha representado como una lista concatenada.

De igual manera, la determinación de parámetros de estructuras de control, tales como el valor inicial, valor final y valor de incremento de una proposición FOR, dictan la manera en que éste se comporta sin tener en cuenta la representación de los datos en memoria, lo cual puede tener un costo muy alto.

El lenguaje propuesto deberá pues, contener alguna forma de poder expresar estructuras de control (o PROCESOS PUROS) que sean consecuencia de la adecuada utilización de los elementos básicos del lenguaje, así como deberá permitir una forma natural de escribir programas, que permitan la mejor representación de la tarea que se desea realizar. Para ello es necesario tener en cuenta que la abstracción en estructuras de control, no depende únicamente del tipo de acciones que se desea efectuar, sino también de la manera en que la definición de datos pudiese determinar cuan abstracto se puede ser en su utilización.

## CONCLUSIONES

A través de una breve introducción a la problemática de desarrollo de software, se aprecia claramente que el enfoque tradicional utilizado se halla determinado en un gran porcentaje, por la posible aplicabilidad que pudiese tener determinado lenguaje de programación, y de su "armonía" con las nuevas técnicas de desarrollo de aplicaciones.

Asimismo, se ha examinado la posibilidad de usar técni-

cas de abstracción, que incluso puedan influir en el diseño mismo del lenguaje; se observa también, que la inclusión o no de determinadas estructuras en un lenguaje, tales como GO TO, CASE, DO ...WHILE, no afecta en general el tipo de aplicaciones que se puede realizar, sino únicamente el modo en que son desarrolladas. Y este es precisamente el punto que se desea hacer destacar; el proceso de desarrollo de software debe ser natural, no forzado ni tampoco limitado por restricciones "artificiales" que el mismo diseñador establece de manera indirecta; el diseño de un lenguaje debe por consiguiente, considerar el poder proveer por lo menos, suficientes facilidades que permitan una adecuada representación de algoritmos, y que sugiera mejores formas de poder expresarlos.

De igual manera se ha presentado en forma muy general, y sin pretender cubrir todos los detalles, aquellos aspectos que creemos debiera considerarse en el diseño de un nuevo lenguaje de programación; al haberse establecido únicamente un primer esqueleto de su estructura, se ha querido dejar para posterior estudio, todo lo concerniente al diseño final y a la etapa de implementación. Creemos sin embargo, que no existe el "lenguaje perfecto", ya que por más elaborado que este sea, no podrá impedir que se escriban con él, programas confusos y con errores, puesto que un programador ingenioso siempre encontrará un método para lograr confundirnos (es decir " ¿Porqué hacer las cosas de una manera fácil, si pueden ser difíciles?").

AGRADECIMIENTO

El autor desea expresar su agradecimiento a Jerry PACCASSI y Carl WICK, con quienes como compañeros de estudios, tuvo oportunidad de aprender mucho de su conversación; de igual manera, a Luis GUILLEN por prestarse voluntariamente a leer y comentar este trabajo; asimismo, al OM1. Juan PEZO y al OM2. Marceliano QUISPE, por su trabajo de mecanografiado.

Finalmente, pero ciertamente no en último lugar, a mi esposa Pelusa por su paciencia y compañía.

## REFERENCIAS


1.- Jerry G. Paccassi y Carl C. WICK.- "A design for a Funcion - Descriptive Programming Language".- trabajo de tésis, USNPGS, 1978.

2.- Glenford J. Myers.- "Software Reliability. Principles and Practices".- John Wiley and Sons, Ing.

3.- William A. Wulf. "Languages and Structured Programs".- Prentice Hall, 1977.

4.- Bárbara Liskov y Stephen Zilles.- "An Introduction to Formal Specifications of Data Abstractions".- Prentice Hall, 1977.

5.- Edsger W. Dijkstra.- "Guarded Commands, Nondeterminacy and Formal Derivation of Programs".- Prentice Hall, 1977.

6.- James R. Low.- "Automatic Data Structure Selection: An Example and Overview".- Communications of the ACM, May 1978, Volume 21, Number 5, pp 376 - 384.

7.- James B. Morris.- " Data Abstraction: A Static Implementation Strategy".- Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colorado. August 6-10, 1979.