SOL: Un Ambiente de Programación

Rafael O. Fontao y Gustavo M. Goñi

Departamento de Ingeniería Eléctrica y de Computadoras
UNIVERSIDAD NACIONAL DEL SUR
Av. Alem 1253 - (B8000CPB) Bahía Blanca, ARGENTINA
e-mail: fontao@uns.edu.ar

Palabras Claves: Metodologías de programación, Lenguajes, Enseñanza de la programación, Concepción de ideas.

Resumen

En este trabajo se describen las características de un ambiente de concepción de programas desarrollado a través de diferentes implementaciones y aplicaciones de la metodología a lo largo de los últimos años (1),(2). En el trabajo se detallan las características del ambiente programado en DELPHI y orientado a la programación en PASCAL.

La idea central consiste en brindar un lenguaje para el programador, que refleje la metodología de concepción de ideas, más que un lenguaje estándar para programar donde finalmente será escrito el problema a resolver.

En este modelo la estructura de control de un programa se define separadamente del resto de las instrucciones del lenguaje de programación, y es modelada a su vez por el comportamiento de una jerarquía de autómatas finitos.

El enunciado inicial de un problema puede concebírselo como un autómata de un solo estado. A cada estado podrá corresponderle hasta dos próximos estados acorde a la evaluación lógica de una condición (si existe) al final de su tarea.

A partir de aquí y en forma recursiva se analiza si un estado particular puede ser sintetizado directamente por el lenguaje de programación disponible. En tal caso la descomposición se detiene, sino el comportamiento del estado particular se descompone en un nuevo autómata en un nivel inferior. Y así sucesivamente en un número finito de pasos se arribará a que todos los estados del autómata (jerarquía de autómatas) pueden ser sintetizados "razonablemente bien" por instrucciones del lenguaje de programación disponible.

La idea de concebir programas mediante este lenguaje (o metodología) resulta de interés como medio unificado de programación, ya que la descripción de la solución de un problema puede plantearse con cierto grado de independencia del lenguaje final a utilizar (paradigma imperativo) y por consiguiente puede permitir la reutilización y portabilidad de programas más eficientemente.

(*) El presente trabajo forma parte del PGI 24/ZK07 - "ESTUDIO SOBRE UN AMBIENTE DE PROGRAMACIÓN" - *Universidad Nacional del Sur*

Introducción

La enseñanza clásica de programación para alumnos de las ingenierías se realiza mediante la utilización de un lenguaje práctico disponible (ej. Pascal, Qbasic, C) a problemas en un contexto académico limitado y aplicando metodologías de programación (típicamente Programación Estructurada) de modo tal de hacer que los programas sean eficientes, claros de interpretar, mantener y por supuesto que hagan lo que se espera que hagan.

En este sentido, entre la escritura de las especificaciones y el listado del programa en que finalmente será ejecutado para cumplirlas, tradicionalmente no hay más que el relato de buenas intenciones y comentarios adecuados en el texto del programa a modo de documentación, pero de la forma cómo el programador fue creando o concibiendo la solución de las especificaciones, la mayoría de las veces quedan pocos vestigios. Los pasos intermedios en la concepción de un programa ceden su lugar al listado de la solución final y su historia suele desaparecer por completo.

Un lenguaje para el programador

El lenguaje **SOL** (por las siglas en Inglés de **Structured Oriented Languaje**) trata de llenar este hueco de historiar la concepción de un programa partiendo de la idea básica de que todo comportamiento secuencial puede modelarse por medio de un autómata finito (1),(3),(4).

La mera observación de cómo se realiza un trabajo, cualquiera sea su naturaleza, puede concebirse como una secuencia de aplicaciones de herramientas de trabajo, seguida de una herramienta de medida que comprueba una condición lógica. Eventualmente la secuencia puede ser vacía y la condición lógica una tautología (es decir, puede obviarse si siempre es verdadera). Y a esta secuencia seguida de una condición, la bautizaremos con el nombre de **trabajo elemental**. Nótese que la noción de elemental corresponde al tipo de herramienta usada y no a la tarea en sí. Por ejemplo, si una herramienta es muy poderosa, hacer una tarea con ella puede resultar elemental pero no si se realiza con herramientas más primitivas.

El nexo entre **estado** de un autómata finito y estado de la programación se da precisamente en la noción de **trabajo elemental**.

El Modelo de descripción de un programa como Autómata finito

Nuestra experiencia docente nos indica que modelar un comportamiento secuencial por medio de autómatas finitos es una noción muy bien aceptada no solo por estudiantes de sistemas digitales sino también por otras disciplinas.

Este hecho está soportado por los trabajos de estudiantes donde autómatas relativamente complejos son diseñados e implementado eficazmente por circuitos digitales.

El comportamiento de un autómata finito (análogamente máquina secuencial) se especifica por medio de una tabla en la cual las filas son los estados (que indican alguna acción a tomar) y las columnas son los estímulos o entradas al sistema. Cada entrada a esta tabla representa el próximo estado que seguirá el autómata dependiendo del estímulo o entrada. Hay un estado inicial y uno o más estados finales.

Para explorar las ventajas de modelar comportamiento secuencial mediante la concepción de autómatas finitos, se propone al siguiente esquema:

Sea L un lenguaje de escritura natural para expresar ideas.

i) Hay un estado inicial.

ii) Cada estado se representa por un programa elemental escrito en L. Es decir, una secuencia de aplicaciones de herramientas de trabajo seguida de una aplicación de medida o testeo.

iii) Hay dos entradas (columnas) que denominamos SI y NO (True and False) que representan los valores lógicos de la condición al final de cada programa elemental (por razones de simplicidad y sin perder generalidad, podemos considerar solo una condición lógica).

El estado actual es el que se está ejecutando y el próximo estado será determinado por el valor lógico de la condición al final del programa elemental.

Sintaxis de SOL

La definición del lenguaje SOL por producciones BNF es la siguiente:

FSA quiere decir: **Autómata de Estados Finito**.
LPD reemplaza a **Lenguaje Práctico Disponible**

<program> ::= <control structure> <actions description>
<control structure> ::= <state refinement> FIN | <state refinement> <control structure>

<state refinement> ::= REF <depth identifier><FSA definition> END <depth identifier>

<depth identifier> ::= null | <identifier>
<identifier> ::= <positive integer > | <positive integer> . <identifier>

<FSA definition> ::= <state description> | <state description><FSA definition>
<state description> ::= <positive integer> DO <task description> THEN <next state list>

<task description> ::= es una descripción escrita del propósito de la tarea elemental

<next state list> ::= <next state identifier> | <next state identifier> OR <next state list>
<next state identifier> ::= <positive integer> | EXIT <positive integer> | STOP

<positive integer> ::= <digit> | <digit><positive integer>
<digit> :: = 0 | 1 | 2 | 3 | 4| 5 | 6 | 7 | 8 | 9

<actions description> ::= DEC <data declarations> < actions list >

<data declarations> ::= etapa de declaraciones (si es que existen) requeridas por el LPD.

<actions list> ::= <action> FIN | <action> <action list>

<action> ::= ACT <identifier> <output state> <exit condition>
<output state> ::= sentencias (en LPD) que sintetizan la salida del estado
<exit condition> ::= NEXT | NEXT (<logic condition>)
<logic condition> ::= expresión lógica permitida en el LPD.

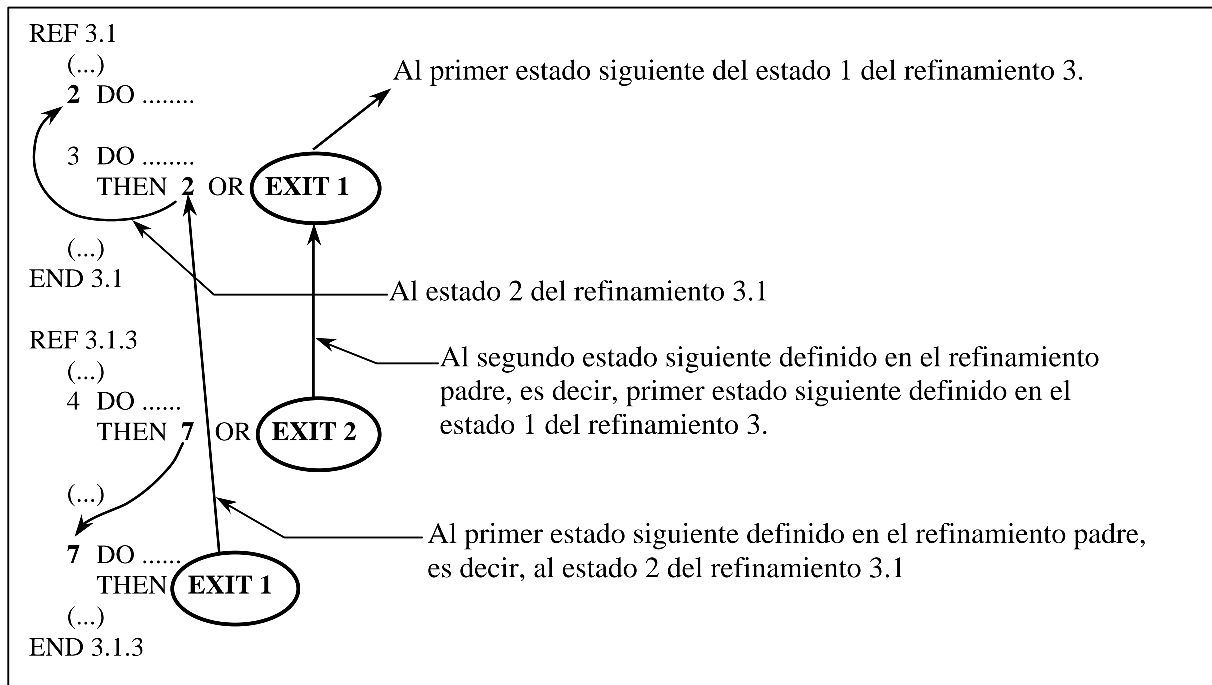
Nota: El uso de los símbolos "< " y " >" en este contexto quiere significar un concepto a definir. Sin embargo en la escritura real de la sintáxis de SOL, las palabras reservadas como REF, END, NEXT, etc. , se delimitarán con estos mismos símbolos a modo de semejanza con la escritura en HTML. Por tal motivo y para evitar confusiones no se los incluyó en la definición de la sintáxis de SOL.

Semántica de SOL

Como se establece en sección anterior, la estructura de control se define separadamente del resto de las sentencias que conforman un programa. La estructura de control a su vez, se compone de una sucesión de refinamientos.

La definición de una FSA se delimita por las palabras REF y END seguidas de un identificador formado por el identificador que refina, seguido de punto y el entero que identifica al estado que está siendo refinado.

Ver la siguiente figura.



El proceso de refinamientos

Los principios de la programación estructurada establecen que un programa debe concebirse en sucesivos refinamientos: En cada paso o refinamiento, un módulo o parte del programa se refina explicitando su comportamiento en mayor detalle con las propias instrucciones del lenguaje. El proceso de refinamiento concluye hasta que prácticamente todo el programa esté escrito lo suficientemente claro en el lenguaje de programación.

En forma análoga el proceso de concepción de un programa puede hacerse mediante autómatas finitos de la siguiente manera:

Sea LPD un lenguaje práctico disponible donde finalmente será escrito el programa.
Paso 0:
Elija un lenguaje de escritura L para expresar sus ideas y conciba a todo el enunciado del problema como si fuera un FSA de un solo estado y una sola entrada (SI).

Paso 1 a N:
Mientras que exista algún estado que no pueda ser escrito "razonablemente claro" en el LPD, concíbalo expresado en lenguaje L como un nuevo autómata.
Por consiguiente, un estado de un autómata es reemplazado por un nuevo autómata en un nivel inferior. Como regla práctica se aconseja que el nuevo autómata tenga pocos estados. Claro está que al menos deben ser 2 nuevos estados ya que de otro modo no se estaría refinando nada. Solo sería establecer un mero sinónimo o re expresión de lo mismo con otras palabras.

Paso N+1:
A esta altura todos los estados podría expresarse claramente en el LPD y en tal caso el paso último es precisamente escribir el código en LPD para todos los estados.

Características de SOL

La metodología de concebir mediante SOL permite aventurarnos en áreas como la portabilidad y reutilización de programas ya escritos en lenguajes convencionales y traducirlos, al menos parcialmente, en otros lenguajes disponibles.

El comando LIKE.

Al conjunto original de comandos aceptados se agregó la instrucción LIKE que permite insertar una nueva definición similar a otra previamente definida. En este sentido es equivalente a expresar que la nueva definición "hereda" de otra anterior la estructura de comportamiento. Es decir, que lo único que se está duplicando es el comportamiento implícito de los estados, ya que ambas partes deberán tener una definición de acciones independientes entre sí. Una restricción de su uso está en que un estado redefinido por LIKE no puede estar dentro del mismo árbol que se especifica en el argumento.

A continuación se ilustra el uso de la instrucción LIKE y lo que implícitamente representa:

```
(...)

REF 3.1
1 DO .......
  THEN 2 OR EXIT 1
2 DO .......
  THEN 1
END 3.1

REF 3.1.1
1 DO .......
  THEN 2
2 DO .......
  THEN 1 OR 3
3 DO .......
  THEN EXIT 1 OR EXIT 2
END 3.1.1

(...)

REF 5.1.2
  LIKE 3.1
END 5.1.2

(...)
```

Uso del comando LIKE

```
(...)

REF 5.1.2
1 DO .......
  THEN 2 OR EXIT 1
2 DO .......
  THEN 1
END 5.1.2

REF 5.1.2.1
1 DO .......
  THEN 2
2 DO .......
  THEN 1 OR 3
3 DO .......
  THEN EXIT 1 OR EXIT 2
END 5.1.2.1

(...)
```

Código implícito en el comando LIKE 3.1

Implementación del Precompilador SOL.

La implementación actual se programó en DELPHI y el primer precompilador está destinado a Pascal por ser este lenguaje el utilizado en los cursos de programación de nuestro Departamento de Ingeniería Eléctrica y de Computadoras.

El precompilador consta básicamente de los siguientes módulos: un editor de código SOL, un editor de código Pascal (donde se almacenará el código generado) y el precompilador propiamente dicho, todo esto integrado en un sólo ambiente EID (Entorno Integrado de Desarrollo).

La tarea del precompilador se divide en dos partes: la primera consiste en generar una Tabla de Estados (o Tabla de Refinamientos) considerando únicamente las definiciones ingresadas dentro de la *estructura de control*, y la segunda en generar el código en *Lenguaje Práctico Disponible* (en nuestro caso Pascal) a partir de la *descripción de acciones* definidos en el código SOL.

Estructura de Control y Tabla de Estados.

Analizando las definiciones de los refinamientos ingresados en el código SOL, el precompilador construye una *Tabla de Estados* donde se almacenan los siguientes datos para cada estado definido:

- Identificador del Refinamiento. (Definido explícitamente en el comando REF)
- Identificador del Estado dentro del Refinamiento.
- Número de Estado siguiente si el resultado de la condición es verdadero.
- Número de Estado siguiente si el resultado de la condición es falso.
- Si el estado ha sido refinado, entonces se indica la posición dentro de la tabla del primer estado correspondiente al refinamiento.
- Si el estado es parte de un refinamiento, se indica en qué posición se encuentra aquel estado que está siendo refinado (estado generador o predecesor)

También se almacenan otros campos que facilitan la tarea a la hora de generar el programa en lenguaje Pascal, por ejemplo: ubicación de los comandos <ACT> para cada uno de los estados que lo requieran.

Para generar la tabla sólo es necesario realizar una única pasada por la porción del código SOL donde se definen los refinamientos. A continuación se muestra un ejemplo de cómo se construye dicha tabla:

```
REF x
…
y DO…
   THEN ….
…
END x
…
REF x.y
u DO…
   THEN v OR exit 1
…
v DO …
  THEN…
END x.y
```

Código SOL

	Sgte V	Sgte F	ID. Ref.	Estado	Refinamiento	Generador
...						
i	j	-	*x*	*y*	k	
...						
...						
k	m	j	*x.y*	*u*		i
...						
m			*x.y*	*v*		i
...						

Tabla de Estados

Si analizamos el *Código SOL* de la izquierda podemos ver que un determinado refinamiento *x* posee un estado denominado *y*. A su vez vemos que dicho estado ha sido refinado y que posee al menos dos estados *u* y *v*. Si observamos la *Tabla de Estados* que se ha construido vemos que en la posición *i* se encuentra la información referente al estado *y* del refinamiento *x,* y en las posiciones *k* y *m* la información de los estados *u* y *v*, respectivamente, pertenecientes ambos al refinamiento *x.y*.

En el campo *Refinamiento* asociado al estado *y* vemos que el mismo ha sido refinado y que el primer estado se encuentra en la posición *k* de la tabla, que lógicamente corresponde al estado *u*. Aquí podemos ver que se completa una referencia cruzada entre ambos estados ya que en el campo *Generador* asociado al estado *u* se indica la posición que tiene su padre dentro de la tabla.

Con respecto a los campos Sgte_V y Sgte_F podemos decir que los mismos se completan a partir de la información indicada en los comandos THEN de cada estado.

Generación del Código en Lenguaje Práctico Disponible (LPD).

Una vez que la definición de refinamientos ha concluido, es entonces cuando comienza el armado del programa en el Lenguaje Práctico Disponible (LPD).

La primera tarea que se realiza al generar el código en LPD es copiar todas las líneas que el programador ha escrito entre los comandos <DEC> y <ACTIONS>. Luego se recorre la tabla en busca de los estados que no han sido refinados y se declaran las etiquetas correspondientes a dichos estados.

Como la tabla de estados representa una estructura de tipo árbol, lo que se hará es recorrer la misma en modo Depth-First (en profundidad) en busca de aquellos estados que no poseen refinamiento (estados hojas), ya que los mismos son los que tienen asociados acciones en LPD. El árbol es recorrido en profundidad ya que de esa forma el código generado mantendrá el orden lógico con el que ha sido diseñado.

Una vez hallado un estado hoja se procede a insertar la etiqueta asociada a dicho estado y a copiar todo el código que aparece entre los comandos <ACT> y <NEXT> que pertenezcan a dicho estado. Luego se analiza si el comando <NEXT> debe poseer alguna condición para pasar al estado siguiente. Si la misma existe entonces se crea un bloque IF..THEN ó IF..THEN..ELSE en el código LPD, usando la condición indicada en el comando <NEXT>.

Control de Errores de Sintaxis.

Mientras el precompilador realiza su tarea verifica si el código que está siendo analizado es sintácticamente correcto. En caso de detectar un error avisará al usuario en qué línea ha encontrado el mismo y lo acompañará con un mensaje indicando la causa de la detección. Algunos de los posibles mensajes error son los siguientes:

#1 : 'Falta Identificador de Refinamiento.'
#2 : 'Falta sentencia DO.'
#3 : 'Identificador de Refinamiento no coincide.'
#4 : 'Parámetro incorrecto en sentencia THEN.'
#5 : 'Estado no definido.'
#6 : 'END sin REF.'
#7 : 'REF sin END.'
#8 : 'Identificador de Refinamiento incorrecto.'
#9 : 'Identificador de Estado incorrecto.'
#10 : 'Error de sintaxis.'
#11 : 'La tarea que se desea refinar no ha sido definida.'
#12 : 'Refinamiento duplicado de una misma tarea.'
#104 : 'Parámetro incorrecto en sentencia NEXT.'
#106 : 'ACTIONS sin DEC.'
#109 : 'No se pueden asignar acciones a un estado con refinamiento.'
#112 : 'Acciones duplicadas para un mismo refinamiento.'

Trabajo Actual.

Actualmente se está desarrollando una extensión del lenguaje SOL donde se incluye un soporte para subprogramas o módulos (procedimientos y/o funciones). La estrategia consiste, básicamente, en implementar cada uno de los módulos en lenguaje SOL (en forma independiente) y luego permitir la inclusión de los mismos dentro de la zona declarativa (<DEC>) del código SOL del programa. Al momento de realizar el precompilado se procesará cada uno de los módulos por separado y luego se integrarán los códigos obtenidos al programa que los incluye. Esto implica que se

```
(...)
<DEC>
(...)

procedure PROC1(parámetros);
   <SOL 'proc1.sol'>
(...)
<actions>

(...)
```

Subprogramas en SOL

generará una Tabla de Estados por cada módulo refinado con SOL.

Líneas de Trabajo:

- Optimización del código generado en LPD.
- Realizar una aplicación que permita simular el comportamiento del programa implementado en código SOL, aún cuando todavía no se hayan definidos las acciones en LPD.
- Soporte para múltiples Lenguajes Prácticos Disponibles (Pascal, C, Fortran, Basic, incluyendo las versiones Visual , etc.)

Ejemplo de una Aplicación desarrollada en SOL (sin la implementación de las acciones).

```
REF   #GESTION COMERCIAL FACTURACIÓN, CTA. CTE  Y STOCK
1 DO  CONSIDERACIONES INICIALES
   THEN 2
2  DO PRESENTACIÓN DEL  MENÚ CON OPCIONES  DE:
          * FACTURACIÓN
          * CUENTA CORRIENTE
          * STOCK
          * FINALIZACIÓN
   THEN 3 OR 4 OR 5 OR 6
3   DO  FACTURANDO
     THEN 2
4   DO EN CUENTA CORRIENTE
    THEN 2
5   DO EN GESTIÓN DE STOCK
     THEN 2
6   DO CONFIRMACIÓN Y CONSIDERACIONES FINALES
    THEN  STOP OR 2
 END

REF 1  # CONSIDERACIONES INICIALES
1  DO  HACER TODO LO NECESARIO ANTES DE INICIAR EL SISTEMA
    THEN  EXIT 1
END 1

REF 2   # EL MENÚ PRINCIPAL
1 DO LA PANTALLA DEBE EXPONER CLARAMENTE LAS OPCIONES
* QUE SE ESPERAN VÍA MENÚ DESPLEGABLE, POR LISTA DE OPCIONES
* O POR BOTONES QUE CON UN CLICK DEL MOUSE SE ELIJAN.
   THEN EXIT 1
END 2

REF  3  # FACTURANDO
1 DO EXPONER EL MENÚ DE FACTURACIÓN QUE DEBERÁ INCLUIR LAS
* OPCIONES DE:
*   CAMBIOS DE FORMAS DE FACTURAR Y PARÁMETROS
*   IR A FACTURAR CON LOS PARAMETROS ACTUALES
*   LISTADOS DE FACTURACIÓN
*   ESTABLECER CICLOS DE VENTA
*   VOLVER
   THEN 2 OR 3 OR 4 OR 5 OR EXIT 1
2  DO CAMBIOS DE FORMAS DE FACTURAR Y PARAMETROS
   THEN 1
```

```
3  DO IR A FACTURAR CON LOS PARÁMETROS ACTUALES
   THEN 1
4  DO LISTADOS DE FACTURACIÓN
    THEN 1
5  DO ESTABLECER CICLOS DE VENTA
* EN ESTA ACCION SE PUEDEN INICIALIZAR LAS ESTADÍSTICAS DE VENTAS
    THEN 1
END 3

REF 3.3   # FACTURAR CON LOS PARÁMETROS ACTUALES
1 DO CONSIDERACIONES INICIALES
* CHEQUEO DE LOS PRÓXIMOS NÚMEROS DE FACTURAS, TIPOS
* VERIFICACIÓN ESTADO DE IMPRESORAS Y/O CONTROLADORES
* FISCALES ETC.
THEN 2 OR 3
2 DO MENSAJES DE ERROR QUE CORRESPONDAN
   THEN EXIT 1
3 DO CICLO DE FACTURACIÓN
   THEN EXIT 1
END 3.3

REF 3.3.3

1 DO INGRESO DEL TIPO DE FACTURACIÓN O CANCELAR
   THEN 2 OR EXIT 1
2  DO INGRESO DE DATOS DEL DESTINATARIO DE LA FACTURA
   THEN 3
3  DO INGRESO DE LOS ÍTEMES DE FACTURACIÓN O FINALIZAR
   THEN 3 OR 4
4 DO MOSTRAR EN PANTALLA LOS DATOS DE LA FACTURACIÓN
* Y CONFIRMAR SU EMISIÓN
   THEN 5 OR 1
5 DO EMITIR FACTURA Y ACTUALIZAR ESTADÍSTICAS Y
* GESTIÓN DE STOCK
   THEN 2
END 3.3.3


REF 4  # CUENTA CORRIENTE
1 DO MENÚ DE OPCIONES DE CUENTA CORRIENTE
   THEN 2 OR 3 OR 4 OR EXIT 1
2   DO ALTAS, BAJAS Y MODIFICACIONES DE DATOS
   THEN 1
3   DO LISTADOS  DE ESTADO DE CUENTAS CORRIENTES
   THEN 1
END 4


REF 5  #  GESTIÓN DE STOCK
1 DO MENÚ DE OPCIONES DE  STOCK
   THEN 2 OR 3 OR 4 OR EXIT 1
2   DO ALTAS, BAJAS Y MODIFICACIONES DE DATOS
   THEN 1
3   DO LISTADOS  VARIOS DE STOCK
* DE EXISTENCIA
* DEBAJO DEL PUNTO DE PEDIDO
* DE MAYOR VENTA
* ETC.
  THEN 4 OR 5 OR 6  OR 1
```

```
4 DO DE EXISTENCIA
   THEN 1
5  DO DEBAJO DEL PUNTO DE PEDIDO
    THEN 1
6   DO DE MAYOR VENTA ( EJEMPLO )
    THEN 1
END 4
```

Este ejemplo muestra como se puede compartir y distribuir por Internet la "inteligencia" de una clase de problemas a modo de repositorio de concepción de ideas básicas. Luego cada programador individual o equipo podrá tomar esta idea ampliarla y/o implementar las acciones en el lenguaje práctico disponible concentrándose en cada acción (supuestamente una pocas instrucciones) y dejando para el precompilador la vinculación con el resto del sistema. Finalmente el compilador del lenguaje respectivo hará el ejecutable de la aplicación.

Referencias

(1) Fontao, R.O., Ardenghi, J.R. y Arroyo, E.H. "Sobre una Metodología de Programación". Segundo Simposium sobre Aplicaciones de la Ingeniería Eléctrica y Electrónica, SIEEM 77, Monterrey, Mexico 1977.

(2) Fontao, R.O. y Codagnone, J.A. "Experiencias sobre Una Metodología de Programación". Anales de la VIII Conferencia Latioamericana de Informática, Buenos Aires 1981.

(3) Fontao, Rafael O, Kalocai, G., Ramoscelli, G. Y Goñi, G. Una propuesta de investigación sobre "A PROGRAMMING METHODOLOGY". Workshop on Programming Methodology - WG 2.3 - IFIP, Tandil Septiembre 2000.

(4) Fontao, Rafael O, Kalocai, G., Ramoscelli, G. Y Goñi, G. "SOL: Un Lenguaje para el Programador". 4to. Workshop de Investigadores en Ciencias de la Computación. Bahía Blanca, Mayo de 2002

Más información en http://lip.uns.edu.ar/sol/