Te
H71
1987

UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ciencias y Humanidades

DISEÑO E IMPLEMENTACION DE UN LENGUAJE DE CUARTA GENERACION

RODRIGO ARIAS PALOMO

BIBLIOTECA
DE LA
UNIVERSIDAD DEL VALLE DE GUATEMALA

Guatemala

1987

DISEÑO E IMPLEMENTACION DE UN LENGUAJE DE CUARTA GENERACION

UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ciencias y Humanidades

DISEÑO E IMPLEMENTACION DE UN LENGUAJE DE CUARTA GENERACION

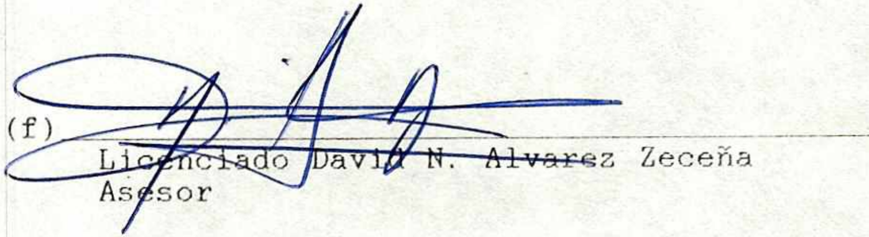
RODRIGO ARIAS PALOMO

Trabajo de investigación presentado para optar
al grado académico de
Licenciado en Ciencias de la Computación

Guatemala

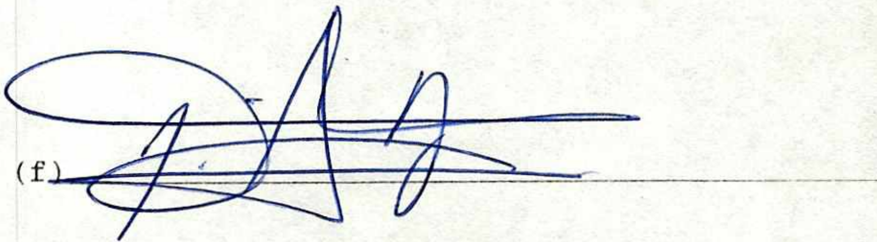
1987

Vo. Bo. :



(f) ______________________________
Licenciado David N. Alvarez Zeceña
Asesor

Tribunal:



(f) ______________________________

(f) ______________________________

(f) ______________________________

Fecha de aprobación:

A mis padres

PREFACIO

Una gran parte del trabajo realizado en Ciencias de la Computación se consume en el desarrollo de sistemas administrativos. Pero no fue sino hasta esta década que se empezó a generalizar el uso de herramientas poderosas destinadas a aliviar sensiblemente el trabajo. Entre estas herramientas los lenguajes diseñados especificamente para implementar sistemas administrativos poseen un lugar preponderante por su flexibilidad y alto nivel de abstracción. Dichos lenguajes son denominados comunmente como: Lenguajes de Cuarta Generación.

Cualquier administrador de centro de cómputo que haya emigrado a un lenguaje de 4a. generación ha sentido una importante reducción en el esfuerzo destinado a la codificación y al mantenimiendo. Lo que redunda en un significativo ahorro.

Por su relación costo - beneficio la computadora AM-1000 de Alpha-Micro es una alternativa rentable para empresas medianas y pequeñas, pero no posee una herramienta económica que permita el desarrollo eficiente de aplicaciones comerciales. Por esta razón se escogió como proyecto de graduación desarrollar un leguaje de 4a. generación para dicha máquina.

Agradecimiento

Este proyecto nunca hubiera podido realizarse sin la disponibilidad, durante varios meses, de una computadora AM-1000 con todos sus recursos: unidad de respaldo, impresora, etc. Por lo que agradezco especialmente al Ing. Jack Trachtenberg quien hizo ésto posible.

R.A.P.

CONTENIDO

PREFACIO IX

1. INTRODUCCION 1

2. DISEÑO 2

2.1. Objetivos 3

2.2. Lenguajes de cuarta generación existentes 5

2.3. Sintaxis y descripción semántica 6

2.3.1. Expresiones 6

2.3.2. Verbos para manejar archivos 6

2.3.3. Instrucciones de control de flujo 9

2.3.4. Definición e invocación de módulos 12

2.3.5. Definición de variables y parámetros 13

2.4. Especificación semántica 15

2.4.1. Notación 15

2.4.2. Verbos para manejar archivos 15

2.4.3. Instrucciones de control de flujo 22

2.5. Evaluación del diseño 25

3. IMPLEMENTACION 28

3.1. Ambiente de desarrollo 29

3.2. Lenguaje objeto 31

3.3. 'Bootstraping' 32

3.4. Estructura de Lest y Assist 34

3.5. Aspectos técnicos importantes 36

3.5.1. Compilación en dos pasadas 36

3.5.2. Reconocimiento y generación de código 36

3.5.3. Evaluación de eficiencia de las estructuras de datos 37

3.6. Implementación del ´parser´ predictivo 39

4. CONSIDERACIONES FINALES 43

5. BIBLIOGRAFIA 44

APENDICES

A. Conveción de símbolos en notación sintáctica 45

B. Sintaxis de la especificación fuente del diccionario 47

C. Código fuente de Sub-Lest

D. Código fuente de Lest

E. Código fuente de Assist

1. INTRODUCCION

En este informe se detallan los aspectos sobresalientes involucrados en el desarrollo de un lenguaje de cuarta generación llamado Assist. Como cualquier desarrollo de software: se inició con una fase de diseño o especificación para posteriormente implementarse. Se pensó que organizar este informe en el mismo orden cronológico del desarrollo de Assist era lo más conveniente. Por lo tanto, está dividido en dos capítulos: el primero se refiere al diseño del lenguaje y el otro describe las técnicas y los criterios de Ciencias de la Computación que permitieron una eficiente implementación de un compilador de Assist.

Si se desea obtener información sobre conceptos tales como: 4a. generación, compiladores, gramática LL(1), ´bootstraping´, etc. Se incluyeron las referencias adecuadas en la bibliografía, ya que por razones de espacio dichas definiciones no fueron transcritas a este informe.

2. DISEÑO

En la actualidad no existe una metodología adecuada que permita sistemáticamente llegar a especificar, sintáctica y semánticamente, un nuevo lenguaje a partir de los objetivos del diseñador. Es, por tanto, un trabajo creativo basado principalmente en la experiencia.

Esto no quiere decir que el diseño de un lenguaje no deba llevar orden alguno. Es conveniente poseer un marco de trabajo que permita concretizar en la especificación del lenguaje los objetivos planteados por el diseñador. Dicho marco de trabajo fue el siguiente:

1. Planteo de objetivos o características a poseer por el lenguaje.
2. Recopilación de información sobre los lenguajes de 4a. generación más exitosos.
3. Diseño de las instrucciones y descripción informal de su semántica
4. Evaluación del diseño: coherencia, cumplimiento de objetivos, etc.
5. Especificación semántica detallada de las instrucciones.

Este capítulo comprenderá: el planteo de los objetivos originales, una breve evaluación de los lenguajes de 4a. generación, especificación de la sintaxis y la semántica final y su evaluación con respecto a los objetivos.

2.1 OBJETIVOS

A continuación se decriben brevemente los objetivos o característica que, se desea, debe poseer el lenguaje. Por su diversidad no fue posible agrupar los objetivos para una mejor comprensión. Pero se intentó enunciarlos del más general al más específico.

1. Lenguaje de 4a. generación:

Assist deberá poseer las ventajas ofrecidas por los lenguajes de 4a. generación [8]:

- Velocidad de desarrollo en aplicaciones comerciales
- Facilidad de mantenimiento de programas
- Alto nivel de abstracción en el manejo de archivos
- Independencia programación - organización de los datos
- Curva de aprendizaje corta.

2. Diccionario:

La definición lógica de los archivos y sus componentes (i.e. tipo, campos, llaves, etc.) deberá ser única, independiente y externa a los programas.

3. Verbos compactos para el manejo de archivos:

Deberán ser suficientes cuatro verbos o instrucciones para: insertar, leer, modificar y remover registros de los archivos, independientemente de la organización de éstos [12].

4. Instrucciones estructuradas de control de flujo:

Debe poseer, por lo menos, instrucciones equivalentes a las de Pascal: for, while, repeat, if, case [13].

5. Modularización:

Es indispensable que el lenguaje posea las reglas necesarias para definir áreas de datos locales. Permitiendo, así, desarrollar bajo el concepto de ´hidding´ de información [9].

6. Curva de aprendizaje corta:

Debe permitir que una persona familiarizada con el sistema operativo AMOS pueda aprender a programar en Assist en más o menos una semana.

7. Compatibilidad con aplicaciones desarrolladas en Alpha-Basic:

Permitiendo una migración suave de Alpha-Basic a Assist o, sencillamente, desarrollar los nuevos procesos en Assist sin necesidad de rehacer los anteriores [4].

8. Incremento significativo en el poder de los archivos indexados:

Debe permitir:
- manejo transparente de llaves duplicadas
- expresiones de campos como llaves
- manejo transparente de cualquier cantidad de archivos de llaves [11].

2.2 LENGUAJES DE CUARTA GENERACION EXISTENTES

Se estudiaron las sintaxis de los siguientes lenguajes de 4a. generación:

- Mantis
- Transact
- Dbase III
- Informix

Se hizo una evaluación sobre la facilidad de desarrollo en cada uno de estos lenguajes. Básandose principalmente en la experiencia previa y en artículos dedicados a la comparación de lenguajes de 4a. generación.

Se encontraron especialmente sencillos de aprender: Mantis y Dbase III a la vez que Dbase III se consideró como el de mayor facilidad y versatilidad para desarrollar.

Los verbos poderosos de Transact (i.e. FIND [14]) se consideraron como interesantes estructuras de control, pero su misma diferencia ocasionan un período de entrenamiento mayor y pueden ser reemplazados por tres o cuatro instrucciones de cualquier otro lenguaje de 4a. generación.

En conclusión, se encontró conveniente diseñar un lenguaje estructuralmente similar a Dbase III o Mantis. La notación de las instrucciones de control de flujo estan basadas en un lenguaje para diseñar procesos llamado PDL [7]. A pesar de lo anterior, todas las instrucciones fueron rediseñadas para acoplarse al ambiente de AMOS y se innovó en otras tales como DO - ENDDO.

2.3 SINTAXIS Y DESCRIPCION SEMANTICA

2.3.1 EXPRESIONES

La sintaxis de las expresiones de Assist es igual a la de las expresiones en Alpha-Basic [4], con las siguientes excepciones:

- El apóstrofe (´) usado en los identificadores de Alpha-Basic se reemplaza por el guión (_) en Assist.
- No se permiten dos guiones seguidos en los identificadores de Assist.
- Los nombres de los campos poseen idénticas propiedades que las variables globales, se denotan así:

    <nombre del archivo>.<campo>

2.3.2 VERBOS PARA MANEJAR ARCHIVOS

USE:

sintaxis (ver Apéndice A)

```
USE <archivo> { EXCLUSIVE } [ FOR [ OUTPUT ] ]
              {           } [     [ INPUT  ] ]
              { SHARED    } [     [ APPEND ] ]
```

semántica

- abre el archivo.
- define sus campos.
- si es indexado, define sus llaves y los archivos que las poseen.
- la cláusula FOR indica el modo de apertura en caso de ser secuencial el archivo.

GET:

sintaxis

```
GET <archivo> [ CURRENT                               ]
              [                                       ]
              [ WITH  [ REC     ] = <expresión>       ]
              [       [         ]                     ]
              [       [ <llave> ]                     ]
              [                                       ]
              [ NEXT  [ <llave> ]                     ]
```

semántica

Obtiene un registro de <archivo>.

- CURRENT: lee el mismo registro (indexado y random).
- WITH REC = ...: lee el registro numero <expresión>.
- WITH <llave> = ...: accesa el registro con <llave> igual a <expresión>.
- NEXT y secuencial: accesa el siguiente registro físico.
- NEXT e indexado: accesa el siguiente registro según llave primaria.
- NEXT <llave>: accesa el siguiente registro según <llave>.

UPDATE:

sintaxis

```
UPDATE <archivo>
```

semántica

Modifica los campos de un archivo random o indexado.

Si fueron alterados campos que conforman llaves, éstas son ajustadas automáticamente.

PUT:

sintaxis

```
PUT <archivo> [ INTO REC = <expresión> ]
```

semántica

Graba un registro en el archivo. Si es indexado crea automáticamente todas sus llaves. En caso de ser random, la cláusula INTO REC... indica la posición física donde debe ir el registro.

DELETE:

sintaxis

```
DELETE <archivo> [ CURRENT                    ]
                 [                            ]
                 [ WITH <llave> = <expresión> ]
```

semántica

Sólo está definido para archivos indexados. Si se usa CURRENT, entonces remueve el último registro accesado. Si WITH <llave>..., localiza el registro cuya <llave> sea igual a <expresión> y lo remueve. Si <llave> admite duplicaciones, entonces regresa un mensaje de error.

CLOSE:

sintaxis

CLOSE <archivo>

semántica

Es opcional, su principal uso es para volver a leer archivos secuenciales. Si <archivo> es indexado, cierra todos sus índices.

2.3.3 INSTRUCCIONES DE CONTROL DE FLUJO (ver Apéndice A)

IF:

sintaxis

```
   IF <condición>
      <código 1>
 [ ELSE
      <código 2> ]
   ENDIF
```

semántica

si la <condición> evalua a verdadero se ejecuta <código 1> si evalua a falso se ejecuta <código 2>.

CASE:

sintaxis

```
   DO CASE
      CASE <condición>
         <código>
   { CASE <condición>
```

<código> }

[OTHERWISE

<código>]

ENDDO

semántica

Se ejecuta el <código> de la <condición> que evalue a verdadero. Si ninguna <condición> evalua a verdadero y está OTHERWISE, el <código> de éste es evaluado.

DO:

sintaxis

DO [<criterio de iteración>]

<código>

ENDDO [<criterio de iteración>]

criterios de iteración:

WHILE <condición>

mientras <condición> evalue a verdadero se ejecuta el ´loop´.

UNTIL <condición>

mientras <condición> evalue a falso se ejecuta el ´loop´.

semántica

Si <criterio de iteración> aparece a la par del DO, éste es verificado antes de ejecutar <código>. Si aparece a la par del ENDDO, <código> es ejecutado una vez, antes de verificar <criterio de iteración>.

FOR:

sintaxis

```
DO FOR <variable> = <expresión 1> TO <expresión 2>
     [ STEP <expresión 3> ]
  <código>
ENDDO
```

semántica

Ejecuta <código> las veces que indique el criterio de iteración.

UNDO:

sintaxis

```
UNDO
```

semántica

Cuando se ejecuta sale del 'loop' más interno en que se encuentre, ejecutando la instrucción siguiente al ENDDO.

CYCLE:

sintaxis

```
CYCLE
```

semántica

Cuando se ejecuta regresa al DO del 'loop' más interno. Si posee criterio de iteración, éste es verificado antes de ejecutar <código>.

2.3.4 DEFINICION E INVOCACION DE MODULOS

MODULE:

sintaxis

```
MODULE <nombre>
[ IN <definición de parámetros de entrada> ]
[ OUT <definición de parámetros de salida> ]
[ LOCAL
        <definición de variables locales> ]
[ ENDDEF ]
  <código>
```

Nota:

ENDDEF debe ir únicamente si se incluye alguna de las opciones anteriores. Los módulos deben codificarse después del programa principal.

semántica

El paso de parámetros es por valor. Si el módulo desea regresar información, las variables que la contengan deben pertenecer al OUT. Los parámetros y las variables LOCALes no pueden ser accesadas fuera del <código> del módulo. El <código> del módulo queda delimitado únicamente por otra instrucción MODULE o por el fin de archivo.

CALL:

sintaxis

```
CALL <nombre> [ IN <lista de expresiones de entrada> ]
              [ OUT <lista de parámetros de salida> ]
```

semántica

Llama a <nombre>. Las expresiones de entrada son evaluadas y asignadas una a una a los parametros de entrada.

2.3.5 DEFINICION DE VARIABLES Y PARAMETROS

Las variables pueden definirse en los módulos y al principio del programa principal, así:

sintaxis

```
[ LOCAL
  { <definición de variable> } ]
[ GLOBAL
  { <definición de variable> } ]
[ ENDDEF ]
```

Nota:

ENDDEF debe ir únicamente si se incluye alguna de las opciones anteriores.

<definición de variable>:

```
[ <nombre> [(<lista de dimensiones>)] ,<tipo> [,<largo>] ]
[          [,<valor inicial>] [,<origen>]                ]
[ <record>                                               ]
```

<record>:

```
RECORD <nombre>
{ <definición de variable> }
ENDREC
```

semántica

Las variables LOCALes sólo pueden ser accesadas dentro del programa principal. Una variable GLOBAL puede ser accesada en cualquier módulo que no haya definido la misma variable como local. La estructura RECORD es semánticamente igual a los records construidos en Alpha-Basic [4].

2.4 ESPECIFICACION SEMANTICA

En la actualidad no existe una notación generalizada para especificar semánticamente un lenguaje. Sin embargo la definición por traducción [3], bajo ciertas circunstancias, puede producir una especificación relativamente compacta y exacta. Esta técnica consiste en definir las instrucciones de un lenguaje por medio de secuencias de instrucciones de otro lenguaje ya definido. La especificación semántica de Assist se enunció en Alpha-Basic [4].

2.4.1 NOTACION

las expresiones entre corchetes angulosos corresponden a los nombres asignados en la especificación sintáctica. Por ejemplo <código> en la especificación semántica, corresponde exactamente a la traducción de <código> de la especificación sintáctica. Se incluye libremente texto en español para indicar si ciertas líneas de código deben considerarse o no, o si se repiten varias veces. También se hace uso de las llaves y los corchetes para denotar repetición y opción respectivamente.

2.4.2 VERBOS PARA MANEJAR ARCHIVOS

USE:

si <archivo> es secuencial:

```
  OPEN #<f>,"<archivo>[.<extensión>]",<cláusula del for>
```

```
  S´´OPFILES(<n> = "O"


si <archivo> es random:

  si modo es exclusivo: <modo> = RANDOM

  si modo es compartido: <modo> = RANDOM´FORCED

  OPEN #<f>,"<archivo>[.<extension>]",<modo>,<largo registro>,
            S´´RN<f>

  S´´OPFILES(<f>) = "O"


si <archivo> es indexado:

  si modo es exclusivo: <modo> = INDEXED´EXCLUSIVE

  si modo es compartido: <modo> = INDEXED

  OPEN #<f>,"<archivo>",<modo>,<largo registro>,S´´RN<f>

  S´´OPFILES(<f>) = "O"

  para todos los archivos de llaves que posea:

    OPEN #<f+i>,"<archivo>",<modo>,<largo registro>,S´´RN<f>

    S´´OPFILES(<f+i>) = "O"



GET:

si <archivo> es secuencial y tipo = input:

   INPUT LINE #<f>,<archivo>

   S´´ERR´C = <f>

   CALL S´´SEQ´ERROR


si <archivo> es random:

   si cláusula = CURRENT:

     READ #<f>,<archivo>
```

```
  si cláusula = WITH REC...:
    S''RN<f> = <expresión>
    READ #<f>,<archivo>
  si cláusula = NEXT:
    S''RN<f> = S''RN<f> + 1
    READ #<f>,<archivo>

si <archivo> es indexado:
  si cláusula = CURRENT:
    READ #<f>,<archivo>
  si cláusula = WITH REC... y no acepta duplicados:
    ISAM #<f>,1,<expresión>
    S''ERR'C = <f>
    S''ERR'OP = 1
    CALL S''ISAM'ERROR
    IF NOT S''ERR'ABORT THEN READ #<f>,<archivo>
  si cláusula = WITH REC ... y acepta duplicados:
    U = <expresión>
    ISAM #<f>,1,U
    S''ERR'C = <f>
    S''ERR'OP = 12
    CALL S''ISAM'ERROR
    IF NOT S''ERR'ABORT THEN READ #<f>,<archivo> : &
      IF <llave> # U THEN G''STATUS = "NOFOUND"
  si cláusula = NEXT:
    ISAM #<fi>,2,S''DUMMY
    S''ERR'C = <f>
```

```
S''ERR'OP = 2
CALL S''ISAM'ERROR
IF NOT S''ERR'ABORT THEN READ #<f>,<archivo>
```

```
UPDATE:
si <archivo> es random:
  WRITE #<f>,<archivo>
si <archivo> es indexado:
  READ #<f>,U''<archivo>
  WRITE #<f>,<archivo>
  para cada llave de <archivo>:
    IF <llave> = <llave anterior> THEN GOTO S''LABEL<n>
    si la llave admite duplicados:
      S''REC'B = S''RN<f>
      ISAM #<fi>,4,<llave anterior>+S''REC'S
      S''ERR'OP = 4
      S''ERR'C = <fi>
      CALL S''ISAM'ERROR
      IF S''ERR'ABORT THEN GOTO S''EXIT<m>
      ISAM #<fi>,3,<llave>+R''REC'S
      S''ERR'OP = 3
      S''ERR'C = <fi>
      CALL S''ISAM'ERROR
      IF S''ERR'ABORT THEN GOTO S''EXIT<m>
    si no admite duplicados:
      ISAM #<fi>,4,<llave anterior>
      S''ERR'OP = 4
```

```
      S''ERR'C = <fi>
      CALL S''ISAM'ERROR
      IF S''ERR'ABORT THEN GOTO S''EXIT<m>
      ISAM #<fi>,3,<llave>
      S''ERR'OP = 3
      S''ERR'C = <fi>
      CALL S''ISAM'ERROR
      IF S''ERR'ABORT THEN GOTO S''EXIT<m>
    S''LABEL<n>:
  S''EXIT<n>:
```

PUT:

si <archivo> es secuencial y modo = output:

```
  PRINT #<f>,<archivo>
```

si <archivo> es random:

```
  si clausula = INTO ...:
    S''RN<f> = <expresión>
  WRITE #<f>,<archivo>
```

si <archivo> es indexado:

```
  para cada llave que no admite duplicados:
    ISAM #<fi>,1,<llave>
    S''ERR'OP = 11
    S''ERR'C = <fi>
    CALL S''ISAM'ERROR
```

```
    IF S´´ERR´ABORT THEN GOTO S´´LABEL<n>

  ISAM #<f>,5,S´´DUMMY
  S´´ERR´OP = 5
  S´´ERR´C = <f>
  CALL S´´ISAM´ERROR
  IF S´´ERR´ABORT THEN GOTO S´´LABEL<n>
  WRITE #<f>,<archivo>
  para cada llave que no admite duplicados:
    ISAM #<fi>,3,<llave>
    S´´ERR´OP = 3
    S´´ERR´C = <fi>
    CALL S´´ISAM´ERROR
    IF S´´ERR´ABORT THEN GOTO S´´LABEL<n>
  para cada llave que admite duplicados:
    S´´REC´B = S´´RN<f>
    ISAM #<fi>,3,<llave>+S´´REC´S
    S´´ERR´OP = 3
    S´´ERR´C = <fi>
    CALL S´´ISAM´ERROR
    IF S´´ERR´ABORT THEN GOTO S´´LABEL<n>
  S´´LABEL<n>:


DELETE:
si <archivo> es indexado:
  si cláusula = WITH... y <llave> no admite duplicados:
    ISAM #<fi>,1,<llave>
```

```
  S''ERR'OP = 1
  S''ERR'C = <fi>
  CALL S''ISAM'ERROR
  IF S''ERR'ABORT THEN GOTO S''EXIT<n>
READ #<f>,<archivo>
para cada llave de <archivo>:
  si la llave no admite duplicados:
    ISAM #<fi>,4,<llave>
    S''ERR'OP = 4
    S''ERR'C = <fi>
    CALL S''ISAM'ERROR
    IF S''ERR'ABORT THEN GOTO S''EXIT<n>
  si la llave admite duplicados:
    S''REC'B = S''RN<f>
    ISAM #<fi>,4,<llave>+S''REC'S
    S''ERR'OP = 4
    S''ERR'C = <fi>
    CALL S''ISAM'ERROR
    IF S''ERR'ABORT THEN GOTO S''EXIT<n>
ISAM #<fi>,6,S''DUMMY
S''ERR'OP = 6
S''ERR'C = <f>
CALL S''ISAM'ERROR
S''EXIT<n>:
```

CLOSE:

si <archivo> es secuencial o random:

```
  CLOSE #<f>
  S''OPFILES(<f>) = "C"
si <archivo> es indexado:
  CLOSE #<f>
  S''OPFILES(<f>) = "C"
  para todos los archivos de llaves secundarias:
    CLOSE #<fi>
    S''OPFILES(<fi>) = "C"
```

2.4.3 INTRUCCIONES DE CONTROL DE FLUJO

IF:

```
      IF NOT <condición> THEN GOTO else<n>
        <código>
      GOTO endif<n>
else<n>: [ <código>
endif<n>: ]
```

CASE:

```
                  IF NOT <condición> then goto case<n>'<m>
                    <código>
                  GOTO endcase<n>
 { case<n>'<m>:   IF NOT <condición> THEN GOTO case<n>'<m+i>
                    <código>
                  GOTO endcase<n> }
  case<n>'<m+i>: [ <código del otherwise> ]
```

```
  endcase<n>:


DO:
  do<n>: [ IF <criterio de iteración> THEN GOTO enddo<n> ]
              <código>
          [ IF <criterio de iteración> THEN GOTO enddo<n> ]
            GOTO do<n>
  enddo<n>:


<criterio de iteración>:
    WHILE <condición>
        NOT <condición>
    UNTIL <condición>
        <condición>


FOR:
    FOR <criterio de iteración>
        <código>
    NEXT <variable>
enddo<n>:


UNDO:
   GOTO enddo<n>


CYCLE:
   si se encuentra detro de un DO <criterio...:
     GOTO do<n>
```

si se encuentra dentro de un DO FOR...:

NEXT <variable>

2.5 EVALUACION DEL DISENO

Este capítulo quedaría inconcluso si no se evaluara cómo se concretizan en el actual diseño los requerimientos abstractos planteados en los objetivos. A continuación los ocho objetivos iniciales son enunciados, ahora del más específico al más general (en orden inverso) y se establece cómo, a juicio del diseñador, dichos objetivos son satisfechos.

8. Incremento significativo en el poder de los archivos indexados:

Los verbos: USE, GET, PUT, UPDATE, DELETE y CLOSE son sintácticamente idénticos, independientemente de si:

- <llave> acepta duplicados, y/o
- <archivo> posee una o más <llave>s.

Lo que, por definición, hace transparente su manejo. Además, todas las <llave>s de un <archivo> son ajustadas automáticamente.

En Assist, las expresiones de campos pueden ser llaves y deben declararse en el diccionario (ver Apendice B). Los verbos como PUT y UPDATE hace uso de dicha declaración para construir las llaves de los archivos indexados.

7. Compatibilidad con aplicaciones desarrolladas en Alpha-Basic:

Assist maneja la misma organización de archivos que Alpha-Basic y no hay ningún inconveniente en que compartan

BIBLIOTECA
DE LA
UNIVERSIDAD DEL VALLE DE GUATEMALA

el acceso de éstos. Esto permite: desarrollar los nuevos procesos en Assist sin necesidad de rehacer los anteriores.

6. Curva de aprendizaje corta:

Este objetivo se logró:

- Reduciendo a cuatro los verbos para manejar la información de los archivos: independientemente de si el archivo es de acceso secuencial, directo o indexado. Esto reduce la cantidad de diferentes instrucciones a aprender.
- Al diseñar las expresiones de Assist prácticamente idénticas a las expresiones de Alpha-Basic. Favoreciendo a los programadores familiarizados con AMOS (el sistema operativo de Alpha-Micro).
- Haciendo Assist similar a otros lenguajes de 4a. generación.

5. Modularización:

Este objetivo se satisface concretamente con la estructura:

```
MODULE <nombre>
       .
       .
       .
   LOCAL
     { <definición de variable> }
   ENDDEF
```

Con la cual, las variables definidas sólo pueden accesarse dentro del módulo llamado <nombre>.

4. Instrucciones estructuradas de control de flujo:

Las instrucciones: DO FOR, DO, IF, DO CASE, UNDO y CYCLE permiten controlar el flujo y son estructuradas [7].

3. Verbos compactos para el manejo de archivos:

Los verbos PUT, GET, UPDATE y DELETE permiten, respectivamente: insertar, leer, modificar y remover registros de los archivos, independientemente de la organización de éstos (ver el objetivo correspondiente).

2. Diccionario:

En el Apéndice B se encuentra la sintaxis para especificar archivos, la cual es, efectivamente: única, independiente y externa a los programas.

1. Lenguaje de 4a. generación:

Por lo establecido anteriormente, Assist posee una curva de aprendizaje corta y un alto nivel de abstracción en el manejo de archivos (objetivos 6, 3 y 2). El alto nivel de abstracción implica el uso de una notación más compacta para denotar el mismo concepto mental, lo que permite una mayor velocidad de desarrollo en aplicaciones comerciales y facilita el mantenimiento.

3. IMPLEMENTACION

Una adecuada implementación de un lenguaje involucra la aplicación conjunta de técnicas y metodologías de muchas ramas de Ciencias de la Computación. Tales como: Ingeniería de Software [9], Teoría de Lenguajes [3], Estructura de Datos [1] y Análisis de Algoritmos [2]. Por ejemplo: en base a principios de Ingeniería de Software se seleccionó el ambiente de desarrollo, se diseño refinando por pasos y se mantuvo el ´hidding´ de la información entre los módulos. Con Teoría de Lenguajes, se construyó un ´parser´ recursivo-descendente predictivo de gramáticas LL(1), para validar sintácticamente las expresiones. Con Estructura de Datos se construyeron las diferentes tablas y ´stacks´ para almacenar la información de las entidades y con Análisis de Algoritmos se determinó la eficiencia de dichas estructuras para largos realísticos de ´inputs´.

En este capítulo se condensan los aspectos sobresalientes del proceso de implementación.

3.1 AMBIENTE DE DESARROLLO

Sobre el sistema operativo AMOS de Alpha-Micro no existe un ambiente de desarrollo de ´software´ organizado. Mas bien, ofrece al usuario varias herramientas de trabajo dispersas: procesadores de palabras, compiladores de varios lenguajes, un formateador de PDL, etc.

Seleccionar las herramientas a usar, se simplifica prácticamente a escoger el lenguaje de desarrollo. AMOS ofrece confiables compiladores o ensambladores de:

- Pascal
- Fortran
- Alpha-Basic
- Assembler M-68000

Conceptualmente, Pascal [5] sería el lenguaje más adecuado: permite modularizar con ´hidding´, puede procesar ´strings´ y posibilita la implementación de estructuras de datos complejas. El inconveniente de Pascal consiste en que el compilador traduce a código P, el cual es interpretado al ejecutar el programa. Aumentando el tiempo de ejecución a niveles inaceptables para escribir un compilador en él.

El principal inconveniente de Fortran, es su falta de estructuración y su deficiente manejo de ´strings´.

El compilador de Alpha-Basic [4] produce un código bastante rápido y el manejo de ´strings´ es especialmente poderoso. Su incapacidad de modularizar con ´hidding´ y su falta de estructuración, lo hacen un lenguaje prohibitivo para cualquier desarrollo grande de ´software´.

Desarrollar en un lenguaje de bajo-nivel produce una serie de inconvenientes [9] que redundan en un período muy largo de implementación, una mayor posibilidad de errores en el producto final y un mantenimiento más complejo.

En conclusión: un lenguaje estructurado que permita modularizar con ´hidding´ (como Pascal), que sea poderoso en el manejo de ´strings´ y su compilador produzca código eficiente (como Alpha-Basic) sería la solución.

Para resolver el problema anterior, se decidió desarrollar el compilador de Assist con ´bootstraping´ [3]. o sea: se seleccionó un subconjunto de instrucciones de Assist, las cuales constituyen en sí un nuevo lenguaje (llamado Lest), se implementaron y en éllas se codificó de nuevo Lest y posteriormente Assist. Esta técnica posee las siguientes ventajas:

- Se desarrolla en un leguaje que reúne las ventajas de Alpha-Basic y Pascal
- Escribir un compilador en un subconjunto de las instrucciones de Assist es una excelente oportunidad para su evaluación y ajuste.
- Como subproducto del desarrollo de Assist queda una nueva herramienta: Lest. Especialmente adecuada para la implementación de ´software´ extenso.

3.2 LENGUAJE OBJETO

En todo compilador están presentes tres lenguajes: El lenguaje a compilar, en el que está escrito el compilador y el lenguaje producido por el compilador o lenguaje objeto [3]. Hasta aquí ya fueron determinados los dos primeros, pero todavía no se ha decidido sobre el último. Las alternativas son:

- Lenguaje máquina
- ´Assembler´
- Alpha-Basic (y posteriormente compilarse)

Se encontró que traducir un lenguaje de 4a. generación a lenguaje máquina o a ´assembler´ redundaría en un período muy largo de implementación, si se nota que aún el equivalente semántico en Alpha-Basic es extenso (sección 2.4.2). Adicionalmente, se prevee una importante economía de esfuerzo si el lenguaje objeto es Alpha-Basic: la especificación semántica ya está en Alpha-Basic y las expresiones en Assist son muy similares a las expresiones en Alpha-Basic, lo que simplifica su traducción.

3.3 'BOOTSTRAPING'

Como se dijo anteriormente, un compilador esta caracterizado por tres lenguajes, denotados en esta sección como una tupla de tres posiciones:

(<lenguaje fuente>,<lenguaje objeto>,<lenguaje compilador>)

Por ejemplo: el actual compilador de Alpha-Basic está escrito en 'assembler' y traduce a lenguaje máquina. Por lo tanto su tupla es:

AB = (Alpha-Basic,M68000,Assembler)

A continuación se describirán los pasos de 'bootstraping' realizados para la obtención final de un compilador de Assist en Lest.

1. SL = (Sub-Lest,Alpha-Basic,Alpha-Basic)
2. SL' = (Sub-Lest',Sub-Lest,Sub-Lest)
3. L = (Lest,Alpha-Basic,Sub-Lest')
4. Lf = (Lest,Alpha-Basic,Lest)
5. A = (Assist,Alpha-Basic,Lest)

El paso 1 indica que se escribió en Alpha-Basic un traductor de un subconjunto de instrucciones de Lest (Sub-Lest) a Alpha-Basic. Esto significa que después de completado el paso 1 ya se pueden ejecutar programas escritos en Sub-Lest, porque con SL se traduce a Alpha-Basic y después a M68000 con AB. Los siguientes pasos, de igual forma, fueron aumentando la cantidad de instrucciones hasta la obtención de Lest. En el cual fue codificado Assist. Por ejemplo, el primer compilador de Lest se escribió en Sub-Lest' (paso 3) y para compilarlo a M68000 se

aplicaron sucesivamente: SL´,SL,AB. A partir de ese momento cualquier programa en Lest es ejecutable después de aplicar: L y AB.

El proceso de ´bootstraping´ puede describirse gráficamente utilizando el siguente elemento fundamental:

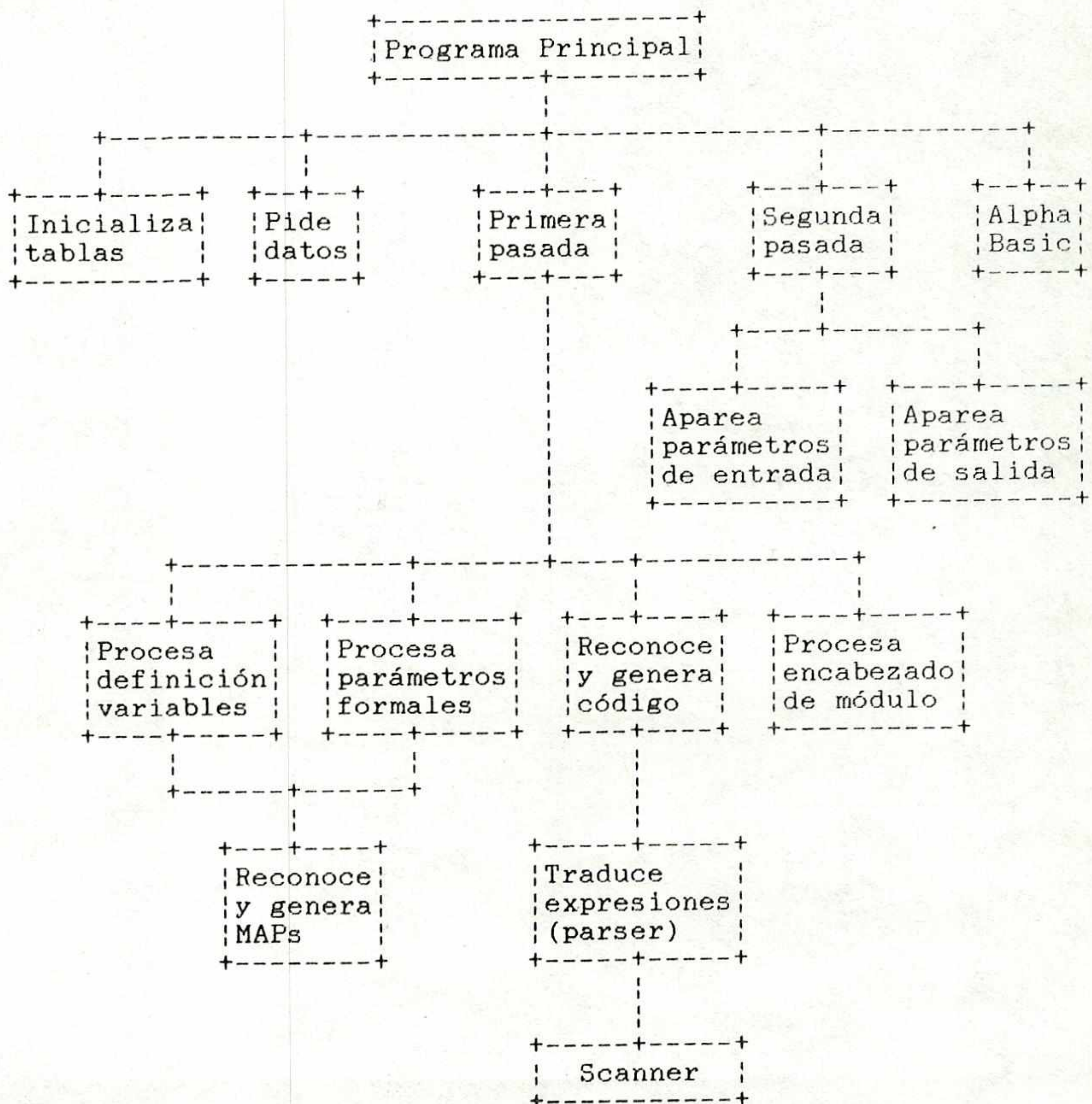
```
<lenguaje fuente> -> [<compilador>] -> <lenguaje objeto>
                            |
                            |
              <lenguaje compilador>
```

Obviamente, es necesario poder procesar <lenguaje compilador> para utilizar <compilador>. Por lo que <lenguaje compilador> debe ser, a su vez, <lenguaje fuente> de otro <compilador>. En la gráfica siguiente se establece la relación entre los diferentes <compilador>es hasta la obtención de Alpha-Basic, el cual es traducido a M68000 por AB.

```
Assist -> [A] -> Alpha-Basic
           |
           |
        Lest -> [Lf] -> Alpha-Basic
                  |
                  |
              Lest -> [L] -> Alpha-Basic
                        |
                        |
            +---------+
            |
            |
      Sub-Lest´-> [SL´] -> Sub-Lest -> [SL] -> Alpha-Basic
                   |                    |
                   |                    |
                   |               Alpha-Basic
                   |
                   |
             Sub-Lest -> [SL] -> Alpha-Basic
                           |
                           |
                      Alpha-Basic
```

3.4 ESTRUCTURA DE LEST Y ASSIST

En esta sección se presenta la relación estructural entre los módulos más significativos de Lest y Assist. El diagrama indica la jerarquía en la invocación de módulos y, cuando es posible, el orden de invocación se representa en la alineación de izquierda a derecha de los módulos de un mismo nivel.



En los Apéndices C y D se encuentra el código de Lest y Assist. Allí, si se desea, se puede observar al detalle el funcionamiento de dichos módulos.

3.5 ASPECTOS TECNICOS IMPORTANTES

Se consideró conveniente describir en este informe los criterios y técnicas más importantes que dieron forma a los módulos más significativos del compilador de Assist.

3.5.1 COMPILACION EN DOS PASADAS

Debido a que la definición de los modulos va generalmente después de las llamadas a éstos, es necesario separar el proceso de compilación en dos pasadas. Esto se debe a que parte de la información necesaria para generar el código de las llamadas no se conoce en el momento que aparecen.

Se escogió realizar prácticamente todo el proceso de compilación en la primera pasada y dejar únicamente para la segunda pasada la completación del código faltante. Para lo cual se utiliza la información recopilada en la primera pasada.

3.5.2 RECONOCIMIENTO Y GENERACION DE CODIGO

El problema de reconocer sintácticamente las instrucciones de Assist y generar el código correspondiente se atacó con dos métodos. Se escogió procesar las instrucciones y verbos de Assist directamente desde programación, debido a que poseen una sintaxis relativamente sencilla. Pero para validar y traducir las expresiones se

adaptó una técnica de 'parse' predictiva. En primer lugar porque la gramática de estas expresiones es del tipo LL(1). En segundo lugar, porque validar y traducir las expresiones con programación sería más ineficiente y potencialmente provocaría más errores. La formalización de este método aparece en la sección 3.6.

3.5.3 EVALUACION DE EFICIENCIA DE LAS ESTRUCTURAS DE DATOS

En términos generales, el ADT (Abstract Data Type [1]) necesario para almacenar las entidades manejadas por el compilador es el CONJUNTO con las operaciones: INSERCION y BUSQUEDA. No son necesarias operaciones como REMOCION o SIGUIENTE. Esto hace especialmente adecuado el uso de 'hashing' ya que posee una complejidad esperada de O(n) para n inserciones y/o búsquedas. Sin embargo, se estima que muchas tablas guarden únicamente entre 5 y 15 elementos (por ejemplo: la tabla que contiene las variables locales del módulo en proceso de compilación), haciendo más importante la constante de proporcionalidad que el orden del algoritmo. El problema radica en que cualquier implementación de 'hashing' en Alpha-Basic poseería una constante alta. Principalmente porque el código generado por el compilador de Alpha-Basic ejecuta todas las operaciones aritméticas en punto flotante, inclusive los subíndices de los arreglos donde se almacenarían las tablas de 'hashing'. Otra alternativa sería almacenar la información en un 'string', distinguiendo las entidades por caracteres especiales.

Definitivamente, es la estructura que utiliza en la forma más óptima el espacio, pero debe ser también eficiente en el tiempo de ejecución de INSERCION y BUSQUEDA. Como no se requiere orden alguno, INSERCION se simplifica a únicamente agregar al final del 'string'. Si se posee un puntero indicando el final del 'string', una INSERCION toma O(l(s)) donde l(s) es el largo de s. La búsqueda de un elemento sería O(l(t)) donde t es la tabla. Estos ordenes son considerablemente mayores que los de 'hashing'; pero si pensamos que para ambas operaciones el compilador de Alpha-Basic genera una única instrucción en lenguaje máquina (aunque de O(l(t)) ciclos para BUSQUEDA), notamos una constante de proporcionalidad comparativamente mucho menor. Por lo tanto, para largos de 'inputs' realísticos es más conveniente esta alternativa.

Sin embargo, la estructura anterior no satisface todas las demandas de almacenamiento y recuperación de información, por lo que se implementaron también: 'stacks', arreglos, tablas con punteros a 'strings', etc.

3.6 IMPLEMENTACION DEL ´PARSER´ PREDICTIVO

La técnica seleccionada para validar las expresiones de Assist consiste en un programa que, en base a la gramática almacenada en una tabla, determina si un ´input´ dado pertenece al lenguaje definido por la gramática.

El programa para procesar cualquier gramática del tipo LL(1) es el mismo [3]. Lo único que varía es la tabla, en la cuál, después de algunas transformaciones, se encuentra definida la gramática.

En esta sección se detalla la construcción de dicha tabla a partir de la gramática. El estándar de notación es BNF [13]:

1. Definición gramatical de las expresiones de Assist:

<exp> ::= <exp><bop><exp> ¦ <exp>[<exp><sep><exp>] ¦ <uop><exp>
¦ (<exp>) ¦ <idp><exp-1>) ¦ <id> ¦ <n> ¦ <s>

<exp-1> ::= <exp>,<exp-1> ¦ <exp>

<sep> ::= , ¦ ;

<id> ::= * es un identificador válido en Assist, el ´scanner´ se encarga de validarlo *

<idp> ::= * es un identificador válido en Assist seguido de un paréntesis que abre *

<n> ::= * es un número válido en Assist *

<s> ::= * es un ´string´ válido en Assist *

<bop> ::= * es un operador binario válido de Assist *

<uop> ::= * es un operador unitario válido de Assist *

2. Eliminación de la Recursión a la Izquierda [3]:

```
                 (1)                 (2)               (3)
<exp> ::= <uop><exp><exp´> ¦ (<exp>)<exp´> ¦ <idp><exp-l>)<exp´>

                 (4)          (5)          (6)
          ¦ <id><exp´> ¦ <n><exp´> ¦ <s><exp´>

                 (7)                   (8)                  (9)
<exp´> ::= <bop><exp><exp´> ¦ [<exp><sep><exp>]<exp´> ¦ e

<exp-l> ::= <exp> ¦ <exp>,<exp-l>

          (13) (14)
<sep> ::=   ,  ¦  ;
```

3. Factorización a la izquierda sobre <exp-l> [3]:

```
                 (10)
<exp-l> ::= <exp><exp-l´>

                (11)      (12)
<exp-l´> ::= , <exp-l> ¦  e
```

4. Determinación de los Primeros (P) y Siguientes (S) necesarios:

Las reglas de producción finales están numeradas entre paréntesis. Las funciones P y S hacen referencia á estos números.

Si a es cualquier ´string´ de símbolos gramaticales, P(a) será el conjunto de terminales que inician ´strings´ derivados de a. Si $a \overset{*}{\Longrightarrow} e$, entonces e está también en P(a).

Se define S(N), para un noterminal N, como el conjunto de terminales t que pueden aparecer inmediatamente a la derecha de N en cualquier forma sentencial, que es, $S \overset{*}{\Longrightarrow} xNty$ para cualquier x y y [3].

P(1) = { <uop> }, P(2) = { (}, P(3) = { <idp> },

P(4) = { <id> }, P(5) = 8 <n> }, P(6) = { <s> },

P(7) = { <bop> }, P(8) = { [},

S(<exp´>) = {),<sep>,], , ,$}

P(9) = { e,),<sep>,], , ,$}

P(10) = { <uop>,(,<idp>,<id>,<n>,<s> }

P(11) = { , }

S(<exp-1´>) = {) }

P(12) = { e,) }

4. Construcción de la tabla de ´parsing´:

La tabla T está etiquetada en su columnas por no-terminales y en sus filas por terminales. Para construir la tabla se sigue la siguiente regla: sea A ==> a una producción. Para cada terminal t en P(a), se agrega A ==> a en T[a,A]. Si e está en P(a), se agrega A ==> a en T[b,A] for cada terminal b en S(A).

Tabla de ´parsing´:

T \ NT	<exp>	<exp´>	<exp-l>	<exp-l´>
<uop>	<u><e><e´>		<e><el´>	
(	(<e>)<e´>		<e><el´>	
<idp>	<ip><el>)<e´>		<e><el´>	
<id>	<id><e´>		<e><el´>	
<n>	<n><e´>		<e><el´>	
<s>	<s><e´>		<e><el´>	
<bop>		<bop><e>		
[		[<e><sp><e>]<e´>		
)		e		e
<sep>		e		
]		e		
,		e		,<el>
$		e		

4. CONSIDERACIONES FINALES

El desarrollo de este compilador de Assist es el resultado de 3 años (83-85) de experiencia con el sistema operativo AMOS de Alpha-Micro y 4 años (84-87) con diversos lenguajes de 4a. generación; más la puesta en práctica de muchas técnicas de ciencias de la computación.

Gran parte del código ya está verificado al desarrollar el compilador en él mismo. Pero aunque las características del diseño parecieran excelentes y la implementación se probara óptima, el trabajo no tendría gran valor si simplemente los usuarios no encuentran en él una herramienta útil.

Al momento de escribirse este informe ya se tiene instalado en una AM-1000 un preprocesador de Lest y se están considerando las críticas del usuario.

5. BIBLIOGRAFIA

[1] Aho, A.; J. Hopcroft y J. Ullman. Data structures and
1983 algorithms. Reading (Estados Unidos), Addison-Wesley. 417 pp.

[2] _______, J. Hopcroft y J. Ullman. The design and analysis
1974 of computer algorithms. Reading (Estados Unidos), Addison-Wesley. 469 pp.

[3] Aho, A.; J. Ullman. Principles of compiler design. Reading (Estados Unidos), Addison-Wesley. 604 pp.

[4] AlphaBASIC user´s manual (software manual). Alpha Micro.
1984 Irvine (Estados Unidos). (DWM-00100-01 REV. B05).

[5] AlphaPASCAL user´s guide (software manual). Alpha Micro.
1982 Irvine (Estados Unidos). (DWM-00100-08 REV. B01).

[6] Amos/L Lokser user´s manual (software manual). Alpha Micro.
1983 Irvine (Estados Unidos). (DSS-10034-00 REV. A00).

[7] Caine, H.; E. Gordon. "PDL-a tool for software design".
1985 ACM Sigsoft software engineering notes (Estados Unidos) 6(6): 68-73

[8] Dowdell, W. "What MIS professionals need from their 4GLs"
1986 Computerworld (Estados Unidos). (septiembre).

[9] Ford, H. Software engineering. Reading (Estados Unidos),
1982 Addison-Wesley. 513 pp.

[10] Ford, R.; C. Marlin. "Implementation prototypes in the
1982 development of programming language features". ACM Sigsoft software engineering notes (Estados Unidos) 7(5): 61-66.

[11] Isam system user´s manual (software manual). Alpha Micro.
1984 Irvine (Estados Unidos). (DWM-00100-06 REV. C01).

[12] Mantis programming reference manual. Cincom Systems, Inc.
1984 Cincinnati. (P19-1104-00).

[13] Wirth, N. Pascal report. Zurich, Springer-Verlag. 95 pp.

[14] Transact/3000 reference manual. Hewlett Packard. Cuperti-
1983 no (Estados Unidos). (Part No. 32247-90001).

APENDICE A

CONVENCION DE SIMBOLOS EN NOTACION SINTACTICA

- Caracteres libres

Indican que tales caracteres deben ir tal como aparecen en la definición. Las letras que aparezcan pueden indistintamente ponerse en mayúscula o minúscula.

Ej.: UNDO

- Caracteres entre corchetes angulosos (<>)

Dichos caracteres nombran a la expresión sintáctica que va en el mismo lugar que éstos.

Ej.: IF <condición>

- Expresión sintáctica entre llaves ({})

Indica que dicha expresión puede repetirse 0 veces o más.

```
Ej.:  { CASE <condición>       o      { CASE <condición>   }
                                      {                    }
           <código> }                 {     <código>       }
```

- Expresión sintáctica entre corchetes ([])

Indica que dicha expresión puede incluirse o no

```
Ej.:  [ OTHERWISE          o      [ OTHERWISE       ]
                                  [                 ]
           <código> ]             [    <código>     ]
```

- Expresiones sintácticas, apiladas una sobre otra,

entre corchetes

Indica que debe ir una y sólo una de dichas expresiones.

```
Ej.:  [ EXCLUSIVE ]
      [           ]
      [ SHARED    ]
```

Si una de las expresiones va subrayada, indica que ésta puede estar explícita o implícitamente. Equivalente al valor 'default'.

Ej.: [EXCLUSIVE]
      [   ]
      [<u>SHARED</u>   ]

APENDICE B

SINTAXIS DE LA ESPECIFICACION FUENTE DEL DICCIONARIO

Esta especificación es compilada para: verificar su correctitud y producir un formato más cómodo para el compilador de Assist.

```
FILE <nombre archivo (6 letras)>[.<extensión>]

   TYPE [ SEQUENTIAL ]
        [            ]
        [ RANDOM     ]
        [            ]
        [ INDEXED    ]

   RECORD

     { <definición campo> }

   ENDREC

 [ LENGTH <largo en bytes del registro> ]

 [ KEYS                                                                ]
 [                                                                     ]
 [ { <nombre índice>[<largo llave>] = <expresión>[,DUPS] }            ]
 [                                                                     ]
 [ ENDKEYS                                                             ]

 ENDFILE
```