"Un lenguaje intermedio para mejorar y traducir programas"

Cobo, Hernán Mauco, M. Virginia Moreno, Marcos Zoia, Ariel
Instituto de Sistemas de Tandil
Depto. de Computación y Sistemas
Facultad de Ciencias Exactas
Universidad Nacional del Centro de la Pcia. de Buenos Aires
San Martín 57 - (7000) Tandil - Bs. As. - Argentina
e-mail: hcobo@rec.unicen.edu.ar vmauco@tandil.edu.ar azoia@rec.unicen.edu.ar

Resumen

Actualmente perduran una gran cantidad de sistemas implementados en lenguajes imperativos previos a la revolución de la programación estructurada. Aunque esos programas se hubieran desarrollado con las mejores técnicas de diseño y codificación disponibles en su momento, los cambios continuos degradaron su estructura. Este artículo presenta un prototipo que permite la reestructuración y traducción de sistemas realizados en algún lenguaje imperativo (Cobol, Fortran, Pascal o C) hacia cualquier otro del mismo conjunto. Describe también la estructura intermedia creada para tal fin.

Palabras claves: reestructuración de código, mantenimiento de software, grafo de dependencias del programa, reingeniería, transcompilador, lenguajes imperativos.

1. INTRODUCCION

La mayor parte del esfuerzo en el ciclo de vida de los sistemas se dedica al mantenimiento. Algunas estimaciones indican que la proporción de recursos y tiempo dedicados al mismo varían entre un 50% y un 70% [9].

Una gran cantidad de sistemas existentes tienen una media de edad de entre 10 y 15 años [11]. Aunque esos programas se hubieran desarrollado con las mejores técnicas de diseño y codificación disponibles en su momento, los cambios continuos degradaron su estructura, transformándolos en sistemas con estructuras de datos pobremente diseñadas, una codificación, una lógica y una documentación desactualizadas, que tienen que seguir funcionando en la actualidad. Generalmente, esto implica que los costos de mantenimiento sean más elevados.

En los últimos años las herramientas de reestructuración, ingeniería reversa y reingeniería están empezando a ocupar un lugar fundamental en el proceso de mantenimiento del software [2]. La reestructuración de software es una opción importante para poner los altos costos de mantenimiento bajo control. Dentro de las técnicas de reestructuración de software, la reestructuración de código ocupa un lugar muy importante.

El objetivo de la reestructuración es transformar programas no estructurados en programas estructurados equivalentes que se puedan expresar en términos de secuencias y anidamientos de programas más pequeños con una entrada y una salida, y que se puedan leer y entender sistemáticamente [2]. El gran potencial de la reestructuración automática de programas está en manejar la estructura de programas grandes, en la escala de miles de líneas de código, donde hay un gran impacto económico.

Las representaciones intermedias elegidas para los sistemas de software juegan un rol central para el proceso y los objetivos de las herramientas antes mencionadas [2]. Estas representaciones deben facilitar la implementación de las herramientas permitiendo realizar todas las operaciones mediante procedimientos fácilmente programables. El problema de la representación tiene que ver con determinar cuáles son adecuadas y bajo qué circunstancias se deberían usar, ya que cada una de ellas puede ser más apropiada dependiendo del contexto específico [6] [7] [10].

Este trabajo es la primera etapa de un proyecto cuyo objetivo es desarrollar una herramienta para contribuir a simplificar el mantenimiento de los sistemas desarrollados en algún lenguaje imperativo. Para esto, es necesario obtener toda la información contenida en el código fuente de un programa correctamente escrito en un lenguaje imperativo, y representarla en una estructura de más alto nivel que se pueda analizar y manipular de forma tal que permita generar código que, además de mantener la funcionalidad original, sea estructurado, más legible, modular, flexible, reusable, y por lo tanto más fácil de mantener. Algunas de estas características pueden significar un cambio de lenguaje, y para este cambio es necesario reescribir el programa con diferente sintaxis.

Para representar toda la información extraída del código fuente de los programas y para que la herramienta se pueda aplicar a programas escritos en distintos lenguajes imperativos, se desarrolló una nueva representación llamada lenguaje intermedio. Una de las ventajas de la misma es que los algoritmos diseñados para mejorar la calidad del código de los programas, son independientes del lenguaje fuente del programa. Además permite que los programas reestructurados se escriban en un lenguaje distinto al lenguaje original. Los lenguajes de programación imperativos considerados son C, Pascal, Cobol y Fortran.

En este artículo se presenta una definición del lenguaje intermedio y se describen las etapas a seguir para obtener desde un programa escrito en C, Pascal, Fortran o Cobol un programa estructurado de funcionalidad equivalente escrito en el mismo lenguaje o en otro. Se presentan además, una definición del grafo de dependencias del programa extendido que es la estructura utilizada para representar los programas escritos en lenguaje intermedio y una breve descripción del algoritmo de reestructuración desarrollado.

2. LENGUAJE INTERMEDIO (LI)

La necesidad de una representación intermedia surge al plantearse como objetivo mantener la información de un sistema despojada de la sintaxis del lenguaje en el cual fue concebido dicho sistema. Teniendo entonces la información necesaria, es posible reproducir el sistema en otra sintaxis que mantenga las formas básicas del paradigma de programación imperativa.

Se seleccionaron cuatro lenguajes fundados en el paradigma imperativo: Cobol, Fortran, Pascal y C. Dichos lenguajes fueron elegidos a partir de su popularidad y trayectoria en el desarrollo de sistemas. Además, los dos primeros son propensos a prácticas de programación desprolijas.

Se definió una estructura capaz de representar, entre otras cosas, el árbol sintáctico de cualquiera de los lenguajes mencionados anteriormente. Dicha estructura contiene también la información necesaria para reconstruir los programas, tanto en el lenguaje original como en cualquier otro.

A partir de la elección de los lenguajes fue necesario hacer un estudio detallado de su sintaxis. Esto permitió obtener un conjunto de sentencias tipo que comparten los lenguajes en estudio.

Tales sentencias fueron representadas en LI siguiendo diferentes caminos.

Algunas de estas sentencias mantienen una relación uno a uno con su representación en LI. Las mismas se construyeron agrupando las características comunes en cada uno de los lenguajes escogidos.

Otras, de funcionalidad más compleja, necesitaron de más de una sentencia de LI para ser representadas, por lo que fue necesario crear otro tipo de sentencia de un nivel de abstracción superior (sentencia de bloque) que englobe dicho conjunto para no perder información. Este tipo se utiliza además para caracterizar sentencias que no son factor común en los cuatro lenguajes pero que pueden construirse componiendo otras más simples.

También se definieron sentencias primitivas propias del LI que son necesarias, entre otras cosas, para representar estructuras de control a partir únicamente de saltos condicionales, incondicionales y etiquetas.

Por último, se creó un tipo de sentencia llamada particular destinada a representar las sentencias que no sean comunes y que tampoco puedan representarse a partir de relaciones entre las ya definidas. Sobre estas sentencias se actúa extrayendo las variables de entrada y salida correspondientes.

En resumen, las sentencias de LI se clasifican en dos niveles: uno inferior e indivisible en el que se representan las sentencias simples de funcionalidad básica y otro de abstracción superior para las sentencias de funcionalidad más compleja .

Un sistema en LI queda representado en distintos niveles de abstracción. Las sentencias del nivel inferior se encadenan entre sí para formar toda la estructura de control del programa representado; para esta tarea, cuenta con primitivas de control ya mencionadas. Las sentencias de nivel superior referencian las sentencias simples y primitivas que conforman las de funcionalidad más compleja. Además, se utilizan para referenciar las sentencias que son contenidas en las estructuras de bloque como los ciclos, condicionales, etc. También es importante destacar que este tipo de sentencias no forman parte del flujo de control.

Existen primitivas en algunos lenguajes que involucran más de una acción conceptual; en estos casos se utiliza una sentencia de bloque de LI que engloba a todas las primitivas de LI correspondientes para realizar la tarea equivalente. Por ejemplo la sentencia writeln en Pascal se representa por una sentencia de bloque que contiene dos primitivas, una es escribir y otra desplazar.

Todas estas sentencias están encuadradas en una estructura que las almacena y que cuenta también para cada una con un número de sentencia, el texto original de la sentencia en el programa fuente y un número que utilizan las sentencias primitivas para referenciar a la sentencia de bloque en la que están contenidas.

Este LI mantiene toda la información del sistema sin necesidad de definir estructuras adicionales y permite contar con la posibilidad de tener dos enfoques distintos del sistema en estudio. Uno desde el flujo de control directo del sistema y otro desde su estructura de bloques, pudiendo graduar el nivel de detalle para visualizar el mismo.

Por otro lado contar con sentencias primitivas, tales como los saltos condicionales e incondicionales, permite representar cualquier estructura de control nueva. Además, al disponer de una construcción de más alto nivel, como las sentencias de bloque, no se pierde información de esta nueva estructura.

Al final de este artículo se muestra un ejemplo donde se representa en LI un pequeño programa Cobol.

2.1. Especificación de la estructura de Lenguaje Intermedio

Programa ::= sentencia(**Identificador**, TipoSentenciaLI, **Texto**, **Composición**)*
TipoSentenciaLI ::= SentBloque | SentSimple | SentPrimitiva | SentParticular
SentBloque ::= sent_bloque(**Nombre**,**ListaComponentes**,**ListaSentenciasContenidas**)
SentSimple ::= **nombre_sent**(**Contenido**)
SentPrimitiva ::= **nombre_sent**(**Contenido**)
SentParticular ::= sent_particular(**ListaDatosUtil**,**ListaDatosModif**)

Identificador: Número unívoco creciente que representa una sentencia en LI
Texto: Almacena el texto de la sentencia. Es vacío en caso de ser una sentencia primitiva.
Composición: Identificador de la sentencia de bloque que compone. Está vacía en caso de ser una sentencia de bloque o una sentencia simple
Nombre: Nombre que identifica el tipo de sentencia que está agrupando.
ListaComponentes: Identificadores de sentencias primitivas del LI que construyen la sentencia de bloque.
ListaSentenciasContenidas: Identificadores de las sentencias contenidas en el bloque.
nombre_sent: Texto que representa una acción o un tipo de acción dentro del LI
Contenido: Una serie de parámetros que detallan aspectos propios de cada sentencia.

ListaDatosUtil: Lista de datos utilizados por la sentencia.
ListaDatosModif: Lista de datos modificados por la sentencia.

2.2. Descripción de las sentencias más representativas

- **Sentencias simples**

asig(ListaAsignantes, ListaAsignados)
Define la asignación del contenido (valores, funciones, etc.) de la lista de asignantes en un lugar físico asociado con un nombre que figura en la lista de asignados. La operación sigue el formato fuente - destino y las dos listas son necesarias porque algunos lenguajes cuentan con asignaciones múltiples.

operación (Operador, PrimerOperando, SegundoOperando)
Abstrae la definición de toda operación unaria o binaria, donde el primer parámetro corresponde al operador y, el segundo y tercero corresponden a los operandos a tratar, siendo posible agruparlas en construcciones anidadas. En las operaciones unarias el tercer parámetro se identifica con un "vacío". Tienen definidas reglas para las operaciones que no sean conmutativas con el objetivo de que su interpretación no sea ambigua.
Las operaciones que se tratan son: suma; resta; multiplicación; división; potencia; dirección; tamaño; etc.

goto(etiq_interna(NúmeroSentencia))
Define un salto de control producido por el usuario hacia una sentencia cuyo número se encuentra como parámetro de la etiqueta interna. Ésta almacena el identificador de la externa en la primera fase del analizador.

- **Sentencias primitivas**

subrutina(ident(Nombre), tipo(NombreTipo))
Actúa como primitiva de la sentencia de bloque SUBRUTINA, siendo la sentencia destino de los saltos de control. El primer parámetro define el nombre del procedimiento y el segundo su tipo, en caso de que sea una función.

ira(etiq_interna(NúmeroSentencia))
Define un salto de control hacia una sentencia cuyo número se encuentra como parámetro de la etiqueta interna. Forma parte de la construcción de casi todas las estructuras de control, y fue creada con ese nombre para diferenciarla de la sentencia goto que es un salto de control existente en el código original.

cond(Expresión, etiq_interna(NúmeroSentencia), etiq_interna(NúmeroSentencia))
Define un salto condicional; el primer parámetro almacena la expresión a evaluar siendo el segundo y el tercero las representaciones de los saltos por verdadero y por falso respectivamente. Forma parte de las sentencias de bloque condicionales y de repetición.
Una expresión se representa como:
relacional(NombreOperador, PrimerOperando, SegundoOperando)
Define las operaciones relacionales (mayor; menor; igual; in; conj_iguales; conj_distintos; incluídos)
lógico(NombreOperador, PrimerOperando, SegundoOperando)
Define las operaciones lógicas (o; y; no; numérico; alfabético; eof; tamaño)

- **Sentencias de bloque**

COND Define la estructura de control "IF - THEN - ELSE", con sus respectivos bloques de sentencias. Contiene una sentencia primitiva *cond* para almacenar la expresión y en el caso de tener ELSE contará con una primitiva de salto. Además, mantiene los números de las sentencias (ya sean simples o de bloque) contenidas en esta estructura de control.

SUBRUTINA Define la estructura de una subrutina conteniendo su identificador, parámetros, retorno y bloque de sentencias. El identificador se mantiene en una sentencia primitiva subrutina detallada anteriormente y los parámetros en sentencias simples de asignación. Todas las sentencias que forman parte de la subrutina están referenciadas por la lista de sentencias contenidas.

CICLO Define las distintas estructuras de control iterativas. Como por ej : While de Pascal, Perfom de Cobol, For de C. Cada construcción cuenta con sentencias primitivas condicionales, de salto y otras propias de las características de la sentencia origen. Todas las sentencias que incluye el ciclo están almacenadas en la lista de sentencias contenidas.

EXCEPCION Se utiliza para definir sentencias con condiciones de excepción; expresa entonces, la estructura de dicha sentencia, su condición de excepción, y las sentencias correspondientes a la acción.

ABRIR Define la estructura de la apertura de un archivo, incluyendo tipo de archivo, asignación y posicionamiento del buffer.

- **Representaciones constructoras**

asoc(Expresión)
Representa una expresión limitada por paréntesis. Es utilizada en operaciones aritméticas, lógicas y relacionales.

ident(Nombre)
Representa un identificador de variable simple.

numero(Nombre)
Representa una constante numérica.

matriz(ident(IdentificadorMatriz), ListaIndices)
Representa una estructura vectorial, donde el primer argumento es el nombre asociado a la estructura y el segundo argumento es una lista dinámica determinando los valores de uno o varios índices.

3. GRAFO DE DEPENDENCIAS DEL PROGRAMA EXTENDIDO (GDPE)

Para poder representar la información de flujo de control y de datos contenida en el LI, se definió una nueva estructura cuya principal característica es que permite un fácil acceso a la información que almacena, simplificando así los algoritmos diseñados para manipularla.

Esta estructura es el grafo de dependencias del programa extendido (GDPE) [3] que corresponde a una extensión del grafo de dependencias del programa (GDP) propuesto en [4].

La representación correspondiente al GDPE es un grafo orientado en el cual los nodos son sentencias y expresiones de predicados y los arcos incidentes a un nodo representan los valores de datos y las condiciones de control de las cuales dependen las operaciones del nodo.

Las dependencias son de dos tipos. Dependencias de datos, que surgen entre dos sentencias cuando una variable que aparece en una de ellas podría tener un valor incorrecto si se invirtiera el orden entre las dos

sentencias. Dependencias de control, que se dan entre una sentencia y el predicado cuyo valor controla inmediatamente la ejecución de esa sentencia.

El conjunto de todas las dependencias de un programa induce un orden parcial sobre sus sentencias y predicados, el cual se debe seguir para preservar la funcionalidad del programa original.

Una de las características más importantes del GDPE es que expone paralelismo potencial. Esto significa que dos nodos n_1 y n_2 se podrían ejecutar simultáneamente a menos que existiera una dependencia entre ellos.

Se puede considerar al GDPE separado en dos componentes: el subgrafo de dependencias de control y el subgrafo de dependencias de datos. El subgrafo de dependencias de control contiene información sobre las dependencias de control que existen entre las sentencias del programa. Se construye a partir del grafo de flujo de control del programa y del árbol de post-dominantes [4]. Cada nodo n de este subgrafo tiene asignado un número de nivel que representa la longitud del camino desde el nodo inicial del subgrafo al nodo n. La incorporación del nivel para cada uno de los nodos, permite determinar rápida y simplemente si un programa está o no estructurado, y además contribuye a simplificar las transformaciones definidas para convertir un programa no estructurado en uno estructurado [3]. El subgrafo de dependencias de datos almacena información relativa a las dependencias entre las sentencias del programa debido a definiciones y usos de los items de datos.

Como en este trabajo se presenta la reestructuración de programas en relación a su estructura sintáctica, se tendrán en cuenta, únicamente, las dependencias de control. Pero será necesario agregar, temporalmente, arcos que indiquen el orden de ejecución de los nodos que tienen el mismo nivel, para asegurar la preservación de la funcionalidad del programa original. Este orden se corresponde con el orden impuesto por el grafo de flujo de control.

4. ETAPAS DEL PROCESO DE TRANSFORMACIÓN

4.1. Pasaje de Lenguaje Fuente a Lenguaje Intermedio

Cada lenguaje fuente requiere un algoritmo propio para resolver la conversión de sus sentencias a LI; dicho algoritmo posee las características normales de un transcompilador [1].

Una particularidad de esta transformación es que la conversión de sentencias se realiza en dos niveles de abstracción simultáneamente.

4.2. Pasaje de Lenguaje Intermedio A Grafo de Dependencia de Programa Extendido

Un programa escrito en LI se representa mediante un conjunto de GDPE's, uno de los cuales corresponde al programa principal y los restantes a cada una de las subrutinas del mismo.

Para esto, se genera un grafo de flujo de control y un árbol de post-dominantes para el programa principal y para cada subrutina. Cada sentencia se representa con un nodo en el grafo de flujo de control correspondiente, con excepción de las sentencias de salto de control (goto, ira, etc.) que generan arcos, y las sentencias de llamada a una subrutina que se representan con un nodo y un arco de llamada cuyo destino es el grafo de flujo de control de la subrutina correspondiente. En la construcción de los grafos de flujo se resuelven los saltos de control de manera tal que los mismos no excedan los límites de cada subrutina. De esta forma, la única relación posible entre cualquier par de grafos de flujo de control es a través de los arcos que representan la llamada de una subrutina a la otra.

Se obtiene así un conjunto de GDPE's vinculados mediante arcos de llamada. En cada GDPE, cada nodo representa una sentencia de LI y depende por control de nodos del mismo GDPE al que pertenece. La determinación del nivel de cada nodo de un GDPE se realiza en forma independiente para cada GDPE.

4.3. Reestructuración

Un programa está estructurado si se construye únicamente a partir de las estructuras de control: secuencia, selección e iteración [8].

Un GDPE está estructurado si se construye a partir de la composición de las estructuras de control permitidas. En un GDPE estructurado cada nodo depende por control de solamente uno de los nodos del nivel inmediato superior. Esto implica que el subgrafo inducido por las dependencias de control es un árbol.

Por lo tanto, un programa está estructurado si se puede representar con un conjunto de GDPE's estructurados. Se puede concluir, entonces, que para determinar que un programa no está estructurado será necesario y suficiente encontrar un arco de control que destruya la estructura de árbol de al menos uno de los GDPE's que constituyen su representación [3].

El proceso de reestructuración analiza cada GDPE por separado para determinar si existen dependencias de control que destruyen la estructura de árbol. En caso de encontrar alguna, se aplica al GDPE correspondiente el algoritmo de reestructuración definido en [3]. Es importante destacar que este algoritmo no modifica los subgrafos del GDPE que ya estén estructurados. Además, como se utiliza solamente para aquellos GDPE's no estructurados, las subrutinas del programa que ya estuvieran estructuradas conservarán la estructura original.

La descripción completa del algoritmo de reestructuración para un GDPE se puede encontrar en [3].

4.4. Pasaje de Grafo de Dependencias del Programa a Lenguaje Intermedio

Esta transformación toma como guía el grafo con sus arcos de secuencia.

Las sentencias que no involucran estructuras de control se transfieren tomando sus referencias de los nodos. También es necesario obtener las sentencias de bloque que están involucradas para transferirlas.

Por otro lado, todas las estructuras de control se deben generar nuevamente a partir de la topología resultante del grafo.

4.5. Pasaje del Lenguaje Intermedio al Lenguaje Fuente Destino

Al igual que el proceso inverso, mencionado anteriormente, esta transformación tiene características similares a las de un transcompilador [1].

5. IMPLEMENTACION

La estructura principal de este prototipo se encuentra implementada en Prolog, manteniendo todas las estructuras intermedias en bases del mismo lenguaje. La única excepción es la implementación del transcompilador de Lenguaje Fuente Origen a Lenguaje Intermedio que utiliza Lex y Yacc para simplificar la incorporación de nuevos lenguajes a estructurar.

6. CONCLUSIONES Y DESARROLLOS FUTUROS

El lenguaje definido y presentado en este trabajo permite representar las sentencias de los cuatro lenguajes estudiados sin perder información relevante.

Debido a que el mismo posee todo el control implementado en su nivel inferior, es sumamente flexible y permite representar, no solamente las sentencias estudiadas, sino también otras sentencias con diferentes estructuras.

Además, el LI demostró ser adecuado para soportar las variaciones implantadas por la reestructuración realizada sobre el GDPE.

Como el objetivo final es mejorar la calidad de los programas, se está trabajando, actualmente, en la transformación de un programa sintácticamente estructurado, en un programa modular. Se ha desarrollado un primer diseño de un algoritmo que, utilizando la técnica de "slicing" de programas, permite dividir un programa monolítico sintácticamente estructurado en un conjunto de módulos.

Además, se está investigando el cambio de paradigma de programación; en particular, la migración de programas implementados en lenguajes imperativos a lenguajes orientados a objetos. Se han definido un conjunto de reglas para reconocer y extraer objetos desde código fuente de programas escritos en lenguajes imperativos, a partir de los tipos y variables que los mismos poseen.

Ejemplo de las Etapas de Transformación

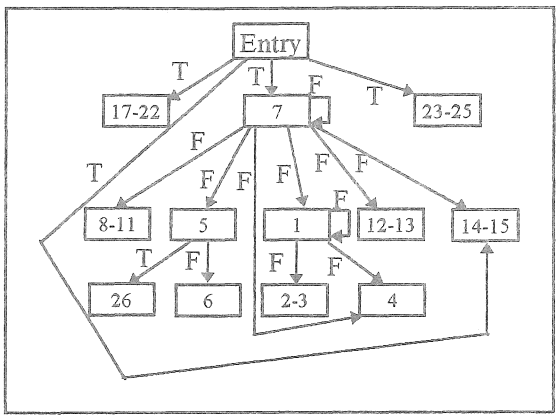
```
PRINCIPAL.
  OPEN INPUT CONTABLE.
  OPEN OUTPUT AJUSTE.
LEER-PROXIMA-CUENTA.
  READ CONTABLE RECORD INTO REG-CONT
     AT END GOTO TERMINAR.
  MOVE NRO-CTA IN REG-CONT TO NRO-CTA IN REG-AJ.
  MOVE BALANCE IN REG-CONT TO BALANCE IN REG-AJ.
  MOVE PREST-ACT IN REG-CONT TO PREST-ACT IN REG-AJ.
  MOVE PREST-MAX IN REG-CONT TO PREST-MAX IN REG-AJ.
  IF BALANCE IN REG-AJ < 0.00
    MOVE 'S' TO MOROSO
  ELSE
    MOVE 'N' TO MOROSO.
MAS.
  IF (BALANCE IN REG-AJ IS NOT LESS THAN 0.00) OR
    (PREST-ACT IN REG-AJ + 100.00 > PREST-MAX IN REG-AJ)
    GOTO ACT-CUENTA.
  ADD 100.00 TO PREST-ACT IN REG-AJ.
  ADD 100.00 TO BALANCE IN REG-AJ.
  GOTO MAS.
ACT-CUENTA.
  WRITE REG-AJ.
  GOTO LEER-PROXIMA-CUENTA.
TERMINAR.
  CLOSE CONTABLE , AJUSTE.
  STOP RUN.
```

Código Cobol desestructurado [5]

4.1

REPRESENTACIÓN INICIAL DEL CÓDIGO EN LENGUAJE INTERMEDIO (página siguiente)

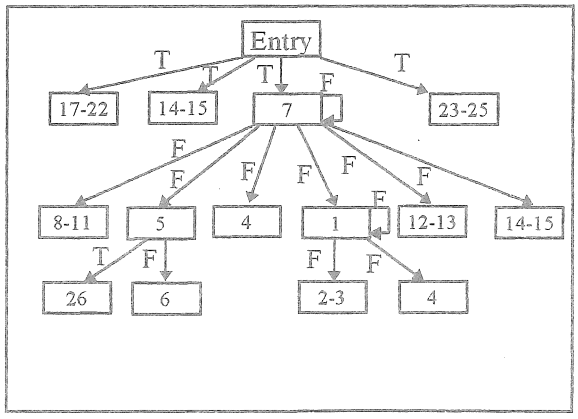
4.2



GDPE desestructurado

(se agruparon nodos y obviaron arcos de secuencia)

4.3



GDPE estructurado

4.4

NUEVA REPRESENTACIÓN DEL CÓDIGO EN LENGUAJE INTERMEDIO

4.5

```
Program Principal()
. . .
Begin
   Assign(CONTABLE,"CONTABLE.DAT");
    Reset(CONTABLE);
   Assign(AJUSTE,"AJUSTE.DAT");
   Reset(AJUSTE);
   Read(CONTABLE,REG-CONT);
   While not eof (CONTABLE) do Begin
      REG-AJ.NRO-CTA:= REG-CONT.NRO-CTA;
      REG-AJ.BALANCE:= REG-CONT.BALANCE;
      REG-AJ.PREST-ACT:= REG-CONT.PREST-ACT;
      REG-AJ.PREST-MAX:= REG-CONT.PREST-MAX;
      If REG-AJ.BALANCE < 0.00 Then
        MOROSO: = 'S'
       Else
        MOROSO: = 'N';
      While not(not(REG-AJ.BALANCE < 0.00) or
       (REG-AJ.PREST-ACT + 100 > REG-AJ.PREST-MAX )) do
      Begin
        REG-AJ.PREST-ACT := REG-AJ.PREST-ACT + 100;
        REG-AJ.BALANCE : = REG-AJ.BALANCE+100
      End;
      Write(AJUSTE,REG-AJ);
       Read(CONTABLE,REG-CONT)
   End;
   Close(CONTABLE);
   Close(AJUSTE);
End.
```

Código Pascal estructurado

Nota: El número dentro de cada flecha identifica la transformación

Representación inicial del código en lenguaje intermedio

```
sentencia(1,sent_bloque(subrutina,[2],[3,6]),"PRINCIPAL.",-1)
sentencia(2,subrutina(ident("PRINCIPAL"),[]),"",1)
sentencia(3,sent_bloque(abrir,[4,5],[]),"\n  OPEN INPUT CONTABLE.",-1)
sentencia(4,tipo_arch(ident("CONTABLE"),r,[]),"",3)
sentencia(5,posicion(ident("CONTABLE"),numero("0")),"",3)
sentencia(6,sent_bloque(abrir,[7,8],[]),"\n  OPEN OUTPUT AJUSTE.",-1)
sentencia(7,tipo_arch(ident("AJUSTE"),w,[]),"",6)
sentencia(8,posicion(ident("AJUSTE"),numero("0")),"",6)
sentencia(9,sent_bloque(subrutina,[10],[11,15,16,17,18,19]),"\nLEER-PROXIMA-CUENTA.",-1)
sentencia(10,subrutina(ident("LEER-PROXIMA-CUENTA"),[]),"",9)
sentencia(11,sent_bloque(excepcion,[12,13],[14]),"\n  READ CONTABLE RECORD INTO REG-CONT\n       AT END",-1)
sentencia(12,leer("READ",ident("CONTABLE"),[ident("REG-CONT")],[]),"",11)
sentencia(13,cond(logico(eof,ident("REG-CONT"),vacio),etiq_interna(14),etiq_interna(15)),"",11)
sentencia(14,goto(etiq_interna(37))," GOTO TERMINAR.",-1)
sentencia(15,asig([registro(ident("NRO-CTA"),ident("REG-CONT"))], [registro(ident("NRO-CTA"),ident("REG-AJ"))]),
"\n  MOVE NRO-CTA IN REG-CONT TO NRO-CTA IN REG-AJ.",-1)
sentencia(16,asig([registro(ident("BALANCE"),ident("REG-CONT"))], [registro(ident("BALANCE"),ident("REG-AJ"))]),
"\n  MOVE BALANCE IN REG-CONT TO BALANCE IN REG-AJ.",-1)
sentencia(17,asig([registro(ident("PREST-ACT"),ident("REG-CONT"))],[registro(ident("PREST-ACT"),ident("REG-AJ"))]),
"\n  MOVE PREST-ACT IN REG-CONT TO PREST-ACT IN REG-AJ.",-1)
sentencia(18,asig([registro(ident("PREST-MAX"),ident("REG-CONT"))],[registro(ident("PREST-MAX"),ident("REG-AJ"))]),
"\n  MOVE PREST-MAX IN REG-CONT TO PREST-MAX IN REG-AJ.",-1)
sentencia(19,sent_bloque(cond,[20,22],[21,23]),"\n  IF BALANCE IN REG-AJ < 0.00\n     ",-1)
sentencia(20,cond(relacional(menor,registro(ident("BALANCE"),ident("REG-AJ")), numero("0.00")),etiq_interna(21),
etiq_interna(23)),"",19)
sentencia(21,asig([texto("'S'")],[ident("MOROSO")]),"MOVE 'S' TO  MOROSO\n  ",-1)
sentencia(22,ira(etiq_interna(25)),"ELSE",19)
sentencia(23,asig([texto("'N'")],[ident("MOROSO")]),"\n     MOVE 'N' TO MOROSO.",-1)
sentencia(24,sent_bloque(subrutina,[25],[26,29,30,31]),"\nMAS.",-1)
sentencia(25,subrutina(ident("MAS"),[]),"",24)
sentencia(26,sent_bloque(cond,[27],[28]),"\n  IF (BALANCE IN REG-AJ IS NOT LESS THAN 0.00) OR\n     (PREST-ACT IN
REG-AJ + 100.00 > PREST-MAX IN REG-AJ)\n     ",-1)
sentencia(27,cond(logico(o,asoc(logico(no,relacional(menor,registro(ident("BALANCE"),ident("REG-AJ")),numero("0.00")),
vacio)),asoc(relacional(mayor,operacion(suma, registro(ident("PREST-ACT"),ident("REG-AJ")),numero("100.00")),
registro(ident("PREST-MAX"), ident("REG-AJ")))))),etiq_interna(28),etiq_interna(29)),"",26)
sentencia(28,goto(etiq_interna(33)),"GOTO ACT-CUENTA.",-1)
sentencia(29,asig([operacion(suma,numero("100.00"),registro(ident("PREST-ACT"), ident("REG-AJ")))],[registro(ident
("PREST-ACT"),ident("REG-AJ"))]),"\n  ADD 100.00 TO PREST-ACT IN REG-AJ.",-1)
sentencia(30,asig([operacion(suma,numero("100.00"),registro(ident("BALANCE"),ident("REG-AJ")))],
[registro(ident("BALANCE"),ident("REG-AJ"))]),"\n  ADD 100.00 TO BALANCE IN REG-AJ.",-1)
sentencia(31,goto(etiq_interna(25)),"\n  GOTO MAS.",-1)
sentencia(32,sent_bloque(subrutina,[33],[34,35]),"\nACT-CUENTA.",-1)
sentencia(33,subrutina(ident("ACT-CUENTA"),[]),"",32)
sentencia(34,escribir("WRITE",vacio,[ident("REG-AJ")],[vacio]),"\n  WRITE REG-AJ.",-1)
sentencia(35,goto(etiq_interna(10)),"\n  GOTO LEER-PROXIMA-CUENTA.",-1)
sentencia(36,sent_bloque(subrutina,[37],[38,41]),"\nTERMINAR.",-1)
sentencia(37,subrutina(ident("TERMINAR"),[]),"",36)
sentencia(38,sent_bloque(cerrar_archivo,[39,40],[]),"\n  CLOSE CONTABLE, AJUSTE.",-1)
sentencia(39,cerrar_arch(ident("CONTABLE")),"",38)
sentencia(40,cerrar_arch(ident("AJUSTE")),"",38)
sentencia(41,stoprun(etiq_interna(42)),"\n  STOP RUN.",-1)
sentencia(42,maxima_sentencia(cobol),"",-1)
```

REFERENCIAS

[1] Aho,A.,Sethi R., Ullman J., Compilers: Principles, Techniques, and Tools, 1986.
[2] Arnold, R., Software Reengineering, 2nd edition, IEEE Computer Society Press, 1994.
[3] Cobo, H., Mauco, M. V., Un algoritmo para reestructurar programas procedurales, Memorias Conf. Latin. de Informática, CLEI'96, Volumen 2, 979-990, 1996.
[4] Ferrante, J., Ottenstein, K., Warren, J., The Program Dependence Graph and its Use in Optimization, ACM Transactions on Programming Languages and Systems, Vol. 9, N° 3, 319-349, 1987.
[5] Hausler P. , Pleszkoch M. , Linger R. , Hevner A., Using Function Abstraction to Understand Program Behavior, IEEE Software, January 1990.
[6] Horwitz, S., Prins, J, Reps, T., On the Adequacy of Program Dependence Graphs for Representing Programs, 15th ACM Symp. on Principles of Programming Languages, 12 pages, Jan 1988.
[7] Horwitz, S., Reps, T., The Use of Program Dependence Graphs in Software Engineering, 14th International Conference on Software Engineering, 20 pages, 1992.
[8] Linger, R., Mills, H., Witt B., Structured Programming: Theory and Practice, Addison-Wesley Publishing Company, Cambridge, Mass., 1979.
[9] Nosek, J., Prashant, P. Software Maintenance Management: The Change in the Last Decade, Journal of Software Maintenance Research and Practice, Vol 2, N° 3, 157-174, 1990.
[10] Ottenstein, K., Ottenstein, L., The Program Dependence Graph in a Software Development Environment, ACM SIGPLAN Notices, Vol. 19, N° 5, 177-184, 1984.
[11] Pressman, R., Ingeniería del Software: Un Enfoque Práctico, 3° edición, Mc Graw Hill, 1993.
[12] Rugaber, S., Clayton, R., The Representation Problem in Reverse Engineering, Technical Report, Georgia Institute of Technology, 1993