E_spanish

COLLABORATORS

	TITLE : E_spanish	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 9, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

E_spanish

1.1 main

```
                +-----------------------------------------------+
                |                                               |
                |                  Amiga E v3.1a                |
                |          Compilador para  El Lenguaje E       |
                |            Por Wouter van Oortmerssen         |
                |                                               |
                +-----------------------------------------------+
```

Contenido:

[Esta es la documentación que acompaña al compilador en catellano.
 También está disponible en Aminet una versión preparada para ser
 impresa ('E_spanish.tex.gz'), la cual ha sido revisada con más
 detalle que esta guía y, creo, mejor estructurada. Si tienes
 alguna duda o comentario, no dudes en escribirme:
 Antonio J. Gomez Gonzalez -- u0868551@oboe.etsiig.uniovi.es]

1.2 ch_0a

```
Introducción
------------
```

E es un lenguaje de alto nivel orientado a objetos/procedural/funcional no puro, influenciado en gran medida por lenguajes tales como C++, Ada, Lisp, etc. Es un lenguaje de programación de propósito general, y la implementación para Amiga está dirigida específicamente a programar aplicaciones de sistema. El número de características del lenguaje es demasiado grande como para resumirlas todas, entre las cuales se incluye: velocidad de compilación superior a 20000 líneas por minuto en un Amiga a 7 Mhz, ensamblador en línea y enlazador (linker) integrados dentro del compilador, un gran conjunto de funciones integradas, buen concepto de módulos con includes de v39 como módulos, sistema de tipos flexible, expresiones 'entrecomilladas', listas inmediatas, con tipo, polimorfismo a bajo nivel y de objetos, manejo de excepciones, herencia, ocultamiento de datos, métodos, valores de retorno múltiples, argumentos por omisión, reserva de registros, manejo rápido de memoria, unificación, Celdas-LISP, (macro-)preprocesador, depurador a nivel de fuente, y mucho más ...

Pulsa aquí para ver el primer programa en E

Distribución
Restricciones de la demo, Registro y Lugares
Uso del compilador
Cambios de v2.1b a v3.0a, y de v3.0a a v3.1a
Información adicional

1.3 hola_mun

Así es como se escribe el 'HolaMundo' en E:

```
/* nominado para ejemplo 'más aburrido' */

PROC main()
  WriteF('¡Hola Mundo!\n')
ENDPROC
```

1.4 ch_0b

```
Distribución
------------
```

La distribución incluye:

```
bin/      compilador EC y las utilidades de apoyo.
modules/  módulos E de Amiga v39 y utilidades como módulos enlazables.
docs/     documentación de E.
src/      fuentes de ejemplo en E.
tools/    herramientas opcionales para usar con E.
```

La distribución de Amiga E v3.0 que incluye la versión de demostración del compilador es FreeWare y se puede copiar libremente. Algunos de vosotros habreis recibido un archivo con la versión registrada del compilador, el cual NO es FreeWare, y NADIE más debe copiarlo.

No está autorizada la distribución de E por un importe superior al de disco+correo (menos de 500 pts), así como tampoco se autoriza ningún otro tipo de distribución cuyo único propósito sea el de obtener beneficios.
Prohibo a Serge Hamouche o su compañia "France Festival Distribution" distribuir E de forma alguna.
Por otro lado, hay una traducción de este documento en francés con mi autorización (realizada por Olivier Anh) que está disponible en Aminet.

Esta distribución debe ofrecerse siempre como un todo, es decir, de la misma forma que el archivo '.lha' original. Sin añadidos, modificaciones, traducciones, distribuciones parciales o cualquier otra cosa que se pueda hacer sin mi permiso.

No hay garantías. Si consigues ahogar al pez que tienes en la pecera usando E, o si E resulta no ser apropiado para pedir pizzas, ese es tú problema (y sólo tuyo). Pase lo que pase no me maldigas por ello.

Fred Fish tiene premiso especial para distibuir E en sus CD-ROMs.

1.5 prohibo

```
Prohibo a Serge Hamouche.
-------------------------
```

Serge Hamouche ha distribuido E sin mi permiso, utilizando mi nombre de forma incorrecta y robándome dinero a expensas de los programadores de E franceses. Si le has pagado dinero, reclámaselo y coménta esta situación a toda la gente que puedas para que no lo sigua haciendo.

1.6 ch_0c

```
Restriciones de la demo, Registro y Lugares
-------------------------------------------
```

La distribución de Amiga E es PD, y contiene la versión demo del compilador. Los programadores registrados obtienen el compilador completo en un archivo a parte, en el que sólo está "EC". EC Registrado

Así que, ¿que hay de la versión demo que se incluye?
El compilador sólo creará ejecutables de menos de 8KB, por encima de esos 8KB mostrará el error "workspace full" ('espacio de trabajo lleno') A parte de eso, es totalmente funcional.

Puedes evaluar la demo durante _dos semanas_, tras las cuales debes decidir si registrarte o dejar de usarla, incluso si sólo escribes programas de menos de 8KB. [Solían ser 12KB, sí, ¡adivina porqué lo cambié!]

Coste de una copia.
Forma de enviar el dinero.
Donde enviar el dinero.
Actualizaciones.

Asegúrate de enviar información tan completa como puedas con el dinero, por ejemplo, dirección completa, dirección de correo electrónico, incluso puedes añadir la configuración de tu Amiga. Yo no voy a escribir aquí uno de esos tontos impresos de registro :-)

1.7 dondeenv

Donde enviar el dinero.

Puedes enviarmelo directamente a mí.

O también puedes registrarte en uno de los lugares de registro de E autorizados:

Australia
Inglaterra
E.E.U.U.
Alemania
(Italia pronto)

Para más información contacta con ellos personalmente. Los residentes en pasíses cercanos también pueden elegir registrarse en esos sitios. Señalar que para esos lugares sirven las mismas reglas: sea cual sea el método de pago elegido, asegúrate de que reciban la cantidad correcta.

1.8 actual

Actualizaciones.

Las actualizaciones de la distribución se pueden encontrar en DP como siempre, y las actualizaciones del EC registrado se distribuyen como parches, también en Domínio Público.

1.9 ecregistrado

EC Registrado.

Si ya has recibido el EC registrado, sería conveniente que lo pusieras en el directorio bin/. Todo el mundo puede copiar libremente el archivo de distribución, pero, por favor, asegúrate de que no distribuyes tu copia registrada de EC a nadie, ni siquiera a los amigos. No he serializado esas copias por la confianza que tengo en mis usuarios registrados, por ello, no rompas esa confianza. La cuota de registro es

bastante asequible, y la versión v2.1b sigue estando disponible, por lo que no hay excusa para que piratees E.

1.10 envio

Formas de enviarme el dinero.

Esto no es sencillo ya que muchos bancos aplican un buen recargo a las transferencias de moneda extranjera. Algunas opciones son :

- Enviar dinero en efectivo (preferible).
  [40$]

- Enviar un "Giro postal internacional". Esto ha funcionado sin problemas desde Francia e Italia (y España).
  [40$]

- En Europa, envía un EuroCheque (en Florines). Creo que no hay recargos por esto, aunque compruebalo o pregunta en tu lugar. [¡Asegúrate de señalar "DFL" como moneda!]
  [40$]

- Por transferencia directa a la cuenta de mi banco.
  [44$ a 50$, holanda: 40$]

- **NO** envies CHEQUES, MONEY ORDERs o métodos de pago similares.
  [50$ to 55$]

Puedes optar por cualquier (otro) método, aunque DEBES asegurarte de que voy a poder obtener todo el dinero con _facilidad_. Si no tienes en cuenta los gastos de transferencia y la cantidad final es menos de 40$, deberás realizar otra transferencia para obtener tu versión registrada.

1.11 nocheq

NO envies CHEQUES, MONEY ORDERs o métodos de pago similares.

Aquí piden 15$ en concepto de gastos de transferencia para hacer efectivos cheques de banco (excepto EuroCheques, !si te los hacen efectivo!).

1.12 mibanco

Transferencia directa a mi banco.

Puedes hacer la transferencia a la cuenta de mi banco:

427875951, ABN-AMRO bank, the netherlands

o a mi cuenta de giro-postal:

6030387, PostBank, the netherlands

Asegúrate de cubrir todos los gastos de transferencia que pueda cargar tu banco y/o me puedan cargar a mí. Por ejemplo, las transferencias por la 'Caja Postal' desde fuera de Holanda cuestan 4$, las transferencias bancarias hasta 10$.

1.13 dinefectivo

Enviar dinero en efectivo.

Puede resultarte extraño, pero es la forma más efectiva de intercambiar dinero. Cambiar tu dinero a papel-moneda (billetes) holandeses siempre es una buena idea. Si no estas seguro enviando dinero por correo, la mayoría de las oficinas de correos te ofrecen un servicio con el que te comunican que me ha llegado todo de forma correcta.

1.14 coste

Pedido de una copia.

El precio básico es 40$.
- Resta 5 $ si estas dispuesto a recibir tu EC registrado por correo electrónico uuencodeado, en vez de en disco. Es la forma más rápida de conseguir la v3.
- Añade 10$ por cada actualización importante (si llega a salir). Aunque esto es sólo para gente que viva en Siberia o en sitios así, ya que las actualizaciones estan disponibles en DP.
- Contacta conmigo para un número de pedidos grande.

También permito el pago en otros tipos de divisas para simplificar el pago a alguna gente. ¡Aunque nunca en monedas raras!

Ejemplos (US$):	40
Florines Holandeses (preferible)	65
Marco Alemán	60
Etc...	

NOTA: estos valores se han calculado con cambio a fecha de 30 de Junio de 1994, si tu divisa oscila deberás ajustarlos. Es una buena idea asegurarse de estar un dolar por encima del valor en florines.

1.15 aust_reg

AUSTRALIA

nombre: Rob Nottage
dirección: 10 Chilver Street, Kewdale, Western Australia, 6105
Fax: (09) 458 0154
FidoNet: 3:690/662.0
AmigaNet: 41:616/662.0
Internet: robbage@cougar.multiline.com.au
BBS: (09) 351 8401 v.32, v.42bis.
precio: AUS$ 55 (correo o fidonet crashmail), AUS$ 50 (internet o fidonet email), dependiendo del cambio actual.
pago:
- Postal or money order (preferido)
- Cheque personal o bancario (pagas los costes)
- Transferen. bancaria directa (National Australia Bank, rellena un depósito por la cantidad correcta, nr. de cuenta 086131 435770223, 'Robert K. Nottage')
- Enviando dinero (a tu riesgo)

1.16 ingl_reg

INGLATERRA

nombre: Jason R. Hulance (Dept. E)
dirección: Formal Systems (Europe) Ltd, 3 Alfred Street, Oxford OX1 4EH, INGLATERRA
telephone: (0869) 324350, fuerea de horas de oficina.
email: m88jrh@ecs.ox.ac.uk
price: 26 UK libras (en disco) o 23 UK libras (via E-mail) "Beginner's Guide to Amiga E" (182 pags.) está disponible en formato AmigaGuide, TeX, PostScript e impreso con un gran índice. Esto cuesta 5 libras esterlinas (formas no impresas) o 10 libras esterlinas (forma impresa).
pago: Por razones de seguridad realiza el pago con un cheque (a nombre de "Jason Hulance"), aunque también se aceptan giros postales/efectivo/transferencias bancarias.

1.17 eeuu_reg

E.E.U.U.

nombre: Barry Wills
dirección: 5528D Pryor Dr., SAFB, IL 62225
email: el269@cleveland.freenet.edu
precio: 40$ (US correo), 35$ (email internet)
pago:
- giro o cheque-dinero: método preferido.
- efectivo -- Vale, pero EL RIESGO ES TUYO.
- cheque personal: Vale, pero cuenta con una espera de hasta 3 semanas por tu compilador registrado.

1.18 alem_reg

```
ALEMANIA

nombre:         Jörg Wach
dirección:      Waitzstr. 75, 24105 Kiel, Deutschland/Alemania
email:          JCL_POWER@freeway.sh.sub.de
precio:         60dm por correo, 54dm por email.
```

1.19 ch_0d

```
Uso del compilador.
-------------------

   Para  instalar  Amiga E en tu sistema,  sólo tienes que copiar toda la
distribución en algún lugar de tu sistema, extender tu path al directorio
BIN,  y asignar 'EModules:' al directorio MODULES.

   Sintaxis del compilador (2.04+):
      SOURCE/A,REG=NUMREGALLOC/N/K,LARGE/S,SYM=SYMBOLHUNK/S,NOWARN/S,
      QUIET/S,ASM/S,ERRLINE/S,ERRBYTE/S,SHOWBUF/S,ADDBUF/N/K,
      IGNORECACHE/S,HOLD/S,WB/S,LINEDEBUG/S,OPTI/S,DEBUG/S:
   Sintaxis para 1.2 a 2.03:
      EC [-opcs] <fichero_fuente>

   Pulsa aquí para ver un ejemplo de compilación.

   El  compilador no se puede hacer residente.  Si un programa usa fiche-
ros módulo para definiciones de biblioteca como:
         MODULE 'GadTools', 'Reqtools'
el compilador necesita saber donde están.  Hay dos posibles soluciones:
1. Realizar la asignación "EModules:" al directorio de módulos (mejor).
2. Indicar en el código fuente donde debe buscar los módulos, como:
         OPT DIR='dh0:src/e/modules'

   Puedes  ver  las  opciones  estandard de AmigaDos en cualquier momento
escribiendo  'EC  ?'.  Las opciones en las versiones 1.2-2.03 del sistema
son  de  tipo Unix (hay que escribirlas todas juntas, precedidas de "-").
```

1.20 opciones

```
2.04+           1.2+ Descripción
--------------- ---- -----------
LARGE  ........ -l   Compila con modelo código/datos grande. ( OPT LARGE )
ASM  .......... -a   Pasa EC a modo ensamblador.               ( OPT ASM )
NOWARN  ....... -n   Suprime advertencias.                   ( OPT NOWARN )
SYM  .......... -s   Inserta un symbolhunk al ejecutable,  que utilizarán
                     los profilers, depuradores, desensambladores etc.
LINEDEBUG  .... -L   Inserta bloque de depuración  "LINE"  al ejecutable,
                     para profilers, depuradores,...     ( OPT LINEDEBUG )
REG=N  ........ -rN  Hace que se reserven  N  registros por PROC.  (0 por
                     omisión, ¡mira en algún otro sitio sobre esto!)
```

```
QUIET  ........ -q   Pone EC en modo silencioso,  sólo escribe cuando hay
                     errores o advertencias.
WB  ........... -w   Pone el WB delante (para scripts).
SHOWBUF  ...... -b   Muestra información de uso de memoria de buffer.
ADDBUF=X  ..... -mX  Hace que EC reserve más memoria para buffers. X está
                     en el intervalo  1..9  --número de mínimo de bloques
                     de 100KB reservados. (1 por omisión, casi no se usa)
ERRLINE  ...... -e   EC retorna el número de línea del error.
ERRBYTE  ...... -E   EC retorna la posición (en bytes) respecto al inicio
                     del fichero en la que se produjo el error.
IGNORECACHE  .. -c   Deshabilita el uso del caché de módulos.
HOLD  ......... -h   Espera que se pulse <return> antes de continuar.
OPTI  .............. Activa las optimizaciones (de momento es REG=5).
DEBUG  ............. Inserta info. de depuración en ejecutable o módulo.

   Ejemplo:  ec LARGE blabla  -> compila blabla.e con modelo grande.
   En la mayoría de los casos no necesitarás usar estas opciones.
```

1.21 ejemcomp

```
Ejemplo de compilación.
-----------------------

   Vamos  a  compilar el programa 'HolaMundo.e'.  El compilador producirá
un ejecutable 'HolaMundo'.

      E:>  ec holamundo
      Amiga~E~Compiler/Assembler/Linker~v2.4f~(c)~91/92/93~$#%!
      lexical analysing ...
      parsing and compiling ...
      no errors
      E:>  holamundo
      ¡Hola Mundo!
      E:>  list
      HolaMundo.e                  89 ----rwed Hoy       17:37:00
      HolaMundo                   656 ----rwed Hoy       17:37:00
      2 files - 4 blocks used
      E:>
```

1.22 ch_0e

```
Cambios de v2.1b a v3.0a, y de v3.0a a v3.1a
--------------------------------------------

   Lo que hay nuevo en v3.0a :

    CARACTERISTICAS NUEVAS IMPORTANTES
    CARACTERISTICAS NUEVAS MENORES
    ERRORES/PROBLEMAS CORREGIDOS
    OTROS CAMBIOS A TENER EN CUENTA

   Cambios en v3.0b :
```

```
    ERRORES CORREGIDOS

   Cambios en v3.0e :

    ERRORES CORREGIDOS

   Cambios en v3.1a :

    CARACTERÍSTICAS NUEVAS
    ERRORES CORREGIDOS

   Truco para usuarios de versiones anteriores:  haz un 'diff' de las dos
versiones de 'reference.doc'.
```

1.23 nuevasimp

```
Caracteriticas nuevas importantes en v3.0a
-------------------------------------------

- Compilación a módulos.
- Orientado a Objetos: herencia de objetos, privado/público, métodos ...
- Argumentos por omisión en PROCs, y algunos predefinidos.
- Valores de retrono múltiples.
- Variables registro/reserva de registros ...
- Integración de reales.
- Opereradores NEW y END,manejo super rápido de memoria.
- Potente variación sintáctica de SELECT.
- Unificación.
- Posibilidad de tipos más complejos en OBJECTos, PTR TO <type>, etc.
  Referencia a elementos más potente (x.y[a].z ...).
- Recogida de residuos con Celdas-Lisp.
- Valores de función (punteros a PROCs).
- Caché de módulos (ShowCache).
- Reconstrucción de errores: obtiene error en punto exacto de la línea.
- Gran cantidad de funciones predefinidas nuevas (E/S,...)
- Gran cantidad de módulos nuevos (que serán más en seguida).
- Varias optimizaciones para hacer  EC  significativamente más rápido.
- Varias optimizaciones del código.
- Enlazador/sistema de módulos inteligente.
- El compilador usa mucha menos memoria.
- Bloques (hunks) de símbolos y depuración.
- Varias utilidades, como  EE (gran editor de Barry Wills), Build (make),
  o2m (convierte ficheros .o de ensamblador en módulos), E-Yacc, ...
- Docs nuevos, EXCELENTE tutorial de Jason Hulance ('The Guide').
```

1.24 nuevasmen

```
Caracteristicas nuevas menores en v3.0a
---------------------------------------

- Mutación de punteros, operador '::'.
```

- EXCEPT DO: continúa en el manejador.
- EXIT <expbool> en WHILE/FOR.
- Comentarios de una línea.
- Para algunos: módulos de OS v39.
- Caracteres const pueden contener ahora "\n\t\e\a" "\0\\b\q". (\q = ")
- Advertencia para asignaciones "sueltas" ahora con número de línea.
- La referencia a PTR TO <objeto> con sólo [] resulta ahora en un puntero (por ejemplo x[1] = siguiente objeto)
- Varias pequeñas optimizaciones del código.
- Muchas funciones de cadenas/listas/E/S ahora retornan valores útiles.
- La línea de comandos de EC ahora tiene el estilo de ReadArgs() para usuarios de 2.04+.
- Mejor finalización de líneas.
- Varias funciones predefinidas nuevas. (E/S,...)
- Expresiones con *2 *4 /2 y /4 optimizadas a sumas y desplazamientos.

1.25 errcorreg

Errores/problemas corregidos en v3.0a

- TAMBIEN RECONOCE '.e' en la línea de comandos.
- Varios errores corregidos (y, obviamente, se introducen nuevos).
- Ahora también se dispone de una variable 'stdin'
- La referencia a un PTR TO INT con [] resultaba en un entero sin signo en lugar de uno con signo (NOTA: ¡esto puede cambiar!)
- EC podía romper cuando faltaban ENDIFs: corregido.
- CHAR se alineaba de forma incorrecta.
- RAISE acepta constantes negativas.
- Div() no aceptaba números negativos.
- errores ensamblador en línea: ADD.L ...,Ax era ADDX, varios pequeños problemas tontos.
- Faltaba documentación de StringF, la de InStr() incorrecta.
- SIZEOF ahora también reconoce CHAR/INT/LONG.
- WaitIMessage() hacía algo tonto.
- EC no podía consumir recursos muy grandes (>500k).
- Un WHILE o un UNTIL con una expresión IF como booleano podía generar código erróneo.
- [exp]:CHAR generaba código erróneo.
- Algunos errones sintácticos podían pasar desapercibidos.
- La consola se abría en modo READWRITE.

1.26 cambiosen

Otros cambios a tener en cuenta en v3.0a

- Opción "-s" de v2.1b eliminada.
- 'OPT' debe ser lo primero del código fuente; otras construcciones tienen una posición más estricta en el código fuente.
- EC puede generar 'internal errors' (errores internos) en alguno casos. Si eso ocurre, comunícamelo, con el código fuente que lo propició.

```
- La palabra clave "IS"  equivale a "RETURN"  directamente después de un
  "PROC"
- Ha cambiado el convenio de los registros del ensamblador en línea.
```

1.27 errcorr0b

```
Errores corregidos en v3.0b
---------------------------
(casi todos eran pequeños errores, ¡con grandes consecuencias!)

- Errores internos en sistemas 68000.
- Problemas con herencia de métodos.
- Los módulos podían tener 4 bytes de código superfluos en ellos.
- Se podían perder los argumentos por omisón en los módulos.
- La redefinición de métodos podía provocar errores de redeclaración.
- Podían perderse los punteros a <objeto> en objetos de módulos.
  [ahora EC intentará encontrar el objeto correcto, sino tipo = LONG]
```

1.28 errcorr0e

```
Errores corregidos en v3.0e
---------------------------

- Error importante: el enlazador no enlazaba módulos >32k.
- Hit de Enforcer en el enlazador.
- String() podía cambiar un registro.
- Los errores de un OBJECTo se comunicaban en la primer línea.
- Código generado para AND.L Dn, varreg e instrucciones similares erróneo.
- Tamaño  .B  en EAs (direcciones efectivas) de An no detectado en varias
  instrucciones.
- Podían definirse métodos fuera del alcance del objeto/módulo.
- Las declaraciones de objetos podían empezar como privadas sin motivo.
```

1.29 nuevas31a

```
Características nuevas en v3.1a
-------------------------------

- ¡¡¡Depuración a nivel fuente!!! opción DEBUG y depurador externo EDBG.
- ¡¡¡Preprocesador incorporado!!!  ( compilación  condicional  y  macro
  preprocesamineto), compatible con Mac2E y Cpp.
- Palabra clave SUPER para llamar a métodos de superclase.
- Algunos  fuentes  de ejemplo  nuevos,  nuevos  módulos  de  herramienta
  (FilledVector, EasyGUI, etc. ).
- Nuevas versiones de Aprof y EE.
- Documentación actualizada: nuevas características, FAQ, etc...
```

1.30 errcorr1a

```
Errores corregidos en v3.1a
---------------------------

- obj.miembrodesconocido podía producir errores extraños.
- ShowModule podía mostrar argumentos erróneos.
- Listas [] no aceptaban números reales negativos.
- Pequeños problemas en definiciones de módulos de E corregidos.
- E-Build ejcutaba las aciones en orden inverso :-)
- FlushCache no permitía el vaciado selectivo.
- Podía fallar la Unificación en listas-LISP anidadas complejas.
- Faltaba rutina de liberación de listas reservadas con NEW.
- La falta de corchetes finales podía causar mensajes de error confusos.
- Algunas construcciones de llamadas a funciones anidadas y NEW con
  constructores podía resultar en código erróneo.
- Tanto RealF() como RealVal() tenían errores que podían dar resultados
  erróneos en máquinas sin coprocesador.
- En algunas situaciones los nombres de los métodos podían chocar con los
  miembros de otros módulos.
```

1.31 ch_0f

```
Información adicional.
----------------------
   El compilador de Amiga E ha sido desarrollado a lo largo de más de
tres años y medio fruto de las ideas del autor sobre un buen lenguaje de
programación, y un compilador de calidad específico del Amiga. Ha sido
programado (como podrás haber apreciado) 100% en ensamblador, usando el
ensamblador AsmOne v1.02. Todos los demás programas de apoyo han sido
programados en E mismo.

   Agradezco especialmente a la siguiente gente:

Barry Wills   - por ser el mejor mejor betatester de siempre.
Jason Hulance - por su trabajo en 'The Beginners Guide', y betatest.
Rob Verver    - por comentarios/inspiración.

   También me gustaría agredecer a los siguientes por varias razones (sin
un orden particular) :

Raymond Hoving, Erwin van Breemen, Michael Zuchhi, James Cooper, Jens
Gelhar, Paolo Silvera, Sergio Ruocco, Jeroen Vermeulen, Jan van den
Baard, Joerg Wach, Norman Kraft, Urban Mueller, Charles McCreary, Olivier
Anh, Lionel Vintenat, Rob Nottage, and many more...

   Este compilador se ha programado poniendo mucho cuidado en lo refe-
rente a fiabilidad, y más cuidado aún en el código que genera, además ha
sido comprobado y depurado durante largo tiempo. Sin embargo, es muy
posible que contenga algún error. Si llegas a encontrar uno, o tienes
otros comentarios/preguntas, escríbeme a la dirección de más abajo:
Prefiero _con diferencia_ correo electrónico sobre correo convencional.

APUNTA BIEN: debido a la gran popularidad de las versiones anteriores de
```

Amiga E, recibo una cantidad casi incontestable de correo electrónico, buena parte del cual (>90%) son preguntas que no serían necesarias si la gente leyera los documentos con cuidado. Lo que quiero decir es que me gusta recibir correo electrónico, y no me importa responder preguntas y ayudar a la gente con problemas de programación, pero, antes de escribirme, asegúrate de comprobar los documentos que tienes a tu disposición (como los documentos de Amiga E o los RKRMs) para ver si la pregunta es relevante. Particularmente no debeis enviarme preguntas que no son específicas de E, sino del Amiga. E intenta ser amable conmigo con comentarios como "Yo creo que E debe tener la característica X..." [N.T.: por supuesto en inglés ;-)]. Hecha un vistazo a la FAQ.

Serán bien recibidos registros y donaciones en la siguiente dirección:

Wouter van Oortmerssen
Levendaal 87
2311 JG Leiden
HOLANDA

o, mejor, si tienes acceso a correo electrónico:

wouter@mars.let.uva.nl
wouter@alf.let.uva.nl (si mars rebota)

1.32 formato

Formato.

Tabs, lf, ...
Comentarios
Identificadores y tipos

1.33 ch_1a

Tabs, lf, ...

El código fuente en E son ficheros en formato ASCII puro, con cambio de línea <lf> y punto y coma ";" como separadores de dos sentencias. Las sentencias que tengan un número particularmente grande de elementos se pueden distribuir a lo largo de varias líneas finalizándolas con una coma (o con cualquier otro elemento léxico que normalmente no puede aparecer en el fin de línea), ignorando de esa forma el siguiente <lf>.

En un fichero de código fuente los elementos léxicos pueden estar separados entre sí por cualquier número de espacios, tabuladores, etc.

1.34 ch_1b

Comentarios.

En un fichero fuente los comentarios se pueden poner en cualquier lugar en el que normalmente un espacio sería correcto. Empiezan con '/*' y finalizan con '*/', y se pueden anidar infinitamente. Lo mismo se puede decir para los comentarios de una línea, los cuales empiezan con un '->' y finalizan en el primer <lf>.

```
/* esto es un comentario */
-> esto también.
```

1.35 ch_1c

Identificadores y tipos.

Los identificadores son cadenas que el programador usa para denotar un cierto objeto, en la mayoría de los casos variables, o incluso palabras claves o nombres de funciones predefinidas por el compilador. Un identificador está formado por:

- caracteres en mayúscula o minúscula.
- "0" .. "9" (excepto como primer caracter)
- "_" (el subrayado)

Todos los caracteres son significativos, aunque el compilador sólo mira los dos primeros para determinar el tipo de identificador con el que está tratando:

```
Ambos en mayúscula:           - palabra clave como IF, PROC, ...
                                constante, como MAX_LENGTH, ...
                                mnemotécinico ensamblador, como MOVE, ...
Primera minúscula:            - identif. de variable/etiqueta/objeto, ...
Primera mayús. y seg. minús.  - función del sistema E como: WriteF(), ...
                                llamada de librería: OpenWindow(), ...
```

Señalar que todos los identificadores verifican esta sintaxis, por ejemplo: WBenchToFront() se convierte en WbenchToFront().

1.36 ch_2

Valores inmediatos.

Todos los valores inmediatos en E se evalúan a un resultado de 32bit, la única diferencia entre todos ellos puede ser su representación interna, o el hecho de que retornen un puntero en vez de un valor.

```
Decimal                    (1)
Hexadecimal               ($1)
```

```
     Binario                      (%1)
     Real                        (1.0)
     Caracter                     (a)
     Cadena                    ('bla')
     Lista y lista con tipo  ([1,2,3])
     Celda-Lisp                (<a|b>)
```

1.37 ch_2a

```
Decimal (1).
------------
```

Un valor decimal es una sequencia de caracteres "0".."9", posiblemente precedidos de un signo menos "-" (denotando un valor negativo).

```
Ejemplos: 1, 100, -12, 1024
```

1.38 ch_2b

```
Hexadecimal ($1).
-----------------
```

Un valor hexadecimal usa además los caracteres "A".."F" (o "a".."f") y van precedidos por el caracter "$".

```
Ejemplos: $FC, $DFF180, -$ABCD
```

1.39 ch_2c

```
Binario (%1).
-------------
```

Los números binarios empiezan con un caracter "%" y sólo usan "1" y "0" para formar un valor.

```
Ejemplos: %111, %1010100001, -%10101
```

1.40 ch_2d

```
Real (1.0).
-----------
```

Los números reales difieren de los decimales en que tienen un "." para separar sus dos partes. Se puede prescindir de cualquiera de las partes, pero nunca las dos. Señalar que la representación interna de los números reales es diferente (32 bits IEEE). (pulsa para más información)

```
Ejemplos: 3.14159, .1 (=0.1), 1. (=1.0)
```

1.41 ch_2e

```
Caracter.
---------
```

El valor de un caracter (entre comillas "") es su valor ASCII, es decir "A" = 65. En E los valores caracter inmediatos formaN una pequeña cadena de hasta cuatro caracteres, por ejemplo "FORM", en la que el primer caracter ("F") es el MSB (Byte Más Significativo) de la representación de 32 bits, y "M" es el LSB (Byte menos Significativo).

1.42 ch_2f

```
Cadena ('bla').
---------------
```

Una cadena es la represtación ASCCI de una secuencia de caracteres que se escribe entre comillas simples ''. El valor de tales tipos de cadena es un puntero al primer caracter de la misma. Más específicamente: 'bla' representa un puntero de 32 bits a un área de memoria en la que encontraremos los bytes "b", "l" y "a". En E todas las cadenas terminan con un byte cero.

Las cadenas pueden contener signos de formato introducidos con una barra inversa "\", ya sea para introducir en la cadena caracteres que no son, por alguna razón, visualizables, o para usarla con funciones de formateo de cadenas como WriteF(), TextF() y StringF(), o la función Vprintf() de KickStart 2.

```
\n       cambio de línea (ASCII 10)
\a o ''  apóstrofe: ' (el usado para encerrar la cadena)
\q       dobles comillas: "
\e       escape (ASCII 27)
\t       tabulador (ASCII 9)
\       la propia barra inversa
\0       byte cero. De escaso uso (todas las cadenas acaban en 0)
\b       retorno de carro (ASCII 13)
```

Además, cuando se usan con funciones de formateo:

```
\d       escribe un número decimal
\h       escribe un número hexadecimal
\s       escribe una cadena
\c       escribe un caracter
\z       fija el byte de relleno al caracter '0'
\l       ajusta a el campo a la izquierda
\r       ajusta a el campo a la derecha (ambos actúan como interruptores)
```

A los códigos \d, \h y \s les pueden seguir especificadores de campo:

```
[x]      indica el ancho exacto del campo, x
(x,y)    indica el ancho mínimo (x) y máximo (y), sólo con cadenas

Ejemplo: escribe un número hexadecimal con 8 posiciones y ceros delante:
         WriteF('\z\h[8]\n',num)
```

Una cadena se puede extender sobre varias líneas continuándolas con un signo "+" y un <lf>:

```
'esta es una cadena deliberadamente larga que ' +
'está dividida en dos lineas separadas'
```

1.43 ch_2g

```
Lista ([1,2,3]) y lista con tipos.
----------------------------------
```

Una lista inmediata es la forma constante del tipo de datos LISTa, de la misma forma que una 'cadena' es la forma constante de los tipos de datos STRING (cadena) o ARRAY OF CHAR (arreglo de caracteres).

```
Ejemplo:    [3,2,1,4]
```

Es una expresión que tiene como valor un puntero a una lista ya inicializada. La representación en memoria de una lista es compatible con un ARRAY OF LONG, con información extra sobre la longitud en un offset negativo. Puedes usar estas listas inmediatas en cualquier lugar en el que una función espere un PTR a un arreglo de valores de 32 bits, o una lista.

Ejemplos:

```
['string',1.0,2.1]
[WA_FLAGS,1,WA_IDCMP,$200,WA_WIDTH,120,WA_HEIGHT,150,TAG_DONE]
```

1.44 ch_2h

```
Celdas-lisp (<a|b>).
--------------------
```

Mira en el apartado de Celdas-Lisp y funciones de celdas.

1.45 ch_3a

```
Formato.
--------
```

Una expresión es un fragmento de código formado por operadores, funciones y paréntesis que conforma un valor.
La mayoría están formadas por:

- valores inmediatos.
- operadores (Matemáticos, ...)
- llamadas a funciones.
- paréntesis () [precedencia y agrupamiento].

```
- variables o expresiones variables.

Ejemplos de expresiones:
         1
         'hola'
         $ABCD+(2*6)+Abs(a)
         (a<1) OR (b>=100)
```

1.46 ch_3b

```
Precedencia y agrupamiento.
---------------------------
```

El lenguaje E no tiene ningún tipo de precedencia. Esto quiere decir que las expresiones se evalúan de izquierda a derecha. Puedes cambiar la precedencia encerrando entre paréntesis algunas (sub-)expresiones.

```
Ejemplos:   1+2*3 /* =9 */     1+(2*3) /* =7 */       2*3+1  /* =7 */
```

1.47 ch_3c

```
Tipo de expresiones.
--------------------
```

Hay tres tipos de expresiones que se pueden usar con diferentes propósitos:

- <var>, formada simplemente por una variable.
- <expvar>, formada por una variable, posiblemente con operadores unarios como "++" (incremento), "." (selección de miembro) o "[]" (operador de arreglo). Denota una expresión modificable (como los valores-i (l-values) del C). Señalar que esos operadores (unarios) no forman parte de ninguna precedencia.
- <exp>. Incluye <var> y <expvar>, y cualquier otra expresión.

1.48 ch_3d

```
Llamadas a funciones.
---------------------
```

La llamada a una función es una suspensión temporal del código actual para 'saltar' a esa función, ésta puede ser una función escrita por uno mismo (PROC), o una función proporcionada por el sistema. El formato de una llamada a función es, el nombre de la función seguido de dos paréntesis que encierran argumentos que pueden ser desde 0 hasta un número ilimitado de ellos, que van separados por comas ",". Señalar que los argumentos de funciones son expresiones de nuevo.
(mira Funciones predefinidas y funciones de biblioteca y módulos).

```
Ejemplos:   foo( 1, 2 )
```

```
            Gadget( buffer, glist, 2, 0, 40, 80+offset, 100, 'Cancela' )
            Close( handle )
```

1.49 ch_4

```
Operadores.
-----------

         Matemáticos              (+ - * /)
         Comparación      (= <> > < >= <=)
         Lógicos y de bits         (AND OR)
         Unarios   (SIZEOF ` ^ {} ++ -- -)
         Tripletes          (IF THEN ELSE)
         Estructuras                  (.)
         Arreglos                   ([])
         Operador real               (!)
         Expresiones de asignación   (:=)
         Secuenciamiento            (BUT)
         Reserva dinámica de memoria (NEW)
         Unificación                (<=>)
         Mutación de punteros        (::)
```

1.50 ch_4a

```
Matemáticos (+ - * /).
----------------------
```

Estos operadores infijos combinan una expresión con otro valor para obtener un nuevo valor. "-" se puede usar como primera parte de una expresión, lo cual implica un 0 (es decir -exp => 0 - exp). Tambien hay que señalar que por omisión "*" y "/" son operadores de 16 bits.

```
Ejemplos:      1+2, MAX-1*5, -a , -b+1
```

```
(mira Mul() y Div() sobre aritmética de 32 bits)
(mira también la sobrecarga de operadores para uso con reales).
```

1.51 ch_4b

```
Comparación (= <> > < >= <=).
-----------------------------
```

Igual que los operadores matemáticos, con la diferencia de que los resultados son, o TRUE (cierto, valor de 32bit -1), o FALSE (falso). También se pueden sobrecargar para operar con reales.

1.52 ch_4c

```
Lógica y con bits (AND OR).
---------------------------
```

Estos operadores pueden combinar valores de verdad para obtener unos nuevos, o producir operaciones AND y OR con bits.

```
Ejemplos:      (a>1) AND ((b=2) OR (c>=3))         /* lógica   */
               a:=b AND $FF                        /* con bits */
```

1.53 ch_4d

```
Unarios (SIZEOF ^ {} ++ -- `).
------------------------------
```

- SIZEOF <identobjeto>
  Simplemente retorna el tamaño de cierto objeto o CHAR/INT/LONG.
  Ejemplo: SIZEOF newscreen + SIZEOF INT
- {<var>}
  Retorna la dirección de una variable o etiqueta. Este es el operador que usarías para pasar una variable como argumento de una fucnión por referencia, en lugar de por valor, que es la forma por omisión en E.
  Ejemplo: Val(input,{x})
- ^<var>
  El compañero de {}, escribe o lee variables que se han pasado por referencia. También se puede usar para "leer" o "escribir" de forma directa valores LONG de memoria, si <var> es un puntero a tal valor.
  Ejemplos: ^a:=1 b:=^a

```
            PROC set(var,exp)   -> Una función propia de asignación...
               ^var:=exp        -> Para ser llamada con set({a},1),
            ENDPROC             -> de forma que equivale a :  `a:=1'
```

- <varexp>++ y <varexp>--
  Incrementa (++) o decrementa (--) el puntero que se denota con <varexp> en el tamaño de los datos a los que apunta. Esto tiene el efecto de que ese puntero apuntará al elemento siguiente o anterior. Cuando se usa con variables que no son punteros, esas se modificarán simplemente en uno. Señalar que ++ siempre actúa después del cálculo de <varexp>, --, sin embargo, actúa _antes_.

```
  Ejemplos:    a++     -> devuelve el valor de a, y luego incrementa a.
               sp[]--  -> decrementa puntero sp en 4  (ARRAYs OF LONG),
                       -> y luego lee el valor al que apunta sp
```

- `<exp>
  Se le llama expresión entrecomillada, del LISP. <exp> no se evalúa, pero en su lugar devuelve la dirección de la expresión, que se puede evaluar más tarde cuando se necesite. En la sección 11 hay más sobre esta característica especial.

1.54 ch_4e

```
Triplete (IF THEN ELSE).
------------------------
```

El operador IF tiene una función similar a la sentencia IF, la única diferencia es que selecciona entre dos expresiones en vez de entre dos sentencias o bloques de sentencias. Igual que el operador x?y:z del C.

IF <expbool> THEN <exp1> ELSE <exp2>
retorna exp1 o exp2, dependiendo de expbool.

Por ejemplo, en vez de:
IF a<1 THEN b:=2 ELSE b:=3
IF x=3 THEN WriteF('x es 3\n') ELSE WriteF('x es otra cosa\n')
puedes escribir:
b:=IF a<1 THEN 2 ELSE 3
WriteF(IF x=3 THEN 'x es 3\n' ELSE 'x es otra cosa\n')

1.55 ch_4f

Estructura (.).

<ptrAobjeto>.<miembrodeobjeto> construye una <expvar>. El puntero debe declararse PTR TO <objeto> o ARRAY OF <objeto> y el miembro debe ser un identificador legal de objeto. Fijáte que el leer un subobjeto de un objeto de esta forma resulta en un puntero a ese objeto.

```
Ejemplos:      estatarea.userdata:=1
               rast:=mipantalla.rastport
```

Si el miembro que selecciona es de tipo PTR TO <tipo>, puedes usar "." y "[]" para referirte a nuevos valores. Si seleccionas un ARRAY o una subestructura en un OBJECT, el resultado es de nuevo un PTR.

1.56 ch_4g

Arreglos ([]).

<var>[<expíndice>] (es una <expvar>). Este operador lee el valor del arreglo al que apunta <var>, de índice <expíndice>. El índice puede ser prácticamente cualquier expresión.

Nota1: "[]" es una abreviatura de "[0]"
Nota2: en un arreglo de n elementos, el índice varía entre 0 .. n-1

```
Ejemplos:      a[1]:=10          -> pone el segundo elemento a 10
               x:=tabla[y*4+1]   -> lee del arreglo
               x.y[3]            -> accede a arreglo dentro de un objeto
```

1.57 ch_4h

```
Operador real (!).
------------------
```

<exp>!<exp>. Convierte una espresión de entero a real y viceversa, y sobrecarga los operadores + - * / = <> < > <= >= con equivalentes para reales.

1.58 ch_4i

```
Expresiones de asignación (:=).
-------------------------------
```

La asignación (dar un valor a una variable) existe como sentencia y como expresión. La única diferencia es que la versión como sentencia tiene la forma <expvar>:=<exp> y la expresión <var>:=<exp>. Esta última tiene el valor de <exp> como resultado.

Fíjate en que como <var>:= tiene lugar en una expresión, a menudo tendrás que ponerla entre paréntesis para forzar su correcta interpretación, como:

```
            IF mem:=New(100)=NIL THEN error()
```

se interpreta como:

```
            IF mem:=(New(100)=NIL) THEN error()
```

que no es lo que quieres hacer: mem debe ser un puntero, no un booleano. Lo que se debe escribir es:

```
            IF (mem:=New(100))=NIL THEN error()
```

Es una buena costumbre escribir entre paréntesis cualquier expresión de asignación que forme parte de otra, siempre que otras construcciones tales como "bla(a:=1)", "b:=a:=1" etc, no hayan eliminado la ambigüedad.

1.59 ch_4j

```
Secuenciamiento (BUT).
----------------------
```

El operador de secuenciamiento "BUT" permite la escritura de dos expresiones en una construcción que sólo permite una. A menudo, al escribir expresiones/llamadas a funciones complejas, a uno le gustaría realizar una segunda cosa sobre la marcha, como una asignación.

```
Sintaxis:   <exp1> BUT <exp1>
            implica: evalúa exp1, pero retorna el valor de exp2.

Ejemplo:    mifunc((x:=2) BUT x*x)
            assigna el valor 2 a x y luego llama a mifunc con x*x.
            Los () entorno a la asignación son necesarios para impedir
            que el operador := tome (2 BUT x*x) como expresión.
```

1.60 ch_4k

```
Reserva Dinámica de Memoria (NEW).
----------------------------------
```

El operador NEW es un poderoso operador para la reserva dinámica de memoria. Tiene varias formas:

(si DEF p:PTR TO cualquierobjeto, q:PTR TO INT)

```
      NEW p
```

es una expresión que reservará memoria para el tamaño del objeto al que apunta p, y la inicializa con 0. El puntero resultante se colocará en p, además de ser el valor de la expresión. Si NEW no puede obtener la memoria, éste lanzará una excepción "NEW". Por lo tanto, 'NEW p' es practicamente equivalente a:

```
      IF (p:=New(SIZEOF cualquierobjeto))=NIL THEN Raise("MEM")
```

con la diferencia de que p nunca recibe un valor si se lanza alguna excepción, y que lo primero es evidentemente una expresión, y no una sentencia.

Pero incluso hay más: se pueden reservar arreglos dinámicamente:

```
      NEW p[10]  -> arreglo de 10 objectos
      NEW q[a+1] -> arreglo de INT, con tamaño calculado en ejecución
```

(esto no funciona al instanciar clases)

Algunas veces el problema de expresiones de lista [1,2,3] es que son estáticas, y al usar éstas para construir estructuras de datos grandes sería conveniente crearlas en tiempo de ejecución. Se podría utilizar el equivalente dinámico de las listas:

```
      p:=[1,2,3]:cualquierobj        -> estructura estática
      p:=NEW [1,2,3]:cualquierobj    -> ¡reservado dinámicamente!
```

Esto funciona tanto con listas como con listas con tipo, y también con arreglos:

```
      NEW [1,2,3]       -> lista const, dinám. (nota: ¡no como varlista!)
      NEW [1,2,3]:obj   -> objeto
      NEW [1,2,3]:INT   -> arrreglo de INTs
```

La liberación de memoria reservada con cualquiera de las variaciones de NEW se realiza de forma automática al final del programa, o a mano con END o FastDispose().
(nota: la única excepción es NEW <lista>, que necesita FastDisposeList())

Si usas NEW [...]:obj, y el número de campos es menor que el que tiene el objeto, se añadirán campos 0/NIL/"\0" o lo que sea. Esto nos permite extender los objetos sin tener problemas con reservas como estas.
(fíjate en que esto es diferente de las [...]:obj estáticas)

Cuando usas NEW como sentencia, le pueden seguir un número indeterminado de punteros, por ejemplo, lo siguente es válido:

```
       NEW a,b.create(),c
```

1.61 ch_4l

```
Unificación (<=>).
------------------
```

La unificación permite un estilo de programación totalmente diferente, y que te será familiar si estas acostumbrado a la programación Lógica (ProLog), ecuacional o funcional (Miranda/Gofer/Haskell). La mayoría de nosotros, cuando usamos estructuras/arreglos para lo que sea, obtenemos valores de ellas por selección ("." y "[]"), sin embargo, en esos lenguajes se usa un patrón de concordancias.

En E:

```
      exp <=> exp_uni
```

exp puede ser cualquier expresión, aunque en v3 sólo es realmente útil si es de alguna manera un puntero a una lista. exp_uni es el patrón que se usa para hacer que concuerde (o unifique) exp, las variables toman sus respectivos valores. Si algo no concuerda, ninguna variable toma valor y la expresión tiene como resultado FALSE. En otro caso TRUE.

```
Ejemplo:    a:=[1,2,3]
            ...
            IF a <=> [1,x,y] THEN ...
```

Esto tendrá éxito con x=2 e y=3. Si la lista tuviera otra longitud que no fuera 2, la unificación también fallaría. El echo de que 'a' sea una lista es algo de lo que deberás asegurarte por tí mismo, EC simplemente intentará hacer que exp_uni encaje con el valor, sea cual sea, que obtenga de exp.

```
Ejemplos FALSE: a <=> [1,x]     -> longitud de lista errónea
                a <=> [1,4,x]   -> 4=2 falla
                'bla' <=> [1,2] -> resultado impredecible/¿caida sistema?
```

Lo divertido de la unificación en que puedes hacer unificaciones realmente complejas, y que si tomas el primer campo (u otro) como constante, indicando el tipo de estructura que representa, tienes una buena forma de tipos dinámicos. ¡Y todo el tiempo sin usar punteros!

```
Un ejemplo más interesante:
         [BLA,[1,'burp'],['bla',"bla"]] <=> [BLA,[1,x],y]
unifica:
         x='burp', y=['bla',"bla"]
o incluso:
    IF miexp <=> [PLUS,[MUL,a,1],[SUBS,[PLUS,c,d],e]] THEN RETURN a+c+d-e
```

Puede que sea un ejemplo tonto, pero uno se puede imaginar cosas como esta en el optimizador de código de un compilador. Es lo mismo que este fragmento de código tradicional:

```
            IF ListLen(miexp)=3
              IF myexp[]=PLUS
                IF ListLen(dummy:=miexp[1])=3
                  IF (dummy[]=MUL) AND (dummy[2]=1)
                    a:=dummy[1]
                    IF ListLen(dummy:=miexp[2])=3
```

```
                    IF dummy[]=SUBS
                      e:=dummy[2]
                      IF ListLen(dummy2:=dummy[1])=3
                        IF dummy2[]=PLUS
                          c:=dummy2[1]
                          d:=dummy2[2]
                          RETURN a+c+d-e
                        ENDIF
                      ENDIF
                    ENDIF
                  ENDIF
                ENDIF
              ENDIF
            ENDIF
          ENDIF
        ENDIF
```

Sólo que de una forma un poco más óptima. Como ves, hay mucho poder expresivo entorno a la unificación comparada con la programación tradicional basada en selección.

De momento, lo único permitido en la unificación son listas sin tipo, constantes (enteras), variables y Celdas-LISP. En el futuro veremos esto ampliado a cadenas, listas con tipo/objetos, e incluso expresiones [!¡]

1.62 ch_4m

```
Mutación de punteros (::).
--------------------------
```

Cuando escribes una expresión como 'p', donde p es un PTR TO objeto, éste te permite referirte a él como tal. El operador mutador de punteros te permite cambiar tal tipo en el momento (al vuelo):

```
        DEF p:PTR TO mp      -> puerto de mensajes

        p.sigbit             -> accede a él
        p::ln.name           -> accede al puntero como si fuera un nodo
```

Esto es muy útil para punteros que pueden apuntar a diferentes tipos de cosas u objetos que forman parte de otros, cuando, por supuesto, sólo es posible una declaración.

Una segunda forma de uso es en OBJECTos que tienen miembros declarados como LONG, que en realidad son punteros. Puedes darle un tipo al vuelo para hacer referencia a él de todas formas:

```
      miventana.userport::mp.sigbit
```

Aquí el OBJECTo ventana tiene un userport definido como LONG, por lo que no podrás hacer referencia a él normalmente.

1.63 ch_5

```
Sentencias.
-----------

         Formato                  (;)
         Etiquetas y saltos    (JUMP)
         Asignación              (:=)
         Mnemotécnicos de ensamblador
         Condicional             (IF)
         Para                   (FOR)
         Mientras             (WHILE)
         Repite              (REPEAT)
         Bucle                 (LOOP)
         Selección           (SELECT)
         Incremento         (INC/DEC)
         Expresiones void      (VOID)
         Liberación de memoria  (END)
```

1.64 ch_5a

```
Formato (;).
------------
```

Como se sugerió en el sobre el formato del código E, generalmente una sentencia se declara en una línea propia, aunque se pueden poner varias sentencias en un misma línea separándolas con un punto y coma, de la misma forma que una sóla sentencia se puede repartir entre varias líneas si se finaliza cada una de ellas con una coma ",".

```
Ejemplos:   a:=1; WriteF('¡hola!\n')
            DEF a,b,c,d,
                e,f,g

Las sentencias pueden ser:    - asignaciones
                              - condicionales, 'para' y similares.
                              - expresiones void
                              - etiquetas
                              - instrucciones en ensamblador
```

La coma es el principal caracter para indicar que no quieres finalizar la sentencia con el siguiente cambio de línea, aunque a partir de la versión 3, cualquier elemento (token) que no pueda ir legalmente al final de una línea provoca la continuación de la sentencia. Es más, si no han sido cerrados todos los "[" y "(" de un sentencia, ésta también continuará en las siguientes líneas.

```
Ejemplos de tales elementos:     + - * / =
                                 < >= <=
                                 := . <=> ::          -> hay otros, pero
                                 { [ (                -> estos son los
                                 AND OR BUT THEN IS   -> los más útiles.

                  /     a:=[
Ejemplo de código |         [1,2],
entre corchetes : |         [3,4]
                  \       ]      -> la asignación finaliza aquí.
```

1.65 ch_5b

```
Etiquetas y Saltos (JUMP).
--------------------------
```

Las etiquetas son identificadores de alcanze global que incorporan ':', como en:

```
        'mietiqueta:'
```

Se pueden utilizar con instrucciones como JUMP, y para hacer referencia a datos estáticos. Se pueden usar para salir de cualquier tipo de bucle (aunque no recomiendo esta técnica), pero no de los procedimientos. En los programas E normales se utilizan principalmente con el ensamblador en-línea. Las etiquetas siempre son visibles globalmente.

Sintaxis: JUMP <etiqueta>

Continúa la ejecuación en <etiqueta>. No es reocomendable usar esta instrucción, está disponible para ser utlizada sólo en casos en los que de otra forma se incrementaría la complejidad del programa.

```
Ejemplo: IF seacabo THEN JUMP termina
         /* otras partes del programa */

         termina:
```

1.66 ch_5c

```
Asigación (:=).
---------------
```

La forma básica de asignación es: <var> := <exp>

Ejemplos: a:=1, a:=mifunc(), a:=b*3

En E se pueden asignar varias variables de una vez, cuando <exp> es una función que devuelve varios valores de retorno.

1.67 ch_5d

```
Mnemotécnicos ensamblador.
--------------------------
```

El ensamblador en-línea es una parte real del lenguaje E, no necesita introducirse con bloques especiales "ASM" o similares, como es normal en otros lenguajes, y tampoco necesita un ensamblador independiente para ensamblar el código. Esto también signifca que obedece las reglas de sintaxis de E, etc.

```
Ejemplo:
         DEF a,b
         b:=2
```

```
        MOVEQ  #1,D0            /* algunas sentencias de ensamblador */
        MOVE.L D0,a             /* a:=1+b  */
        ADD.L  b,a
        WriteF('a=\d\n',a)      /* y escribirá 3 */
```

1.68 ch_5e

```
Conditionales (IF).
-------------------

IF, THEN, ELSE, ELSEIF, ENDIF

    Sintaxis:   IF <exp> THEN <sentencia> [ ELSE <sentencia> ]
    o:          IF <exp>
                  <sentencias>
                [ ELSEIF <exp>          /* puede haber varios ELSEIF */
                  <sentencias> ]
                [ ELSE ]
                  <sentencias>
                ENDIF
```

Construye un bloque condicional. Señalar que hay dos formas generales para esta sentencia, una versión de una línea y la otra de varias.

1.69 ch_5f

```
Para (FOR).
-----------

FOR, TO, STEP, DO, ENDFOR

    Sintaxis:   FOR <var> := <exp> TO <exp> STEP <paso> DO <sentencia>
    o:          FOR <var> := <exp> TO <exp> STEP <paso>
                  <sentencias>
                ENDFOR
```

Construye un bloque 'para', fíjate en las dos formas generales. <paso> puede ser cualquier constante positiva o negativa, excluyendo 0, y es opcional.

```
Ejemplo:    FOR a:=1 TO 10 DO WriteF('\d\n',a)
```

El cuerpo del FOR puede contener sentencias EXIT :

```
    Sintaxis:   EXIT <expbool>
```

que te permite salir del bucle si <expbool> es cierta.

1.70 ch_5g

```
Mientras (WHILE).
-----------------

WHILE, DO, ENDWHILE

    Sintaxis:  WHILE <exp> DO <sentencia>
    o:         WHILE <exp>
                 <sentencias>
               ENDWHILE
```

Construye un bloque 'mientras', el cual se repite siempre que <exp> sea TRUE. Fíjate en la versión de una-línea/una-sentencia y la versión de varias líneas.

WHILE también puede contener sentencias EXIT, como el FOR.

1.71 ch_5h

```
Repite (REPEAT).
----------------

REPEAT, UNTIL

    Sintaxis:  REPEAT
                 <sentencias>
               UNTIL <exp>
```

Construye un bloque 'repite-hasta': el bloque se ejecutará hasta que <exp>=TRUE. (Señalar que el bloque siempre se ejecutará al menos una vez)

```
Ejemplo:
      REPEAT
          WriteF('¿Estas seguro, seguro de querer salir del programa?\n')
          ReadStr(stdout,s)
      UNTIL StrCmp(s,'¡Si, por favor!')
```

1.72 ch_5i

```
Bucle (LOOP).
-------------

LOOP, ENDLOOP

    Sintaxis:  LOOP
                 <sentencias>
               ENDLOOP
```

Construye un bucle infinito.

1.73 ch_5j

```
Selección (SELECT).
-------------------

SELECT, CASE, DEFAULT, ENDSELECT

    Sintaxis:  SELECT <var>
               [ CASE <exp>
                 <sentencias> ]
               [ CASE <exp>
                 <sentencias> ]   /* cualquier número de bloques */
               [ DEFAULT
                 <sentencias> ]
               ENDSELECT
```

Construye un bloque de selección. Se contrastarán varias expresiones con la variable, y sólo se ejecutará el primer bloque que la satisfaga. El bloque por omisión (DEFAULT) se ejecutará si la variable no satisface a ninguna de las expresiones.

```
      SELECT caracter
        CASE 10
          WriteF('Ejem, he encontrado un cambio de línea\n')
        CASE 9
          WriteF('¡Vaya, esto debe ser un tabulador!\n')
        DEFAULT
          WriteF('¿Conoces éste: "\c" ?\n',caracter)
      ENDSELECT
```

Además de el 'SELECT <var>' de siempre, que trabaja con expresiones en las sentencias CASE, E tiene un 'SELECT <maxrango> OF <exp>' que funciona con constantes y/o rangos de constantes en el CASE. No sólo es más potente para muchas aplicaciones, si no que es mucho más rápido para un número de casos grande (generalmente >5), o si los casos son igualmente probables. Sin embargo supone que todos los CASEs caen dentro de un rango de enteros pequeño desde 0 a n-1, donde n es un número razonable, por ejemplo 10, o 255 para caracteres.

```
Ejemplo:
      SELECT 127 OF FgetC(stream)
        CASE "\n","\b"
          WriteF('fin de línea\n')
        CASE "\t"," "
          WriteF('espacio blanco\n')
        CASE "0" TO "9"
          WriteF('Entero\n')
        CASE "A" TO "Z", "a" TO "z", "_"
          WriteF('Identificador\n')
        DEFAULT
          WriteF('otros caracteres\n')
      ENDSELECT
```

DEFAULT se ejecutará no sólo para aquellos valores para los que no hay CASE, si no también para los que estén fuera del rango, (en este ejemplo, de 128 a 255 y >255 o <0).

Señalar que la velocidad tiene un precio: ya que este SELECT usa una tabla de saltos para acceder rápidamente al CASE correcto, usará 2*<maxrango> bytes, 256 en este ejemplo.

1.74 ch_5k

```
Incremento (INC/DEC).
---------------------

INC, DEC

    Sintaxis:  INC <var>
               DEC <var>
```

Es una abreviatura para <var>:=<var>+1 y <var>:=<var>-1. Con la única diferencia de que las primeras son sentencias y no retornan un valor.

1.75 ch_5l

```
Expresiones void (VOID).
------------------------

VOID

    Sintaxis:  VOID <exp>
```

Calcula la expresión sin poner el resultado en lugar alguno. Sólo es útil para una sintaxis más clara, ya que de todas formas, en E las expresiones se pueden usar como sentencias sin usar VOID. Sin embargo, esto puede causar errores difíciles de apreciar, ya que, por ejemplo, "a:=1" asigna el valor 1 a 'a', pero "a=1" no resulta en nada como sentencia. E te indicará con una advertencia si eso ocurre.

1.76 ch_5m

```
Liberación de memoria (END).
----------------------------
```

END es el complemento de NEW. Cualquier puntero obtenido con NEW se debe liberar con (y sólo con) END.

```
    Sintaxis:  END a
   o incluso:  END a,b,c
```

donde los argumentos son PTRs (punteros) a algún tipo. END libera el espacio de memoria al que se está apuntando, de forma que si 'a' es un PTR TO LONG, sólo liberará 4 bytes. Esto quiere decir que si reservaste 'a' con NEW a[NUM], lo debes liberar con END a[NUM].

NEW acepta de forma segura punteros NIL. Los punteros también se vuelven a NIL tras su liberación.

Si 'a' es un PTR a un objeto que es un instancia de una clase, END obtendrá dinámicamente el tamaño del objeto que se desea liberar de la clase, de forma que si tienes un objeto de clase 'b' de 32 bytes, pero el puntero con el que lo vas a liberar es de una clase base (sólo 24 bytes), END liberará correctamente los 32 bytes. Esto sólo funciona con clases.

Puedes imaginar END p como una macro de:

```
IF p
  p.end()
  FastDispose(p,p.classobject.virtuallen)
  p:=NIL
ENDIF
```

Señalar que NEW y END utilizan las funciones FastNew() y FastDispose() (descritas en algún otro lugar) que trabajan de forma bastante distinta a como lo hacen NewR() y Dispose().

1.77 ch_6

```
Definición y declaración de funciones.
--------------------------------------

        Definición de procedimientos y argumentos (PROC)
        Definiciones locales y globales: alcance   (DEF)
        ENDPROC/RETURN
        Función 'main'
        Variables predefinidas del sistema
        Argumentos por omisión de funciones
        Valores de retorno múltiples
        Valores de función
```

1.78 ch_6a

```
Definición de procedimientos y argumentos (PROC).
-------------------------------------------------
```

Puedes usar PROC y ENDPROC para agrupar sentencias formando tus propias funciones. Tales funciones pueden tener cualquier número de argumentos, y varios valores de retorno.

```
PROC, ENDPROC

    Sintaxis:  PROC <etiqueta> ( <args> , ... )
               ENDPROC <valorretorno>, ...
```

Define un procedimiento con cualquier número de argumentos. Los argumentos son de tipo LONG, u opcionalmente de tipo PTR TO <tipo> y no necesitan ningún tipo de declaración. Con ENDPROC se denota el final del procedimiento. Si no se dá un valor de retorno, se retorna 0.

```
Ejemplo:  función que retorna la suma de dos argumentos:
               PROC suma(x,y)        /* x e y son variable locales */
```

```
              ENDPROC x+y           /* retorna el resultado  */

más corto:    PROC suma(x,y) IS x+y
```

1.79 ch_6b

```
Definiciones locales y globales: alcance (DEF).
-----------------------------------------------
```

Con la sentencia DEF puedes definir variables locales adicionales aparte de aquellas que son argumentos. La forma más sencilla es con:

```
     DEF a,b,c
```

que declara los identificadores a, b y como variables de tu función. Señalar que tales declaraciones deben estar al pricipio de tu función.

```
DEF

    Sintaxis:  DEF <declaraciones>,...
```

declara variables. Una declaración tiene una de las siguiente formas:

```
          <var>
          <var>:<tipo>              donde <tipo>=LONG,<identobjeto>,...
          <var>[<tama>]:<type>      donde <tipo>=ARRAY,STRING,LIST
```

Desde ahora usaremos la forma <var>.
Los argumentos de funciones se restringen a tipos básicos.
La declaración de un tipo básico puede tener una inicialización, en la versión actual ésta debe ser un entero (no una expresión):

```
           DEF a=1,b=2
```

Un programa está formado por un conjunto de funciones, llamados procedimientos, PROCs. Cada procedimiento puede tener variables Locales, y el programa como un todo puede tener variables Globales. Al menos uno de los procedimientos del programa debe ser el 'PROC main()', ya que éste es el módulo en el que comienza la ejecución. Un programa sencillo puede paracerse a:

```
         DEF a, b          /* definición de variables globales */

         PROC main()       /* las funciones en orden aleatorio */
            bla(1)
         ENDPROC

         PROC bla(x)
            DEF y,z        /* posiblemente variables locales propias */
         ENDPROC
```

Para resumir, las definiciones locales son las que se hacen al comienzo de los procedimientos, y que sólo son visibles dentro de esa función. Las definiciones Globales se realizan antes del primer procedimiento, de tu código fuente, y son visibles globalmente. Puede haber variables locales con el mismo nombre que variables globales (y por supuesto, lo mismo con variables locales de dos funciones diferentes), en cuyo caso, las variables locales tienen prioridad.

1.80 ch_6c

```
ENDPROC/RETURN.
---------------
```

Como se indicó con anterioridad, ENDPROC marca el final de la definición de una función, y puede retornar un valor. De forma opcional, RETURN se puede usar en cualquier punto de la función para salir de ella, y si se usa en main() provocará la finalización del programa.

```
    Sintaxis:  RETURN [<valorretorno>]

Ejemplo:
      PROC obtenrecursos()
         /* ... */
         IF error THEN RETURN FALSE  /* algo fue mal ---> sal y falla */
         /* ... */
      ENDPROC TRUE   /* llegamos hasta aquí, entonces retorna TRUE */
```

Una versión muy corta de la definición de una función es:

```
   PROC <etiqueta> ( <arg> , ... ) RETURN <exp>
   (o "IS" en vez de "RETURN")
```

Estas son definiciones de funciones que sólo realizan pequeñas computaciones, como funciones factoriales y parecidas: (one-liners :-)

```
        PROC fac(n) IS IF n=1 THEN 1 ELSE fac(n-1)*n
```

1.81 ch_6d

```
Función 'main' .
----------------
```

El PROC llamado 'main' tiene importancia porque es la primera función que se ejecuta; se comporta como las demás funciones, y también puede tener variables locales. 'main()' no tiene argumentos: los argumentos de la línea de comandos se ofrecen en la variable de sistema 'arg', y también se pueden analizar con ReadArgs() (función de dos.library)

1.82 ch_6e

```
Variables del sistema predefinidas.
-----------------------------------
```

Las siguiente variables globales siempre están disponibles en tu programa, se les llama variables del sistema.

```
arg  .........  Como se dijo antes,  tiene un puntero a un cadena acabada
                en cero, con los argumentos de la línea de comandos.
                No uses esta variable si prefieres usar ReadArgs().
stdout  ......  Contiene el file-handle  para la salida estandard.  Si tu
                programa fue iniciado desde el workbench, de forma que no
                hay  salida-de-shell  disponible,  WriteF()  abrirá  una
```

```
                ventana CON: para tí y pondrá su file-handle en una esta.
stdin  .......  file-handle para la entrada estandard.
conout  ......  Aquí  es  donde se pone el  file-handle,  y la ventana de
                consola se cerrará de forma automática cuando tu programa
                finalice. (mira en WriteF() sobre como utilizar estas dos
                variables de forma apropiada.)
execbase,  ...  Estas  cuatro  variables se ofrecen siempre inicializadas
dosbase,        con el valor apropiado.
gfxbase,
intuitionbase
stdrast  .....  Puntero al rastport standard para usarlo con tu programa,
                o NIL.  La usan las funciones gráficas internas (Line()).
wbmessage  ...  Puntero a mensaje recibido si el programa se inició desde
                WB.  Se puede usar como booleano para detectar si el pro-
                grama se inició desde el WB,  o para comprobar  cualquier
                argumento  que se seleccionó con shift junto con el icono
                de la aplicación.  Mira WbArgs.e en directorio  Src/Args/
                para ver como utilizar de forma correcta esta variable.
```

1.83 ch_6f

```
Argumentos por omisión de funciones.
------------------------------------
```

Los argumentos por omisión permiten especificar un valor por omisión para uno o más argumentos de una función, para el caso en el que la función se llama con menos argumentos que parámetros. Por ejemplo, un procedimiento como:

```
      PROC bla(a,b=1,c=NIL)
se puede llamar como:                   que equivale a:
         bla(a,b,c)                       bla(a,b,c)
         bla(a,b)                         bla(a,b,NIL)
         bla(a)                           bla(a,1,NIL)
```

Esto puede ser muy útil, y también puede expresar algo sobre la función del procedimiento, por ejemplo, que la mayoría de las veces se le llamará con NIL, así que, ¿por qué no suprimir ese argumento de la llamada por claridad?

Esa también es una razón para no sobreutilizar los argumentos por omisión: no empieces a dejar fuera valores no esenciales para los procedimientos por mera pereza, si piensas que un parametro no tiene realmente un valor por defecto.

Para impedir que las llamadas con menos argumentos sean ambíguas, sólo pueden declararse como arguementos por omisión los últimos 0..n parámetros de un PROC de n parámetros.

Por ejemplo, es ilegal: PROC bla(a,b=1,c)
(en esos casos, ¡siempre puedes reordenar los parámetros!).

Los argumentos que se proporcionan en una llamada se rellenan de izquierda a derecha, y los argumentos que falten se añaden con argumentos por omisión según sea necesario.

1.84 ch_6g

```
Valores de retorno múltiples.
-----------------------------

   En E puedes retornar cualquier número de valores de retorno (máx. 3 en
Amiga E por razones de implementación). ¿Cómo?

    Sintaxis:  RETURN [<exp>,<exp>,<exp>]       (o ENDPROC, por supuesto)

Ejemplo:    PROC sincos(rad)
               DEF sin,cos
               /* cualquier computación necesaria */
            ENDPROC sin,cos
se llama con:
            s,c:=sincos(3.14)
            s:=sincos(1.00)

   Como puedes ver, hay una nueva sentencia de la forma:
                     <var> , ... := <exp>
donde <exp> prácticamente sólo tiene sentido como llamada a función.

Señalar dos cosas:
- Tu  decides el  número de  argumentos que  quieres recibir.  Esto tiene
  sentido  cuando  el primer  valor  de  retorno  es  el pricipal,  y  el
  segundo/tercero  representan  información  adicional,  que  puede  ser
  importante sólo para algunos llamadores.
- Esta forma  es una  _sentencia_.  Esto  significa que  cuando llamas  a
  sincos()  como parte  de  otra expresión,  sólo se  usa  el primer  (el
  regular) valor de retorno: fun(sincos(1.0))
```

1.85 ch_6h

```
Valores de función.
-------------------

   Con v3,  también  puedes tener  funciones como  valores, y  usar éstos
líbremente.  Son diferentes de las expresiones entrecomilladas, ya que se
llaman como los procedimientos normales.
Ejemplo:    fun:={miproc}      -> obtener ptr a PROC
            ...
            fun(1,2,3)         -> aplicar a args, como un PROC normal

Notas:
- Debes  asegurarte de  que el PROC  del que tienes  el puntero  tiene el
  mismo número de argumentos que los que le vas a aplicar.  El compilador
  no puede comprobar esto por tí.
- Peor  tadavía: debes  asegurarte de  que el puntero  es en  realidad un
  puntero a PROC.  Hay una advertencia del compilador que te puede ayudar
  con esto.
```

1.86 ch_7

```
Declaración de constantes.
--------------------------

     Constantes      (CONST)
     Enumeraciones    (ENUM)
     Conjuntos         (SET)
     Constantes predefinidas
```

1.87 ch_7a

```
Constantes (CONST).
-------------------

    Sintaxis:  CONST <declaraciones>,...

   Te permite declarar una constante. Una declaración es como sigue:
        <ident>=<valor>

   Deben ir en mayúsculas, y el resto del program las trata como <valor>.
Ejemplo:    CONST MAX_LINEAS=100, ER_NOMEM=1, ER_NOFICHERO=2

   No puedes declarar constantes en términos de otras que se estén
declarando en la misma sentencia CONST, ponlas en la siguiente.

   Las declaraciones CONST, ENUM y SET siempre son globales, es decir, no
se pueden declarar constantes locales a un procedimiento.  El mejor lugar
para  poner las declaraciones de constantes es al principio de tu fuente,
aunque EC también te permite ponerlas entre dos PROCs, por ejemplo.
```

1.88 ch_7b

```
Enumeraciones (ENUM).
---------------------

   Las  enumeraciones  son  un  tipo específico de constante al que no es
necesario darle valores, simplemente están en un rango entre 0..n, siendo
la  primera  0.   De  todas  formas,  puedes usar la notación =<valor> en
cualquier punto de la enumeración para cambiar o reiniciar el contador.

Ejemplo:       ENUM CERO, UNO, DOS, TRES, LUNES=1, MARTES, MIERCOLES
               ENUM ER_NOFICHERO=100, ER_NOMEM, ER_NOVENTANA
```

1.89 ch_7c

```
Conjuntos (SET).
----------------
```

Los conjuntos son también como las eumeraciones, con la diferencia de que en lugar de incrementar el valor (0,1,2,...) incrementan el número de bit (0,1,2,...) resultando en valores como (1,2,4,8,...). Esto tiene la ventaja añadida de que se pueden usar como conjuntos de banderas, tal y como dice la palabra clave.

Imagínate un conjunto como el siguiente para describir las propiedades de una ventana:

```
SET SIZEGAD,CLOSEGAD,SCROLLBAR,DEPTH
```

entonces, para inicializar una variable con propiedades DEPTH y SIZEGAD:

```
baderasvent:=DEPTH OR SIZEGAD
```

y para incluir una bandera SCROLLBAR adicional:

```
baderasvent:=banderasvent OR SCROLLBAR
```

o para comprobar si alguna o ambas propiedades están activas:

```
IF banderasvetn AND (SCROLLBAR OR DEPTH) THEN  -> lo que sea
```

1.90 ch_7d

```
Constantes predefinidas.
------------------------
```

Las siguientes constantes predefinidas se pueden usar en cualquier momento:

```
TRUE,FALSE      Representan los valores booleanos (-1,0)
NIL             (=0), el puntero sin incializar.
ALL             Usada con funciones de cadenas como StrCopy() para copiar
                todos los caracteres.
GADGETSIZE      Tamaño mínimo en bytes de un gadget ( Gadget() ).
OLDFILE,NEWFILE Parámetros de modo para usar con Open()
EMPTY           Usado con métodos (puede convertirse en palabra clave)
STRLEN          Tiene la longitud de la última cadena inmediata usada.
                Ejemplo:
                          Write(handle,'¡Hola colegas!',STRLEN)  -> =14
```

1.91 ch_8a

```
El Sistema de Tipos.
--------------------
```

E no posee un sistema de tipos (o sistema de tipos de datos) rígido, como el Pascal o el Modula2, incluso es más flexible que el sistema de tipos del C. Esto debe tenerse bien en cuenta, junto con la filosofía de que en E todos los tipos de datos son iguales: todos los valores básicos pequeños como caracteres, enteros, ..., tienen el mismo tamaño de 32 bits, y todos los demás tipos de datos, como los arreglos y las cadenas están representados por punteros de 32 bits a ellos. De esta forma, el compilador de E puede generar código de una forma muy polimórfica.

Las (des)ventajas son obvias:

- Menor control por parte del compilador sobre errores tontos.

```
   Ventajas:
- Polimorfismo a bajo nivel, crea facilmente potentes funciones genéricas
- Programación  flexible :  sin problemas con tipos de valores de retorno
  que no coinciden y sin "mutaciones" ni mensajes de error superfluos.
- Sin  errores difíciles de encontrar al mezclar  tipos de datos de dife-
  rentes tamaños.

        Básico                 (LONG/PTR)
        Simple            (CHAR/INT/LONG)
        Arreglo                   (ARRAY)
        Complejo            (STRING/LIST)
        Compuesto               (OBJECT)
        Inicialización
        Esencia del sistema de tipos de E
```

1.92 ch_8b

```
Básico (LONG/PTR).
------------------

   Sólo  hay  un tipo de variable básico --no complejo-- en E, el cual es
el tipo LONG de 32 bits.  Por ser el tipo por omisión, se declarara con:

    Sintaxis:  DEF a:LONG     o simplemente:      DEF a

   Este  tipo de  variable puede  contener lo  que se  conoce como  tipos
CHAR/INT/PTR/LONG en otros lenguajes.  Una variación especial de  LONG es
el  tipo  PTR.  Este tipo es compatible con LONG, con la única diferencia
de que indica a qué tipo apunta el puntero.  Por omisión, el tipo LONG se
especifica como PTR TO CHAR.

    Sintaxis:  DEF <var>:PTR TO <tipo>
    donde <tipo> es o un tipo simple o uno compuesto.

Ejemplo:    DEF x:PTR TO INT, mipantalla:PTR TO screen

   'screen'  es el nombre de un objeto definido en 'intuition/screens.m'.
Por ejemplo, si abres tu propia pantalla con:
        mipantalla:=OpenS(...)   etc.
puedes  usar  el puntero 'mipantalla' como en 'mipantalla.rastport'.  Sin
embargo,  si  no  quieres  hacer  nada  con  la  variable  hasta llamar a
CloseS(mipantalla), puedes declararla simplemente como:
        DEF mipantalla

   Las   declaraciones   de   variables   pueden  tener  inicializaciones
opcionales, pero sólo constantes enteras, no expresiones completas:
        DEF a=1, b=NIL:PTR TO textfont
```

1.93 ch_8c

```
Simple (CHAR/INT/LONG).
-----------------------
```

Los tipos simples CHAR (8bit) e INT (16bit) no se pueden usar como tipos para una variable básica; la razón debe estar clara ya. Sin embargo, se pueden usar como tipo de dato para construir ARRAYs de ellos, definir punteros a ellos, usarlos en la definición de OBJECTos, etc. Miras los ejemplos de éstos últimos.

1.94 ch_8d

```
Arreglos (ARRAY).
-----------------
```

Los arreglos de declaran indicando su longitud (en bytes):

```
DEF b[100]:ARRAY
```

esto define un array de 100 bytes. Internamente 'b' es una variable de tipo LONG y un puntero a esa área de memoria.

El tipo por omisión de un elemento de arreglo es CHAR, aunque puede ser cualquier cosa especificando:

```
DEF x[100]:ARRAY OF LONG
DEF mismenus[10]:ARRAY OF newmenu
```

donde "newmenu" es un ejemplo de estructura, llamados OBJECTos en E. El acceso a los arreglos es muy sencillo, con <var>[<exps>]:

```
b[1]:="a"
z:=mismenus[a+1].mutualexclude
```

Señalar que el índice de un arreglo de tamaño 'n' se mueve entre 0 y 'n-1', y no entre 1 y 'n'. Y también que ARRAY OF <tipo> es compatible con PTR TO <tipo>, con la única diferencia de que la variable que es un ARRAY se encuentra inicializada.

1.95 ch_8e

```
Complejo (STRING/LIST).
-----------------------
```

- Las cadenas (STRING), son similares a los arreglos, aunque se diferencian de ellos en que sólo se pueden modificar usando funciones de cadena de E; y en que contienen información sobre su longitud de forma que las funciones de cadena pueden modificarlas de una forma segura, es decir: la cadena nunca puede crecer más allá del área de memoria en la que está.

  Declaración: DEF s[80]:STRING

  El tipo de datos STRING (llamado cadenaE o 'estring') es compatible con PTR TO CHAR, y también con ARRAY OF CHAR, pero no en sentido opuesto.

- Las LISTas se pueden entender como una mezcla entre STRING y ARRAY OF LONG. Es decir: esta estructura de datos mantiene una lista de variables LONG que se puede extender y acortar como los STRINGs.

  Declaración: DEF x[100]:LIST

  Una poderosa adición a este tipo de datos es que también tiene un equivalente constante [], de la misma forma que los STRINGs tienen ''.

```
  Una lista es compatible con PTR TO LONG,  y por supuesto ARRAY TO LONG,
  aunque no en sentido contrario.
```

1.96 ch_8f

```
Compuesto (OBJECT).
-------------------

   Los OBJECT son como una estructura/clase en C/C++ o RECORD en Pascal.

Ejemplo:    OBJECT miobj        -> define un estructura de datos
              a:LONG            -> formada por tres elementos.
              b:CHAR
              c:INT
            ENDOBJECT

    Sintaxis:  OBJECT <nombreobj>
                 <nombremiembro> [ : <tipo> ]    -> cualquier número
               ENDOBJECT
donde <tipo> es uno de los siguientes:
                    CHAR/INT/LONG/<objeto>
                    PTR TO CHAR/INT/LONG/<objeto>
                    ARRAY OF CHAR/INT/LONG/<objeto>
                    (ARRAY es abreviatura de ARRAY OF CHAR)

   Al igual que las declaraciones DEF, omitir el tipo implica :LONG.

   Destacar  que <nombremiembro> no necesita ser un identificador único y
puede aparecer en otros objetos.  Hay muchas formas de usar objetos:
         DEF x:miobj                  -> x es una estrutura
         DEF y:PTR TO miobj           -> y es un puntero a la estrutura
         DEF z[10]:ARRAY OF miobj

         y:=[-1,"a",100]:miobj        -> listas con tipo
         IF y.b="a" THEN /* ... */
         z[4].c:=z[d+1].b++

   Los  elementos de los objetos se redondean a tamaños pares, y se ponen
en offsets pares:
         OBJECT micadena
           lon:CHAR, datos[9]:ARRAY
         ENDOBJECT
SIZEOF de 'micadena' es 12, y "datos" empieza con un offset de 2.

   'PTR TO'  es el único  tipo en OBJECTos  que puede hacer  referencia a
otros objetos que no hayan sido declarados todavía.

(Hecha un vistazo a las características OO de los OBJECTos.)
```

1.97 ch_8g

```
Inicialización.
---------------

1. Siempre se inicializan a NIL (u otra cosa si se indica)
   - Variables globales
     NOTA: con vistas a documentación,  siempre es mejor escribir  `=NIL'
           en definiciones de variables se espera que sean NIL.
2. Se inicializan a '' y [] resp.
   - STRINGs globales y locales
   - LISTas globales y locales
3. No se inicializan
   - variables locales (a no ser que se indique explicitamente)
   - ARRAYs globales y locales
   - OBJECTos globales y locales
```

1.98 ch_8h

```
Esencia del sistema de tipos de E.
----------------------------------
```

Esta sección intenta explicar cómo funciona el sistema de tipos de E desde otra perspectiva.

La mayor parte de los problemas que tiene la gente al programar en E provienen de una visión incorrecta del funcionamiento del sistema de tipos de E. Generalmente se tiene una idea sobre el funcionamiento de los tipos que depende del lenguaje de programación usado con anterioridad y se intenta aplicarla a E, lo que suele ser fatal, ya que E es bastante diferente en lo que se refiere a tipos.

El Sistema de Tipos.

E es, en esencia, un lenguaje SIN-TIPOS. De hecho, las variables pueden tener un tipo, aunque sólo sirve para indicar la forma de hacer referencia a la variable cuando se usa como puntero. En practicamente todas las demás construcciones del lenguaje las variables se tratan como si todas tuvieran el mismo tipo, un valor sin tipo de 32 bits.

En la práctica esto significa que, excepto expresiones con los operadores ".", "[]" y "++", todos los demás operadores y funciones trabajan con valores de 32bits, independiente que si representan booleanos, enteros, reales o punteros a algo.

Tipos punteros.

En el sistema de tipos de E sólo hay 4 tipos, PTR TO CHAR, PTR TO INT, PTR TO LONG y PTR TO <objeto>, donde <objeto> es el nombre de un OBJECTo definido con anterioridad. Cuando una variable (o miembro de un objeto como veremos más tarde) se declara de este tipo, significa que si la variable contiene un valor que es un puntero legal, esa es la forma de hacer referencia a ella.

LONG, ARRAY etc.

Todos los demás tipos que podamos ver en una declaración DEF no son tipos reales, ya que no son más que otras formas de escribir uno de los cuatro anteriores. Por ejemplo, ARRAY OF <tipo> es otra forma de poner

PTR TO <tipo>, con la única diferencia de que la primera recibe de forma automática la dirección de un área de memoria en la pila que es suficientemente grande como para contener los datos para el número de elementos especificado entre corchetes.

Esta es una tabla que muestra todos los 'tipos' de E en términos de los cuatro básicos:

ARRAY OF CHAR, ARRAY, STRING, LONG	(equivalen a)	PTR TO CHAR
ARRAY OF INT	(equivale a)	PTR TO INT
ARRAY OF LONG, LIST	(equivalen a)	PTR TO LONG
ARRAY OF <objeto>, <objeto>	(equivalen a)	PTR TO <objeto>

- LONG son variables que no están pensadas para ser usadas como puntero, por ejemplo, enteros. Su equivalencia con PTR TO CHAR es bastante lógica, ya que ambos hablan conceptualemente de cosas que se miden en unidades de 1. (por ejemplo, "++" tiene el mismo efecto en ambos)
- LIST y STRING son lo mismo que sus equivalentes en ARRAY respecto al hecho de que se incializan a un trozo de la pila, aunque su representación es algo más compleja para facilitar la comprobación de rango en tiempo de ejecución (cuando se usan con funciones apropiadas).
- un <objeto> es equivalente a [1]:ARRAY OF <objeto>. Ambos representan un PTR TO <objeto> inicializado.

Dentro de un OBJECT pueden aparecer estas mismas declaraciones, además de CHAR e INT, y no se permite declarar LIST ni STRING, ya que estos últimos son objetos complejos en sí mismos, y no pueden formar parte de un objeto.

Referencia.
Dado un puntero de algún tipo,

"[]" puede indicar otros elementos que están ordenados secuencialmente a continuación del elemento al que el puntero está apuntando en ese momento. Señalar que esto permite índices tanto positivos como negativos, y tampoco se hacen suposiciones sobre donde y cuantos elementos se han reservado en realidad.

"++" hace que el puntero apunte al siguiente elemento en memoria, y "--" al anterior. Señalar que estos operadores siempre actúan en el puntero, y nunca sobre el elemento al que apuntan.

"." funciona de forma similar a "[]", sólo que indexa el puntero por nombre, es decir, el puntero debe ser un PTR TO <objeto>.

"[]" y "." se pueden concatenar a un puntero p en cualquier secuencia, siempre que se sepa que el valor resultante anterior es de nuevo de tipo "PTR TO".

No siempre es necesario escribir una referencia como en otros lenguajes. Por ejemplo, si p es un ARRAY OF obj, en lugar de escribir p[indice].miembro, puedes escribir simplemente p[indice] lo cual resulta en la dirección de ese objeto. Esto también explica porqué p[].miembro es equivalente a p.miembro, ya que p[] es lo mismo que p cuando éste apunta a un objeto.

Semática de Referencia.
Otro concepto relacionado con los tipos que hace a E de alguna forma diferente de otros lenguajes --y por tanto dificil de entender-- es su acento en la Semántica de Referencia en vez de la Semántica de Valor. Intentaré argumentar aquí los benefícios de la primera.

Informalmente, Semántica de Referencia significa que los objetos en un lenguaje (principalmente los que no son simples como los LONG) se representan mediante punteros, mientras que la Semántica de Valor trata esos objetos como si fueran ellos mismos. Un ejemplo de lenguaje que sólo usa Semántica de Valor es el BASIC, ejemplos de lenguajes que tienen las dos son lenguajes del tipo del C/C++ y del Pascal, y ejemplos de lenguajes con sólo Referencia son los nuevos lenguajes Orientados a Objetos, lenguajes funcionales como el LISP, y por supuesto E.

El uso de Semántica de Referencia no implica estar ocupado todo el tiempo con punteros, es más, te preocuparás de ellos con mucha menos asiduidad que en el caso de lenguajes con semántica mezclada o el caso con sólo valor, debido principalmente a que la mayoría de las estructuras de datos no triviales se reservan de forma dinámica, lo cual implica punteros. El mejor ejemplo de esto es el LISP, en el que se programa fuertemente con punteros sin darse cuenta. En E, uno se puede olvidar facilmente de que STRING es un puntero, debido a la facilidad con que se puede pasar entre funciones; en C son necesarias a veces gran cantidad de "&" cuando en el equivalente en E no se necesitan, y el equivalente en Oberon de bla('hola') se parece a bla(sys.ADR('hola')) debido a que las cadenas no representan un puntero, si no un valor como un todo...

1.99 ch_9

Funciones Predefinidas.

E/S (Entrada/Salida)
Cadenas y funciones de cadenas
Listas y funciones de listas
Intuition
Gráficos
Sistema
Matemáticas y otras
Enlace de cadenas y listas
Celdas-lisp y funciones de celdas

1.100 ch_9a

Entrada/Salida.

WriteF(cadenaformato,args,...)
PrintF(cadenaformato,args,...)

Imprime una cadena (que puede contener códigos de formato) en stdout.

Pueden incorporar un número ilimitado de argumentos. Señalar que, como las cadenas de formato se pueden crear dinámicamente, no es posible cerciorarse de que el número de argumentos sea correcto.
Ejemplos: WriteF('¡Hola Mundo!\n') -> escribe cadena con cambio de línea
WriteF('a = \d \n',a) -> escribe: "a = 123", si a era 123

NOTA: si stdout=NIL, por ejemplo, y tu programa fue iniciado desde el Workbench, WriteF() creará una ventana de salida y pondrá el handle en conout y stdout. Esa ventana se cerrará de forma automática al salir del programa después de que el usuario pulse <return>. WriteF() es la única función que abre esta ventana, de modo que si quieres hacer E/S en stdout, y quieres estar seguro de que stdout<>NIL, haz un "WriteF('')" como primera instrucción de tu programa para asegurar la salida. Si quieres abrir la ventana de consola por tí mismo, debes hacerlo poniendo el handle de fichero resultante en las variables 'stdout' y 'conout', ya que de esa forma la ventana se cerrará de forma automática al finalizar el programa. Si deseas cerrar esa ventana de forma manual, asegúrate de poner 'conout' de nuevo a NIL de nuevo para indicar a E que no hay una ventana de consola para cerrar.
PrintF() es igual que Write(), sólo que utiliza ES con buffers. Ambas retornan la longitud de la cadena que se acaba de imprimir.

```
Out(handlefich,car)
car:=Inp(handlefich)
```

Escriben o leen un sólo byte a/de algún fichero o stdout. Si car=-1 entonces se alcanzó el final del fichero (EOF), o sucedió algún error. Out() retorna el número de bytes que se escribieron en realidad (si <>1 entonces hubo un error).

```
long:=FileLength(cadenanombre)
```

Determina la longitud del fichero que desees leer, y también, si tal fichero existe (retorna -1 si hubo un error, o el fichero no existe).

```
ok:=ReadStr(handlefich,cadenae)
```

(mira Cadenas y funciones de cadenas)

```
salant:=SetStdOut(nuevostdout)
entant:=SetStdIn(nuevostdin)
```

Fija la variable de salida/entrada estandard 'stdout'/'stdin'. Equivale a 'salant:=stdout; stdout:=newstdout'

1.101 ch_9b

Cadenas y funciones de cadenas.

E tiene el tipo de datos STRING. Este es una cadena, que desde ahora llamaremos 'cadenaE' (Estring), que se puede modificar y cambiar de tamaño, al contrario que las 'cadenas' normales, que es como nos referiremos aquí a cualquier secuencia terminada en cero. Las cadenasE son compatibles con las cadenas, aunque no en sentido inverso, de forma

que si un argumento necesita una cadena, podemos pasar cualquiera de las dos. Si se necesita una cadenaE, no uses cadenas normales.

```
Ejemplos:   DEF s[80]:STRING, n   -> s es una cadenaE de longitud máx. 80
            ReadStr(stdout,s)     -> lee entrada desde la consola
            n:=Val(s)             -> recoge un número de la cadenaE
            -> etc.
```

Señalar que todas las funciones de cadenas manejan de forma correcta los casos en los que la cadena tiende a ser mayor que su longitud máxima.

```
            DEF s[5]:STRING
            StrAdd(s,'Esta cadena es mayor de 5 caracteres',ALL)
```

s sólo contendrá 'Esta '.

Las cadenas también pueden reservarse dinámicamente en la memoria del sistema con la función String(), (nota: siempre debe comprobarse si el puntero que retorna esta función es NIL).

```
s:=String(longmax)
```

```
          DEF s[80]:STRING   equivale a   DEF s    y   s:=String(80)
```

```
bool:=StrCmp(cadena,cadena,long=ALL)
```

Compara dos cadenas. 'long' debe ser el número de caracteres a comparar, o 'ALL' si queremos comparar la totalidad de la cadena. Retorna TRUE o FALSE ('long' es un argumento por omisión).

```
StrCopy(cadenaE,cadena,long=ALL)
```

Copia 'cadena' en 'cadenaE'. Si 'long'=ALL copiará toda la cadena. Retorna la cadenaE

```
StrAdd(cadenaE,cadena,long=ALL)
```

Igual que StrCopy(), sólo que ahora la cadena se concatena al final. Retorna la cadenaE.

```
long:=StrLen(cadena)
```

Calcula la longitud de cualquier cadena terminada en cero.

```
long:=EstrLen(cadenaE)
```

Retorna la longitud de una cadenaE.

```
max:=StrMax(cadenaE)
```

Retorna la longitud máxima de una cadenaE

```
StringF(cadenaE,cadenafmt,args,...)
```

Como WriteF(), pero ahora la salida va a cadenaE en vez de stdout. Retorna la cadenaE, y su longitud como segundo valor de retorno.
Ejemplo: StringF(s,'resultado: \d\n',123) -->'s' será 'resultado: 123\n'

```
   RightStr(cadenaE,cadenaE,n)
```

   Rellena la primera cadenaE con los últimos n caracteres de la segunda. Retorna la cadenaE.

```
   MidStr(cadenaE,cadena,pos,long=ALL)
```

   Copia 'long' caracteres (todos si long=ALL) desde la posición 'pos' de 'cadena' a 'cadenaE'. Retorna la cadenaE.

APUNTA BIEN: en todas las funciones relacionadas con cadenas en las que se usa una posición de cadena, el primer caracter de la cadena está en la posición 0, y no 1 como es común en algunos lenguajes como el BASIC.

```
   valor,leidos:=Val(cadena,leidos=NIL)
```

   Extrae el valor entero que representa la secuencia de caracteres ASCII de la cadena. No se tienen en cuenta los espacios/tabuladores,... que le precedan. De esta forma también se pueden leer números hexadecimales (1234567890ABCDEFabcdef) y binarios (01) si van precedidos de "$" o "%" respectivamente. Un menos "-" puede indicar un entero negativo. Val() también retorna el número de caracteres leidos como segundo argumento, el cual debe pasarse por referencia, o se puede recibir como segundo valor de retorno. Si 'leidos' es 0 ('valor' será 0 también) entonces la cadena no contenía un entero, o el valor era muy grande para ponerlo en 32 bits. 'leidos'puede ser NIL'

```
Ejemplos de cadenas que se analizan correctamente:
            '-12345', '%10101010', '   -$ABcd12'
Estos otros retornarán con "valor" y "leidos" igual a 0:
            '', 'hello!'
```

```
   posbusc:=InStr(cadena1,cadena2,posinicial=0)
```

   Busca en cadena1 la aparición de cadena2, probablemente partiendo de una posición diferente de 0. Retorna la posición en la que se halló la subcadena, o -1 en otro caso.

```
   nuevadircadena:=TrimStr(cadena)
```

   Retorna la *dirección* del primer caracter de la cadena que no sea un espacio, tabulador, etc...

```
   UpperStr(cadena)
   LowerStr(cadena)
```

   Cambia la cadena de mayusculas o a minúsculas. Retorna una cadena.

APUNTA BIEN: estas funciones cambian el contenido de la 'cadena', por lo que sólo pueden usarse con cadenasE, o cadenas que formen parte de los datos de tu programa. Esto quiere decir concretamente, que si obtienes la dirección de una cadena mediante la llamada a una función del sistema, deberás primero hacer un StrCopy() a una cadena de tu programa, para luego usar estas funciones.

```
   ok:=ReadStr(handlefich,cadenaE)
```

Lee una cadena (acabada en ASCII 10) de cualquier fichero o de stdout. 'ok' contendrá -1 si ocurrió un error, o se alcanzó el EOF.

Nota: el contenido de la cadena leida hasta ese momento sigue siendo válido. Señalar también que, al igual que Inp(), Out(), ... esta función utiliza un estilo de E/S sin buffers, y por lo tanto puede ser lenta. La función de la dos.library FGets() es una buena alternativa.

SetStr(cadenaE,nuevalong)

Cambia manualmente la longitud de la cadena. Sólo es útil cuando pones datos en una cadenaE mediane una función no específica de cadenasE y quieres seguir utilizandola como cadenaE. Por ejemplo, después de usar una función que simplemente pone una cadena terminada en cero en la dirección de la cadenaE, entonces puedes usar SetStr(micade,StrLen(micade)) para volverla manipulable de nuevo. Si la cadena es demasiado larga, SetStr() no hará nada (aunque eso debe evitarse siempre).

AstrCopy(cadena1,cadena2,tamaño)

'Array String Copy' copia cadena2 en el área de memoria denotado por cadena1. cadena1 no suele ser una cadenaE, sino un ARRAY. 'tamaño' es el número total de caracteres que puede contener cadena1, por ej., si es 5 y cadena2='holamundo', cadena1 tendrá entonces 'hola' + terminación en 0.

orden:=OstrCmp(cadena1,cadena2,max=ALL)

'Ordered String Compare' retorna 1 si cadena2>cadena1, 0 si son iguales y -1 si es menor. Sólo se camparan 'max' caracteres.

(Conviene mirar el apartado sobre Enlace de cadenas).

1.102 ch_9c

Listas y funciones de listas.

Las listas son como las cadenas con la única diferencia de que están formadas por LONGs en lugar de CHARs. También pueden reservarse de forma global, local o dinámica.

```
DEF milista[100]:LIST   -> local o global */
DEF a
a:=List(10)            -> dinámica.  En este caso, 'a' puede
                        -> ser NIL
```

De la misma forma que las cadenas se pueden representar con constantes en expresiones, las listas tienen equivalentes constantes:

```
[1,2,3,4]
```

el valor de tales expresiones es un puntero a una lista incializada. Una característica que tienen estas listas es que pueden tener partes dinámicas, es decir, partes que se completan en tiempo de ejecución:

```
a:=3
[1,2,a,4]
```

incluso pueden tener otro tipo que no sea el LONG por omisión, como:

```
[1,2,3]:INT
[65,66,67,0]:CHAR                    -> equivalente a 'ABC'
['topaz.font',8,0,0]:textattr
OpenScreenTagList(NIL,[SA_TITLE,'MiPantalla',TAG_DONE])
```

Como se ve en los ejemplos anteriores, las listas son extremadamente útilies con funciones del sistema: son compatibles con un ARRAY OF LONG, y las de objetos con tipos se pueden utilizar en cualquier lugar que necesite una estructura, o arreglo de ellas. Las funciones de Taglist y de número variable de argumentos se pueden utilizar de esta forma.
NOTA: todas las funciones de listas funcionan sólo con listas de LONG, las listas con tipos sólo son convenientes para la construcción de estructuras de datos complejas y expresiones.

Al igual que las cadenas, las listas mantienen cierta jeraquía:
listas var. -> listas const. -> arreglos de long/puntero a long
cuando una función necesita un arreglo de LONGs puedes usar tranquilamente una lista como argumento, pero si la función necesita una lista variable, o una lista constante, entonces no servirá un arreglo de LONGs.

Es importante entender el poder de las listas, y en particular las listas con tipo: éstas pueden evitar una buena cantidad de problemas en la construcción de prácticamente cualquier estructura de datos. Intenta usar las listas en tus propios programas, y observa la función que tienen en los programas de ejemplo.

Resumen:

```
[<elem>,<e>,... ]          lista inmediata (LONGs, con funcs. de listas)
[<elem>,<e>,... ]:<tipo>   lista con tipos (para crear estruct. de datos)
```

Si <tipo> en un tipo simple como INT o CHAR, lo que obtienes es un equivalente inicializado de un ARRAY OF <tipo>, si <tipo> es un nombre de objeto, estarás construyendo objetos inicializados, o ARRAY of <objeto>, dependiendo de la longitud de la lista.

Si escribes [1,2,3]:INT crearás una estructura de 6 bytes, de 3 valores de 16 bits para ser más preciso. El valor de esta expresión es, entonces, un puntero a ese área de memoria. Lo mismo ocurre si, por ejemplo, tienes un objeto como este:

```
OBJECT miobjeto
  a:LONG, b:CHAR, c:INT
ENDOBJECT
```

escribir [1,2,3]:miobjeto implicará la creación de una estructura de datos en memoria de 8 bytes, siendo los cuatro primeros bytes un LONG de valor 1, el siguiente byte un CHAR de valor 2, luego un byte relleno, y los dos últimos un INT (2bytes) de valor 3. También podrías escribir:

```
[1,2,3,4,5,6,7,8,9]:myobject
```

y estarías creando un ARRAY OF miobjeto de tamaño 3. Fíjate que tales listas no tienen porqué ser completas (con 3,6,9,... elementos), puedes crear objetos parciales con listas de cualquier tamaño.

Un último apunte sobre el tamaño de los datos: en el Amiga puedes depender del hecho de que una estructura como 'miobjeto' tiene un tamaño de 8 bytes, y que tiene un byte de relleno para obtener alineación de palabras (16 bit). Sin embargo, es bastante probable que un compilador de E para arquitecturas 80x86 no usaría el byte de relleno y crearía una

estructura de 7 bytes, y que un compilador de E para una arquitectura sun-sparc, si no me equivoco, intentaría alinear los datos a palabras dobles (32 bits), creando una estructura de 10 o 12 bytes. Algunos microprocesadores (son raros, pero existen) incluso usarían (36:18:9) como número de bits para sus tipos (LONG:INT:CHAR), en lugar de los (32:16:8) a los que estamos habituados. Con esto quiero decir que no hagas muchas suposiciones sobre la estructura de los OBJECTos y las LISTas si quieres escribir código que tenga alguna oportunidad de ser portable y que no dependa de efectos laterales.

```
   ListCopy(listavar,lista,num=ALL)
```

Copia num elementos de lista a listvar. Retorna listavar.

```
Ejemplo:    DEF milista[10]:LIST
            ListCopy(milista,[1,2,3,4,5],ALL)
```

```
   ListAdd(listavar,lista,num=ALL)
```

Copia num elementos de lista en la cola de listavar. Retorna listavar.

```
   ListCmp(lista,lista,num=ALL)
```

Compara dos listas, o una parte de ellas.

```
   long:=ListLen(lista)
```

Retorna la longitud de 'lista', ej: ListLen([a,b,c]) retornaría 3.

```
   max:=ListMax(listavar)
```

Retorna la longitud máxima posible de listavar.

```
   valor:=ListItem(lista,indice)
```

Opera como value:=list[index] con la diferencia de que la lista puede ser un valor constante en vez de un puntero. Esto es bastante útil en situaciones como la siguiente en la que queremos usar una lista de valores directamente:

```
            WriteF(ListItem(['¡vale!','¡sin mem!','¡no fichero!'],error))
```

que imprime un mensaje de error dependiendo de "error". Equivale a:

```
            DEF temp:PTR TO LONG
            temp:=['¡vale!','¡sin mem!','¡no fichero!']
            WriteF(temp[error])
```

```
   SetList(listavar,nuevalong)
```

Cambia la longitud de la lista de forma manual. Sólo será útil cuando leas información en la lista con una función que no sea específica para listas, y quieres seguir utilizandola como una lista de verdad.

(Mira las funciones de listas con expresiones entrecomilladas.)
(Tambien interesan la funciones de enlace de listas.)

1.103 ch_9d

Intuition.

```
ptrv:=OpenW(x,y,ancho,alto,IDCMP,wflags,titulo,patalla,sflags,
            gadgets,taglist=NIL)
```

Crea una ventana donde wflags son los flags de características de la ventana (como BACDROP, SIMPLEREFRESH,..., generalmente $F) y sflags son los que indican el tipo de pantalla en la que se abrirá la ventana (1=wb, 15=propia). Sólo es necesario que 'pantalla' sea un puntero válido si sflags=15, en otro caso sirve NIL. 'gadgets' puede apuntar a una estrutura glist que puedes crear fácilmente con la función Gadget(), en otro caso NIL.

```
CloseW(ptrv)
```

Cierra la ventana de nuevo. La única diferencia con CloseWindow() es que acepta punteros NIL, y que vuelve a poner stdrast a NIL.

```
ptrp:=OpenS(ancho,alto,planos,sflags,titulo,taglist=NIL)
```

Crea una pantalla propia. 'planos'=número de bitplanes (1-6,1-8 AGA).

```
CloseS(ptrp)
```

Igual que CloseW(), sólo que para pantallas.

```
buffersig:=Gadget(buffer,listag,id,flags,x,y,ancho,cadena)
```
[Atención: esta función está un poco anticuada]

Esta función crea una lista de gadgets, los cuales se pueden luego incluir en tú ventana pasándola como argumento a OpenW(), o más tarde con funciones de Intuition como AddGlist().
'buffer' es por lo general un ARRAY de al menos GADGETSIZE bytes para mantener todas las estructuras asociadas con un gadget, 'id' es número identificador que nos ayudará a recordar el gadget pulsado cuando llegue un IntuiMessage. 'flags' puede ser: 0=gadget normal, 1=gadget booleano, 3=gadget booleano seleccionado. 'ancho' es el ancho en pixels, y debe ser lo suficientemente grande como para acoger 'cadena', la cual estará centrada automáticamente. 'listag' debe ser NIL para el primer gadget, y varlistag para todos los demás, para que E pueda enlazar los gadgets.
La función retorna un puntero al siguente buffer (=buffer+GADGETSIZE). Ejemplo para tres gadgets:

```
CONST MAXGADGETS=GADGETSIZE*3
DEF   buf[MAXGADGETS]:ARRAY, sigui, ptrv

sigui:=Gadget(buf,NIL,1,0,10,20,80,'bla')   -> el 1er gadget
sigui:=Gadget(sigui,buf,... )
sigui:=Gadget(sigui,buf,... )               -> enlaza gadgets
ptrv:=OpenW( ...,buf)
```

```
codigo:=Mouse()
```

Retorna el estado actual de los 2 o 3 botones del ratón; izqdo=1, dcho=2 y central=4. Si, por ejemplo, codigo=3 querrá decir que se pulsaron los botones izquierdo y derecho a la vez.

NOTA: Esta no es una función real de Intuition, si quieres conocer los eventos del ratón de una forma correcta, tendrás que analizar los IntuiMessage que reciba tu ventana. Esta es la única función de E que utiliza el hardware directamente, y por tanto sólo es aconsejable para programas de tipo demo, o para pruebas. (NO USES ESTA FUNCION EN PROGRAMAS QUE SE SUPONE VAN A FUNCIONAR BAJO EL SO).

```
bool:=LeftMouse(ven)
WaitLeftMouse(ven)
```

Alternativa de Intuition a Mouse(), comprueban la pulsación del ratón.

```
x:=MouseX(ven)
y:=MouseY(ven)
```

Permite leer la posición del ratón. Coordenadas relativas a 'ven'.

```
clase:=WaitIMessage(ventana)
```

Esta función hace más fácil la espera de un evento de ventana. Espera la llegada de un IntuiMessage y retorna la clase del evento. Guarda otros datos como 'codigo' y 'calificadores' en variables globales privadas, que pueden leerse con las siguientes funciones. WaitIMessage() equivale al siguiente fragmento de código:

```
PROC waitimessage( ven:PTR TO window )
DEF puerto,mens:PTR TO intuimessage,clase,codigo,calif,diri
puerto:=ven.userport
IF (mens:=GetMsg(puerto)) = NIL
  REPEAT
    WaitPort(puerto)
  UNTIL (mens:=GetMsg(puerto)) <> NIL
ENDIF
clase:=mens.class
codigo:=mens.code             /* almacenado internamente */
calif:=mens.qualifier
diri:=mens.iaddress
ReplyMsg(mens)
ENDPROC clase
```

como puedes ver, recoge exactamente un mensage, y no se olvida de múltiples mensajes que llegan en el mismo evento, si se llama más de una vez. Digamos, por ejemplo, que has abierto una ventana que muestra algo y simplemente espera a que la cierres (sólo indicaste IDCMP_CLOSEWINDOW):

```
WaitIMessage(miventana)
```

o puedes tener un programa que espere por más tipos de eventos, que controlarías en un bucle, y finaliza con un evento CLOSEWINDOW:

```
WHILE (clase:=WaitIMessage(ven))<>IDCMP_CLOSEWINDOW
   /* trata otras clases */
ENDWHILE
```

```
codigo:=MsgCode()
calif:=MsgQualifier()
diri:=MsgIaddr()
```

Proporcionan las variables globales privadas mencionadas antes. Los valores que retornan son los definidos durante la llamada a WaitIMessage() más reciente.

Ejemplo:

```
IF clase:=IDCMP_GADGETUP
```

```
        migadget:=MsgIaddr()
        IF migadget.userdata=1 THEN  -> ... usuario pulsó gadget #1
      ENDIF
```

1.104 ch_9e

Gráficos.

Todas las funciones gráficas de apoyo que requieren explícitamente un rastport utilizan la variable de sistema 'stdrast', definida automáticamente en la última llamada a OpenW() o OpenS(), y que toma el valor NIL con CloseW() y CloseS(). Se puede llamar a estas dos últimas funciones cuando 'stdrast' aún es NIL. Se le puede dar un valor a 'stdrast' manualmente con SetStdRast() o 'stdrast:=mirastport'.

Plot(x,y,color=1)

Dibuja un sólo punto en tu pantalla/ventana con uno de los colores disponibles. El número del color va desde 0 a 31, o 0 a 255 bajo AGA.

Line(x1,y1,x2,y2,color=1)

Dibuja una línea.

Box(x1,y1,x2,y2,color=1)

Dibuja un rectángulo o caja.

Colour(dibujo,fondo=0)

Fija los colores para todas las funciones gráficas (de la librería) que necesiten un color como argumento. Este es el *registro* del color (por ejemplo, de 0 a 31) y no el *valor* del color.
NOTA: las funciones que necesitan un "color" cambian el Apen de stdrast.

TextF(x,y,cadenaformato,args,...)

Exactamente igual que WriteF(), sólo que escribe en algún (x,y) de tu stdrast, en lugar de hacerlo a stdout. Retorna la longitud de la cadena resultante.

rastant:=SetStdRast(rastnuevo)

Cambia el rastpor de salida de las funciones gráficas de E.

SetTopaz(tamaño=8)

Te permite fijar la fuente del rastport "stdrast" a topaz, simplemente para asegurarte de que alguna fuente propia no desfigure la composición de la ventana. Por supuesto, 'tamaño' puede ser 8 o 9. Está para usarse como último recurso si no permites sensibilidad a fuentes.

SetColour(pantalla,regcolor,r,g,b)

Fija el registro de color (0..255) de la pantalla a cierto valor RGB. Cada valor de RGB puede oscilar entre 0..255, es decir, color de 24 bits. Esta función reescalará el color de forma automática si no se dispone de un modo AGA, utilizando la función apropiada.

1.105 ch_9f

Sistema.
---

bool:=KickVersion(vers)

Retornará TRUE si el KickStart de la máquina en que está corriendo tu programa es igual o superior a 'vers', en otro caso FALSE.

mem:=New(n)

Esto crea dinámicamente un arreglo (área de memoria, si lo prefieres) de n bytes. Se diferencia de AllocMem() en que se llama de forma automática con flags $10000 (es decir, memoria inicializada a zero de cualquier tipo), y en que no es necesario llamar a Dispose(), ya que está enlazada a una lista de memoria liberada automáticamente al finalizar el programa.

mem:=NewR(n)

Igual que New(), solo que ahora alcanza la excepción "MEM" de forma automática en lugar de retornar cuando no hay suficiente memoria.

mem:=NewM(n,flags)

Igual que NewR(), y además permite especificar flags (MEMF_CHIP etc.)

Dispose(mem)

Libera memoria reservada con New(). Sólo hace falta usar esta función cuando quieres liberar memoria de forma explícita _durante_ el programa, ya que, de todas formas, al final del programa se libera toda la memoria.

CleanUp(valorret=0)

Da por finalizado el programa en cualquier punto. Sustituye al Exit() de DOS: !no uses Exit() nunca!, utiliza CleanUp() en su lugar, ya que libera la memoria, cierra las librerias correctamente, ... 'valorret' se da al DOS como código de retorno. Sólo se necesita CleanUp() cuando hay que finalizar el programa en un punto diferente al ENDPROC de main().

cantidad:=FreeStack()

Retorna la cantidad de espacio libre que hay en la pila. Que debe ser siempre 1000 o superior. No debes preocuparte de tal cantidad mientras no utilice recursiones profundas.

bool:=CtrlC()

Retorna TRUE si se ha pulsado Ctrl-C desde la última comprobación, en

```
otro caso  FALSE.  Sólo funciona en programas  que se están ejecutando en
una consola, es decir, programas de CLI.

Ejemplo de uso de las tres últimas funciones:
         -> calcular el factorial de un argumento de la línea de comandos
         OPT STACK=100000

         PROC main()
           DEF num, r
           num:=Val(arg,{r})
           IF r=0
             WriteF('args erróneos.\n')
           ELSE
             WriteF('resultado: \d\n',fac(num))
           ENDIF
         ENDPROC

         PROC fac(n)
           DEF r
           IF FreeStack()<1000 OR CtrlC() THEN CleanUp(5) -> test extra
           IF n=1 THEN r:=1 ELSE r:=fac(n-1)*n
         ENDPROC r
por  supuesto,  es  difícil que esta recursión acabe con el espacio de la
pila,  y si lo hace, el programa se detiene con FreStack() tan rápido que
no  te  dará  tiempo a pulsar Ctrl-C, pero aquí lo que cuenta es la idea.
Una definición de fac(n) menos segura sería:
         PROC fac(n) IS IF n=1 THEN 1 ELSE fac(n-1)*n

   mem:=FastNew(tamaño)
   FastDispose(mem,tamaño)
   FastDisposeList(lista)

   FastNew() y FastDispose() substituyen a NewR(size) y Dispose(ptr) (NEW
y END los utilizan).  Esto es lo que tienen en común:
      - pueden lanzar excepciones "NEW".
      - la memoria siempre se inicializa a cero.
      - liberación automática al final del programa.
aunque deben señalarse las siguientes diferencias positivas :
      - son entre 10 y 50 veces más rápidas (!)
      - usan algo menos de memoria para objetos pequeños.
      - no fragmentan la memoria.
      [con objetos <=256 bytes, con mayores se usa NewR() y Dispose() ].
y negativas :
      - no liberan la memoria, la reciclan.
      - FastDispose() necesita tamaño exacto de la reserva. END también.

   Para liberar listas reservadas con NEW se necesita utilizar la función
FastDisposeList().  Dado que las listas tienen longitud, no hace falta el
parámetro de tamaño.
```

1.106 ch_9g

```
Matemáticas y otras.
--------------------
```

```
a:=And(b,c)
a:=Or(b,c)
a:=Not(b)
a:=Eor(b,c)
```

Estas funcionan con las operaciones usuales, tanto booleanas como aritméticas. Señalar que existen operadores para And() y Or().

```
a:=Mul(b,c)
a:=Div(a,b)
```

Realizan la misma operación que los operadores '*' y '/', aunque con 32 bits completos. Por razones de velocidad, las operaciones normales son 16bit*16bit=32bit y 32bit/16bit=16bit. Suficiente para prácticamente todos los cálculos, y si no lo es, puedes usar Mul() y Div().
NOTA: en el caso de Div(), a se divide entre b, y no b entre a.

```
bool:=Odd(x)
bool:=Even(x)
```

Retorna TRUE o FALSE si la expresión es impar (Odd()) o par (Even()).

```
Min(a,b)
Max(a,b)
```

Computan el mínimo y máximo de dos enteros.

```
numaleat:=Rnd(max)
semilla:=RndQ(semilla)
```

Rnd() computa un número aleatorio a partir de una semilla interna, oscilando entre 0..max-1. Por ejemplo, Rnd(1000) retorna un entero entre 0..999. Para inicializar la semilla interna, llama a Rnd() con un valor negativo; el Abs() de ese valor será la semilla inicial.
RndQ() computa un número aleatorio más rápido que Rnd(), aunque retorna números en todo el rango de los 32 bits. Usa el resultado como semilla para la siguiente llamada, y como semilla inicial utiliza cualquier valor grande, como $A6F87EC1.

```
valorabs:=Abs(valor)
```

Computa el valor absoluto.

```
s:=Sign(v)
```

Computa el signo de v, es decir, retorna -1, 0 o 1.

```
a,b:=Mod(c,d)
```

Divide c (32bit) entre d (16bit) y retorna el módulo a (16bit) y opcionalmente el resultado b (16bit) de la división. Mod() retornará resultados erróneos si se excede el límite de los 16 bits.

```
x:=Shl(y,num)
x:=Shr(y,num)
```

Desplaza y num bits a la izquierda o a la derecha (bits nuevos a 0).

```
   a:=Long(dir)
   a:=Int(dir)
   a:=Char(dir)
```

Mira en una dirección determinada de la memoria y retorna el valor encontrado. Funcionan con valores de 32, 16 y 8 bits respectivamente. Señalar que el compilador *no* comprueba si 'dir' es una dirección válida. Estas funciones están disponibles para que se utilicen sólo en los casos en los que la lectura y escritura de memoria mediante PTR TO <type> haría el programa más complejo o menos eficiente. Es _desaconsejable_ el uso de estas funciones.

```
   PutLong(dir,a)
   PutInt(dir,a)
   PutChar(dir,a)
```

Escribe el valor 'a' en memoria. Mira Long(), Int() y Char().

```
   y:=Bounds(x,a,b)
```

Asegura que x se encuentre entre a y b, y lo ajusta de forma acorde si es necesario. Equivale a 'y:=IF x<a THEN a ELSE IF x>b THEN b ELSE x'.

1.107 ch_9h

```
Enlace de cadenas y listas.
---------------------------
```

E proporciona un conjunto de funciones que permite la creación de listas enlazadas con los tipos de datos STRING y LIST, o cadenas y listas creadas con String() y List() respectivamente. Como ya debes saber, las cadenas y las listas, tipos de datos complejos, son punteros a sus respectivos datos, y tienen campos extra en offsets negativos de ese puntero, que indican su longitud y longitud máxima actuales. Los offsets de los campos son privados (PRIVATE). Todos los tipos de datos complejos tienen, además de esos dos, un campo 'next' (siguiente), inicializado a NIL por omisión, y que se puede usar para construir listas o cadenas enlazadas, por ejemplo. Desde ahora, usaré 'complejo' para denotar un puntero a STRING o LIST, y 'cola' para denotar otro puntero de ese tipo, o que ya tenga otras cadenas enlazadas a él. 'cola' también puede ser un puntero NIL, indicando el final de una lista enlazada.
[Nota: estas funciones de cadenas/listas no tienen nada que ver con las listas-E o las listas Celdas-Lisp]
Se pueden usar las siguiente funciones:

```
   complejo:=Link(complejo,cola)
```

Pone 'cola' en el 'next' de 'complejo'. Retorna 'complejo' de nuevo.

```
Ejemplo:    DEF s[10]:STRING, t[10]:STRING
            Link(s,t)
```

creará una lista enlazada como: s --> t --> NIL

```
   cola:=Next(complejo)
```

Lee el campo 'next' de 'complejo'. Puede ser NIL, o una lista enlzada completa. Next(NIL) resultará en NIL, luego es seguro llamar a Next() cuando no estás seguro si estas al final de la lista enlazada o no.

cola:=Forward(complejo,num)

Como Next(), sólo que se adelanta 'num' enlaces en lugar de uno, es decir 'Forward(c,1)=Next(c)'. Puedes llamar a Forward() de forma segura con un número grande, Forward() se detendrá si encuentra NIL mientras recorre los enlaces, y retornará NIL en ese caso.

DisposeLink(complejo)

Como Dispose(), con dos diferencias: sólo sirve para cadenas y listas reservadas con String() o List(), además liberará el resto de 'complejo' automáticamente. Señalar también que se puede usar con listas enlazadas grandes que contengan tanto cadenas reservadas con String() como algunas reservadas localmente o globalmente con STRING.

Mira Src/Utils/D.e para ver un buen ejemplo de como se puede hacer buen uso de listas o cadenas enlazadas en programas reales.

1.108 ch_9i

Celdas-Lisp y funciones de celdas.

Vaya. Correcto. Pensabas que el LISP era divertido, entonces prueba E. [o: la historia de por qué E es un LISP mejor que el propio LISP :-)]

A partir de la versión v3, E tiene el tipo de datos celda, casi idéntico a las celdas del lenguaje LISP. Más técnicamente, E tiene:

'Celdas-Lisp recogidas de residuos con marcas conservadoras e intercambio'
'Conservative Mark and Sweep Garbage-Collected Lisp-Cells' [!¿#!]

Básicamente, implica el ser capaz de reservar celdas-LISP, las cuales son pares de valores:

x:=<a|b>

Lo cual se parece mucho a NEW [a,b]:LONG, sólo que ahora E liberará los 8 bytes en cuestión por sí sólo cuando se encuentre que necesita memoria, y no haya punteros apuntando a la celda. En la práctica esto significa que puedes tener funciones que crean celdas temporales sin preocuparse de liberarlas. Y, cualquier programador de LISP sería capaz de explicarte que con celdas puedes construir cualquier tipo de estrutura de datos (más notablemente árboles y listas).
[nota: este texto no explica detenidamente como aprovechar las celdas por completo, ya que se han escrito docenas de libros a ese respecto.]

La selección de los valores se puede realizar usando Car(x) y Cdr(x), dos funciones de LISP que seleccionan los elementos cabeza y cola (primero, segundo) de la celda. Si x es un PTR TO LONG, incluso se puede utilizar x[0] y x[1].

También se pueden escribir listas de celdas (atento a las comas):

```
<a,b,c>    como abreviatura de    <a|<b|<c|NIL>>>
```

La unificación de E sirve de alternativa a la selección Car()/Cdr():

```
x <=> <a,b|c>
a+b+c
```

en vez de:

```
Car(x)+Car(Cdr(x))+Cdr(Cdr(x))
```

La unificación de celdas-LISP se parece a la unificación con listas-E. Por ejemplo :

```
x <=> <1,2|a>
```

equivale a:

```
IF Car(x)=1
  IF Car(Cdr(x))=2
    a:=Cdr(Cdr(x))
    ...
```

También existe el valor nil de LISP "<>", que equivale a NIL y 0.

Veamos algunas funciones disponibles (señalar que Cons() _solo_ se puede usar por medio de <...>)

```
h:=Car(c)       t:=Cdr(c)
```

Fija el valor de la cabeza y la cola de la celda c.

```
bool:=Cell(c)
```

Determina si 'c' apunta o no a una celda, de forma que Cell(<1>)=TRUE, y Cell(3.14)=FALSE. No es una función rápida.

```
n:=FreeCells()
```

Determina el número de celdas libres disponibles. Es muy lenta. No hay necesidad alguna de llamar a esta función, si no es por curiosidad.

```
SetChunkSize(k)
```

Fija el tamaño de los bloques reservados para celdas en k kilobytes. Por omisión es 128k. Esta función sólo se puede llamar una vez, y sólo antes de que tenga lugar la primera reserva constante (<..>). A partir de ese momento esta función no tiene ningún efecto.

Resumiendo, hazte con un buen libro sobre LISP para entender bien la programación con celdas.

Se pueden escribir funciones de LIST en E, con exactamente la misma funcionalidad:

```
PROC append(x,y) IS IF x THEN <Car(x)|append(Cdr(x),y)> ELSE y
PROC nrev(x) IS IF x THEN append(nrev(Cdr(x)),<Car(x)>) ELSE NIL
PROC sum(x) IS IF x THEN Car(x)+sum(Cdr(x)) ELSE 0
```

También se permite el uso de implementación destructiva para funciones como estas.

Notas técnicas:
El recolector de residuos (RR) de E implementa un algoritmo de marca e intercambio conservativo que se ha comprobado que es entre 5 y 25 veces más rápido que las distintas implementaciones lógicas y funcionales del Amiga. Conservativo indica que en caso de duda el RR no libera la celda. Esto es necesario debido a que en un lenguaje sin tipos como el E, el RR podría llegar facilmente a un valor que no es un puntero válido, etc.

El RR reserva bloques grandes (por omisión 128k), en los cuales guarda las celdas. Si se acaban las celdas, recolectará los residuos analizando la pila y los registros en busca de punteros al área de celdas, y los marcará de forma recursiva. Tras eso, todas las celdas que no hayan sido marcadas se volverán a utilizar, y si la ganancia tras la recolección fue pequeña, se reservará un nuevo bloque (si eso falla se alcanzará una excepción "NEW").

Interacción con otros valores de E:
- No representa ningún problema almacenar otros valores en las celdas. Se pueden poner objetos, cadenas, reales, cualquier cosa en celdas sin confundir demasiado al RR.
- Sí es problemático el almacenar celdas en otros valores, por ejemplo un puntero a una celda en un objeto dinámico, ya que el RR no será capaz de encontrarlo allí. Sin embargo, pienso que ese caso aparecerá raras veces. Los punteros a celdas se pueden guardar de forma segura en variables globales o locales, incluso registros, y en cualquier estructura de datos de la pila.
  [¡y más importante, en otras celdas!]

Intrínsecos:
- El RR no puede, de momento, recolectar celdas cuya lista-Car tenga una longitud >1000 o similar, por ejemplo <<<NIL:a>:b>:c>, pero con 1000 entradas en vez de 3. Esto es muy dificil que ocurra ya que las listas de este tipo se suelen formar con listas-Cdr, que el RR puede controlar hasta de tamaño infinito. (alcanzará una "STCK" si eso falla).
- El código ensamblador en línea no debe poner cosas en la pila que no estén alineadas a 32 bits. Esto ya era necesario en v2.1b, pero ahora es más imprescindible.

Hay una relación entre velocidad y espacio con respecto al tamaño de los bloques. La reserva de bloques pequeños es, obviamente, interesante ya que no desperdicia memoria, sin embargo, al recolectar residuos, el trabajo de control de los punteros es proporcional al número de espacios. Por tanto:
- Si es más importante la velocidad, ajusta el tamaño de los bloques de forma que sólo se necesite un bloque. Si el uso máximo de memoria de celdas en cualquier momento es de 50k, un espacio de bloque de 100k o 150k ofrecerá un rendimiento óptimo.
- Si lo más importante es la memoria, en el ejemplo anterior un tamaño de bloque de 20k o 30k será bastante óptimo en cuanto a uso de memoria. En resumen, temporiza el uso del algoritmo de celdas en situaciones complicadas para ver la relación velocidad/memoria más apropiada.

1.109 ch_10

Funciones de biblioteca y módulos.

Llamadas a funciones predefinidas
Interface con el sistema Amiga con los módulos v39
Compilación de módulos propios
Caché de módulos

1.110 ch_10a

Llamadas a funciones predefinidas.

Como habrás notado en secciones anteriores, el fragmento de código enlazado de forma automática antes de tu código, llamado "código de inicialización", siempre abre las tres librerías intuition, dos y graphics (y algunas veces mathieeesigbas), y debido a esto, el compilador tiene todas las llamadas a estas cuatro librerías (incluyendo exec) integradas en el compilador (hay unos cientos de ellas). Estas son hasta AmigaDos 3.00 (v39). Para llamar al Open() de 'dos' sólo tienes que escribir:

```
handle:=Open('mifichero',OLDFILE)
```

o para AddDisplayInfo() de la librería graphics:

```
AddDisplayInfo(midispinfo)
```

Es tan sencillo como eso.

[nota: a lo que me refiero con librería es en realidad una 'biblioteca', es decir, una biblioteca o colección de funciones. Lo que pasa es que debido a que en inglés se les llaman libraries (como debe ser), parace ser que a uno se le pega más rápido lo de librería ...:-(]

1.111 ch_10b

Interface con el sistema Amiga con los módulos de v39.

Para usar cualquier otra librería, a parte de las cinco de la sección anterior, debes recurrir a los módulos. Igualmente si deseas utilizar algúna definición de OBJECTo o CONST de los includes del Amiga, como es normal en C o ensamblador, también necesitarás los módulos. Los módulos son ficheros binarios que pueden incluir definiciones de constantes, objetos librerías y funciones (código). El hecho de que sean binarios tiene la ventaja sobre el ascii (como en C y en ensamblador), que no es necesario compilarlos una y otra vez, cada vez que se compila un programa. La desventaja es que no se puede leer con tanta facilidad, se necesita una utilidad como ShowModule para ver su contenido. Los módulos que contienen las definiciones de librerías se encuentran en el directorio raiz de EMODULES: (el directorio modules/ en la distribución), las definiciones de constantes/objetos están en los subdirectorios,

estructurados de la misma forma que los originales de Commodore.

MODULE

Sintaxis: MODULE <nombremódulo>,...

Lee un módulo. Un módulo es un fichero binario con información sobre librerías, constantes, y a veces, funciones. El uso de módulos permite usar librerías y funciones que eran desconocidas por el compilador.

Veamos a ahora un ejemplo, una versión reducida del ejemplo source/Examples/asldemo.e, que usa módulos para obtener un requester de ficheros de la Asl.library 2.0.

```
MODULE 'Asl', 'libraries/Asl'

PROC main()
  DEF req:PTR TO filerequester
  IF aslbase:=OpenLibrary('asl.library',37)
    IF req:=AllocFileRequest()
      IF RequestFile(req)
        WriteF('Fichero: "\s" in "\s"\n',req.file,req.drawer)
      ENDIF
      FreeFileRequest(req)
    ENDIF
    CloseLibrary(aslbase)
  ENDIF
ENDPROC
```

el compilador obtiene del módulo 'asl' las definiciones de las funciones de asl como RequestFile(), y la variable global aslbase, que sólo necesita que el programador la inicialize. De 'libraries/Asl' obtiene la definición del objeto 'filerequester', el cual se usa para leer el fichero que seleccionó el ususario.

Bueno, ¿Pensabas que era más difícil tener un requester de ficheros en E?

1.112 ch_10c

Compilando módulos propios.

A partir de la versión v3 puedes agrupar en un sólo fuente todos los PROCs, CONSTs, OBJECTos, y hasta cierto punto también las variables globales, que te parezca esten relacionados entre sí de alguna forma. Utiliza "OPT MODULE" para indicar al compilador EC que ese fichero se supone un módulo, y luego compílalo a un fichero .m que se podrá usar en el programa principal, igual que con los demás módulos.

Por omisión todos los elementos de un módulo son PRIVATE, es decir, no son accesibles al código que importa el fichero .m. Para hacer visibles los elementos que quieras, simplemente escribe EXPORT antes de ellos:

```
EXPORT ENUM TESTING,ONE,TWO,THREE,FOUR
EXPORT DEF var_global_importante, bla:PTR TO x
EXPORT OBJECT x
  sig, indice, termino
ENDOBJECT
```

```
EXPORT PROC burp()
 /* lo que sea */
ENDPROC
```

"EXPORT" es útil para hacer la distinción entre privado y público, especialmente, cuando se puede acceder a todas las funciones de un objeto mediante PROCs, puedes desear mantener un OBJECTo privado como una forma efectiva de ocultamiento de datos.
[EXPORT puede aparecer en cualquier línea, en ese caso no afectará nada.]

Si es necesario exportar _todos_ los elementos de un módulo (por ejemplo, uno que sólo tenga constantes), con 'OPT EXPORT' se exportrá todo, sin que haga falta utilizar EXPORTs individuales.

Las variables globales necesitan una atención especial:
- Intenta eludir muchas variables globales. El tener muchas variables globales en los módulos hace los proyectos propensos a errores.
- Las variables globales de un módulo no pueden tener inicializaciones directas en la sentencia DEF (la razón de esto se aclará más tarde). Por ejemplo: DEF a y no DEF a=1
  DEF a:PTR TO x y no DEF a[10]:ARRAY OF x
- Las variable globales de un módulo que no se exportan operan como locales al módulo, es decir, nunca chocarán con globales de otros módulos. Aquellas exportadas, se combinan con las demás, es decir, si se usa una variable con el mismo nombre, tanto en el programa principal como en módulos, ésta será una sóla y la misma para los dos. Este es el motivo de que se pueda escribir DEF a[10]:ARRAY OF x en el programa principal, y EXPORT DEF a:PTR to x en el módulo, para compartir el arreglo. Además, si ambos usan, por ejemplo, 'gadtools.m', sólo uno de los dos tiene que inicializar 'gadtoolsbase' para que ambos puedan hacer llamadas a la librería. Si no quieres compartir las bases de las librerías (es decir, quieres tener una base de librería local,privada), simplemente redeclárala en un DEF de un módulo que no la EXPORTe. Si exportas una variable en un módulo de proprosito general, asegúrate de que tenga un nombre único.
- El uso de variables en módulos que proporcionan tipos de datos de propósito general necesita de una atención especial, ya que el módulo puede ser usado desde más de un módulo, en cuyo caso no está claro quien es el responsable de los recursos. Ten esto bien en cuenta.

Uso de módulos dentro de módulos

Esto requiere una atención extra. Si el módulo (B) que incluyes en tu propio módulo (A) es uno que sólo declara CONSTs, LIBRARYs y OBJECTos (sin código) no sucede nada especial, sin embargo, si B incluye PROCs, entonces es obvio que ese código necesita ser enlazado más tarde con el programa principal cuando se enlace A. Por ello, si un programa principal usa A, B debe estar presente en la compilación. El hecho de que A necesite a B se guarda en A, y se puede ver con ShowModule. Esta cadena de usos puede crecer creando un árbol de dependencias, que tiene como resultado que aunque sólo utilices un módulo en tu programa, se enlazarán a él de forma automática unos cuantos más. Por ello, el sistema de módulos de E mantiene de forma automática las dependencias para las que otros lenguajes necesitan makefiles. EC también permite dependencias circulares, y lee/enlaza un módulo a lo sumo una vez (es decir, no enlaza módulos que no se usaron). Una cosa que el sistema de módulos de E no realiza de forma automática es recompilar módulos dependientes. Si cambias B, a menudo será necesario recompilar A también, ya que puede

hacer referencia de offsets, etc, de la versión antigua de B, lo cual puede provocar que el código rompa. Si esto se empieza a complicar en tu proyecto, puedes usar una utilidad como E-Build.

Prueba el nuevo ShowModule para ver lo que pone EC en los módulos.

Inclusión de módulos de otros directorios.
Por omisión, al nombre del módulo se le pone como prefijo 'EMODULES:' para obtener el fichero en concreto. Ahora puedes anteponer un '*' al módulo para indicar el directorio en el que se encuentra el código fuente, de forma que si:

```
MODULE 'bla', '*bla'
```

estuviera en el fuente 'WORK:E/burp.e', el compilador buscará los módulos 'EModules:bla.m' y 'WORK:E/bla.m'.

Esta es, naturalmente, la forma de incluir componentes de tu aplicación en otras partes. Si escribes módulos que usas en muchos de tus programas, sería bastante conveniente guardarlos dentro de la jerarquía de 'EModules:', y el lugar adecuado para esto es el directorio 'EModules:other/'.

1.113 ch_10d

El caché de módulos.

(mira ShowCache/FlushCache, sobre esto).

1.114 ch_11

Expresiones entrecomilladas.

Entrecomillado y alcance
Eval()
Funciones predefinidas

1.115 ch_11a

Entrecomillado y alcance.

Las expresiones entrecomilladas empiezan con una comilla inversa "`". Su valor NO es el resultado de la computación de la expresión, sino la dirección del código que la computa. Este resultado se puede utilizar como una variable, o como argumento de ciertas funciones.
Ejemplo: mifunc:=`x*x*x
hace que ahora `mifunc' sea un puntero a una `función' que computa x^3 cuando se evalúa. Estos puntero a funciones son bastante diferentes de los PROCs normales, y nunca se deben mezclar o confundir. La principal

diferencia de las expresiones entrecomilladas es que sólo son expresiones simples, y por tanto no pueden tener sus propias variables locales. En nuestro ejemplo, 'x' es o una variable global o una variable local. Ahí es donde debemos tener precaución: si evaluamos 'mifunc' en algún otro lugar del mismo PROC, x puede ser local, pero si 'mifunc' se pasa como parámetro a otro PROC, y luego se evalúa, x debe ser entonces global. El compilador no realiza esta comprobación.

1.116 ch_11b

```
Eval().
-------

   Eval(func)

   Simplemente evalúa la expresión entrecomillada (exp = Eval('exp)).
```

NOTA: Debido a que E es, de alguna forma, un lenguaje sin tipos, el escribir "Eval(x*x)" de forma accidental en lugar de "Eval('x*x)" pasará desapercibido al compilador, y propiciará muchos problemas en tiempo de ejecución: el valor del x*x se usará como un puntero a código.

Para entender el motivo por el que las 'expresiones entrecomilladas' representan una característica importante piensa en los siguientes casos: si quisieras realizar una serie de acciones en un grupo de variables diferentes, normalmente escribirías una función y la llamarías con diferentes argumentos. Pero, ¿qué ocurre cuando el elemento que quieres dar como argumento es un fragmento de código? En lenguajes tradicionales esto no sería posible, por lo que necesitarías 'copiar' los bloques de código que representan la función, y poner la expresión en ellos. En E no. Digamos que quieres escribir un programa que mida el tiempo de ejecución de diferentes expresiones. En E simplemente escribirías:

```
            PROC mide(func,titulo)
              -> realizar todo tipo de inicializaciones del tiempo
              Eval(func)
              -> y el resto
              WriteF('\s tardó \d en ejecutarse\n',titulo,t)
            ENDPROC
```

y luego lo llamas con:

```
            mide('x*x*x,'multiplicación')
            mide('calcenorme(),'cálculo grande')
```

En cualquier otro lenguaje imperativo, necesitarías escribir copias de mide() para cada llamada, o poner cada expresión en una función diferente. Este es un ejemplo sencillo, piensa en lo que podrías hacer con estructuras de datos (LISTas) y código sin evaluar:

```
            funcsdibujar:=[ 'Plot(x,y,c), 'Line(x,y,x+10,y+10,c),
                            'Box(x,y,x+20,y+20,c) ]
```

Señalar que esta idea de funciones como variables normales/valores no es nuevo de E, las expresiones entrecomilladas viene literalmente de LISP, que además también tiene las llamadas funciones Lambda, de alguna forma más potentes, y que también se pueden dar como argumentos a funciones. Las expresiones entrecomilladas de E se pueden ver como

lambdas sin parámetros (o sólo de parámetro global).

1.117 ch_11c

```
Funciones predefinidas.
-----------------------
```

MapList(dirvar,lista,listavar,func)

Aplica una función a todos los elementos de la lista y retorna los resultados en 'listavar'. 'func' debe ser una expresión entrecomillada, y 'var' (que tomará todos los valores de la lista) se dará por referencia. Retorna listavar.
Ejemplo: MapList({x},[1,2,3,4,5],r,'x*x) resulta con r: [1,4,9,16,25]

ForAll(dirvar,lista,func)

Retorna TRUE si la función (expresión entrecomillada) se evalúa a TRUE para todos los elementos de la lista, y FALSE en otro caso. También se usa para aplicar una cierta función a todos los elementos de la lista:

```
ForAll({x},['uno','dos','tres'],'WriteF('ejemplo: \s\n',x))
```

Exists(dirvar,lista,func)

Como ForAll(), sólo que ésta retorna TRUE si para algún elemento la función se evalúa a TRUE (<>0). ForAll() siempre evalúa todos los elementos, pero Exists() probablemente no.

SelectList(v,lista,listavar,exprentrecom)

Muy parecida a MapList(), sólo que ahora no guarda el resultado de 'exprentrecom', lo usa como valor booleano, y sólo los valores para los cuales es cierta se guardan en 'listavar' --que debe poder guardar el mismo número de elementos que 'lista'. Retorna la longitud de listavar.
Ejemplo: SelectList({x},[1,2,0,3,NIl],r,'x<>0) resulta con r=[1,2,3]

Ejemplo práctico de utilización:
Reservamos diferentes tamaños de memoria en una sentencia, los comprobamos a la vez, y los liberamos todos de una vez, pero sólo los que se pudieron reservar. (Este ejemplo necesita v37+)

```
PROC main()
  DEF mem[4]:LIST,x
  MapList({x},[200,80,10,2500],mem,'AllocVec(x,0))          -> reservamos
  WriteF(IF ForAll({x},mem,'x) THEN 'Yes!\n' ELSE 'No!\n')    -> ¿éxito ?
  ForAll({x},mem,'IF x THEN FreeVec(x) ELSE NIL)    -> liberar sólo <>NIL
ENDPROC
```

Fíjate en la ausencia de iteración en este código. Intenta escribir este ejemplo en cualquier otro lenguaje para ver porqué es especial.

1.118 ch_12

Reales.

En E los REALes (o FLOATs, como quieras) son bastante diferentes de los de otros lenguajes. Esto es debido, sobre todo, al hecho de que E realmente no hace discriminación entre tipos de valores. Te aviso de que debes entender _bien_ esta sección antes de intentar usar reales.

1.119 ch_12a

Valores Reales.

En E un real no es más que otro valor de 32 bits. El compilador de E los trata como si fueran enteros o punteros, con la diferencia de que su representación interna significa algo totalmente diferente. El formato de los reales en E es el estandard IEEEsingle.

Un valor real es similar a un entero, con la salvedad de que tiene un punto en algún lugar. Por ejemplo, los siguientes son reales válidos:

```
3.14159    .1    1.    -12345.6
```

y estos no lo son:

```
.    1234
```

(es decir, debe tener al menos un "." y un caracter "0-9").

Puedes usar estos valores en prácticamente todos lo lugares en los que es legal un valor LONG, es decir, si tienes una función o estructura de datos que utiliza valores LONG cualesquiera, también utilizará reales.

```
DEF f=3.14
myobj.x:=.1
fun(f,2.73)
```

1.120 ch_12b

Computación con reales.

Debido a que los reales son como LONGs para E, éste utilizará con toda seguridad la aritmética de enteros con ellos cuando se usan en una expresión, lo cual seguramente no es lo que quieres. Además, también sería interesante poder convertir enteros en reales y viceversa. Con el operador "!" se puede hacer todo esto.

Supongamos que en los siguientes ejemplos a,b,c contienen valores enteros, y que x,y,z contienen valores reales.

```
   Por omisión, una expresión en E se considera una expresión entera.  Lo
que hace "!" cuando apararece en una expresión es:
- Cambia la expresión de  entero  a real.  Cualquier operador que le siga
  (+ * - / = <> > < >= <=)  realizará  operaciones con reales.  "!" puede
  aparecer tantas veces como quieras.
- La expresión que aparece antes de "!",  si hubo alguna, se convierte al
  tipo apropiado.

Ejemplos:
      x:=a!  -> convierte "a" a real, y guarda el resultado en x.  "a" es
             -> una expresión  entera,  que  se  cambia por real, lo cual
             -> implica una conversión.
      a:=!x!  -> convierte "x" a entero y guarda el resultado en "a".
      x:=y       -> aquí no se necesita "!" ya que no es necesario ningún
      x:=Ftan(y) -> operador matemático ni conversión.
      x:=!y*z  -> el operador "*" actua sobre 'y' y 'z' como reales, dado
               -> que  "!"  denota el todo como  una expresión  real.  El
               -> resultado real se guarda en x.

      a:=b!*x+y!  -> un ejemplo más complejo:  el entero 'b' se convierte
                  -> a real,  luego se multiplica por 'x' (como reales) y
                  -> se  le  suma  'y'  (como reales).  El  resultado  se
                  -> convierte a entero y se guarda en el entero 'a'".
      x:=!y*z-z*y+(a!)+z/z  -> todos los operadores (+ * - /) se computan
      z:=!x*Fsin(!x*y)      -> como reales,  y el entero 'a' se convierte
                            -> a real  en alguna parte  de  la expresión.
                            -> Dado que "(" ")" denotan  una nueva expre-
                            -> sión,  tiene su propio estado de  "!".  La
                            -> misma idea para la segunda función.
      IF !x<0.1 THEN WriteF('¡Valor real demasiado pequeño!\n')
      -> Como ves "!" también funciona con los seis oper. de comparación.
```

1.121 ch_12c

```
Funciones de reales predefinidas
------------------------------------

   Se  incluyen  algunas  funciones  matemáticas  de  transformación,  es
probable que se incluyan más en el futuro.

   Fsin(y)
   Fcos(y)
   Ftan(y)

   Las funciones sin(), ... de siempre.  Funcionan con radianes.

   Fabs(y)

   Computa el valor absoluto de y (|y|).

   Ffloor(y)
   Fceil(y)

   Computa el entero más próximo a 'y' (menor y mayor respectivamente).
```

```
Fexp(y)
Flog(y)
Flog10(y)
Fpow(y,z)
Fsqrt(y)
```

Computa 'e^y', 'ln(y)', 'log base 10', 'z^y' y raiz cuadrada de 'y'.

```
x,n:=RealVal(s)
```

Analiza la cadena 's' para producir el valor real 'x'. No tiene en cuenta los espacios o tabuladores que lleve delante. 'n' es el número de caracteres analizados desde el comienzo de la cadena, o 0 si la cadena no representa a un valor real. 'x' valdrá en ese caso 0.0. Acepta valores negativos (e incluso números sin ".").

```
Ejemplo:       RealVal(' 3.14 ')    resulta    3.14, 5
               RealVal('blabla')    resulta    0.0, 0
```

```
s:=RealF(s,x,n)
```

Retorna en 's' el valor real 'x' como cadenaE, con 'n' decimales. El máximo de 'n' es 8, incluso menos si la parte entera del número es grande. Un valor de 'n' igual a 0 indicará que no se quiere fracción. La cadena se retorna como resultado, para poder utilizarla en un WriteF().

```
Ejemplo: WriteF('real=\s\n',RealF(s,3.14159),4)  resulta  'real=3.1416\n'
```

RealF() se afana en realizar redondeos razonables para un cierto 'n', como muestra el ejemplo. Los números negativos también son tratados de forma correcta.

```
            RealF(s,-3.14159,0)    resulta    '-3'
```

1.122 ch_12d

```
Notas sobre la implementación de los reales.
--------------------------------------------
```

Como se dijo antes, el formato de reales de E es el IEEE (simple), esto significa que el código con reales anterior, que usaba el formato FFP con las funciones SpXxx se debe reescribir (como se indicaba en los docs de la v2.1b). EC v3.0 ya no apoya directamente la librería mathffp, y tendrás que abrir esta librería directamente si quieres usarla.

Se eligió IEEE simple porque:

- Los IEEE dobles no entran en un LONG.
- Las rutinas del formato FFP no utilizan el 68881 si está presente, las de IEEE sí lo hacen. Además, el formato IEEE es compatible con el 68881, que también utiliza el formato IEEE.
- IEEE es el formato estandard para reales, reconocido en todos los lugares, lo cual permite compatibildad de ficheros de datos entre software/plataformas.

Las rutinas de reales en E usan la mathieeesingbas.library y la mathieeesingtrans.library, las cuales no se ofrecían con la antigua versión 1.3 del sistema operativo. Esto quiere decir que si quieres

escribir bajo/apoyar la v1.3 del sitema operativo, y quieres usar los reales _internos_, debes asegurarte de que esas librerías están presentes (paracen estar disponibles, ¿quizas por medio de Commodore?).

Tanto EC como los programas que genera no abren esas librerías mientras no se usen reales.

Si falla todo lo demás, siempre puedes usar otras librerías de reales para usar reales bajo 1.3. Yo recomendaría el módulo 'tools/longreal.m', el cual usa dobles.

En un futuro, es probable que EC permita que las llamadas de la librería mathieee se substituyan por código de 68881 en línea de forma transparente.

[nota: se sabe que la v3.1 (v40) del sistema operativo del Amiga contiene un error en el código de IEEE. Usa un SetPatch que lo corrija.]

1.123 ch_13

Manejo de Excepciones.

Definición del controlador de excepciones (HANDLE/EXCEPT)
Uso de la función Raise()
Definición excepciones para funcs predefinidas (RAISE/IF)
Uso de identificadores de excepción

1.124 ch_13a

Definición del controlador de excepciones (HANDLE/EXCEPT).

El mecanismo de excepciones de E es básicamente igual al del lenguaje ADA; proporciona una reacción a los errores en tu programa, y un manejo complejo de recursos. NOTA: en E el término 'excepción' tiene poco que ver con las excepciones o interrupciones causadas directamente por los procesadores 680x0.

Un manejador de excepciones es un fragmento de código que se invoca cuando sucede un error en tiempo de ejecución. El sistema en tiempo de ejecución, o tú mismo, puedes 'señalar' que algo ha ido mal (esto es lo que se llama 'lanzar una excepción'), y luego el sistema en tiempo de ejecución probará y buscará el manejador de excepciones apropiado. Digo "apropiado" porque un programa puede tener más de un manejador de excepciones, en todos los niveles del programa. La siguiente puede ser la definición de una función normal (como ya sabemos):

```
PROC bla()
  /* ... */
ENDPROC
```

mientras que una función con un manejador de excepciones puede ser:

```
            PROC bla() HANDLE
              /* ... */
            EXCEPT
            /* ... */
            ENDPROC
```

donde el bloque entre PROC y EXCEPT se ejecuta como siempre, y si no hay ninguna excepción se salta el bloque entre EXCEPT y ENDPROC, abandonando el procedimiento en ENDPROC. Si se alcanza alguna excepción, ya sea dentro del procecimiento, o en cualquier función llamada desde ese bloque, se invocará un manejador de excepciones.

1.125 ch_13b

```
Uso de la función Raise().
--------------------------
```

Hay muchas formas de 'lanzar' una excepción, la más sencilla es por medio de la función Raise():

```
        Raise(IDexcepcion=0)
```

donde 'IDexcepción' es simplemente una constante que define el tipo de excepción, y lo usan los controladores para determinar lo que fue mál.

```
      ENUM NOMEM,NOFILE  /* y otros */

      PROC bla() HANDLE
        DEF mem
        IF (mem:=New(10))=NIL THEN Raise(NOMEM)
        mifunc()
      EXCEPT
        SELECT exception
          CASE NOMEM
            WriteF('¡Sin memoria!\n')
          /* ... y otras */
        ENDSELECT
      ENDPROC

      PROC mifunc()
        DEF mem
        IF (mem:=New(10))=NIL THEN Raise(NOMEM)
      ENDPROC
```

La variable "exception" del manejador siempre contiene el valor del argumento de la llamada al Raise() que lo invocó.

La función Raise() invoca al manejador de la función bla() en ambos New(), y retorna correctamente al que había llamado a bla(). La llamada New() de mifunc() habría llamado a su manejador de excepciones si ésta tuviera alguno. El alcance de un manejador va desde el comienzo del PROC en que se define hasta la palabra clave EXCEPT, incluyendo todas las llamadas que se hagan desde el procedimiento.

Esto tiene tres consecuencias:

A. Los manejadores se organizan de forma recursiva, y el manejador que se invoca se selecciona dependiendo de las llamadas de las funciones en tiempo de ejecución.
B. Si se lanza una excepción dentro de un manejador, se invoca el

manejador de nivel justamente anterior. Esta característica de los manejadores se puede usar para implementar esquemas complejos de reserva recursiva de recursos con gran facilidad, (como se verá).

C. Si se lanza una excepción en un nivel en el que no existe un manejador de menor nivel (o en programas que no tienen ningún tipo de manejadores), se finaliza el programa. (es decir, Raise(x) tiene el mismo efecto que CleanUp(0))

Otras funciones:

Throw(IDexcepcion,valor)

Como Raise(), sólo que ahora lleva un valor arbitrario con ellas. Y luego se puede escrutar ese valor en el manejador con la variable "exceptioninfo".

ReThrow()

No tiene argumentos. Simplemente hace un Throw() con el valor de excepción actual. Sí y solo sí es <>0

1.126 ch_13c

Definición de excepciones para funciones predefinidas (RAISE/IF).

Con lo que vimos hasta ahora de las excepciones ya tenemos grandes ventajas sobre la forma de definir la función "error()" de siempre, aunque hay que escribir demasiado para comprobrar la igualdad a NIL de cada llamada a New().

El sistema de manejo de excepciones de E permite la definición de excepciones para todas las funciones de E (como New(), OpenW(), etc.), y para todas las funciones de Librería (OpenLibrary(), AllocMem(), etc.), incluso para aquellas que se incluyen como módulos.

Sintaxis: RAISE <IDexception> IF <func> <comp> <valor> , ...
la parte que sigue a RAISE se puede repetir con una ",".

```
Ejemplo:    RAISE NOMEM IF New()=NIL,
                  NOLIBRARY IF OpenLibrary()=NIL
```

la primera línea quiere decir algo como: "siempre que una llamada a New() resulte en NIL, lanza automáticamente la excepción NOMEM". <comp> puede ser cualquiera entre '= <> > < >= <='. Tras esta definición, a lo largo de nuestros programas podemos escribir: 'mem:=New(size)' sin tener que escribir 'IF mem=NIL THEN Raise(NOMEM)'.

Fíjate que la única diferencia es que 'mem' nunca recibe un valor si el sistema invoca al manejador: para cada llamada a New() se genera código que comprueba su resultado y llama a Raise() si es necesario.

Vamos a implementar un pequeño ejemplo que sería complejo de resolver sin manejo de excepciones: llamamos a una función recursivamente, y reservamos un recurso en cada llamada (en este caso se reserva memoria antes de la llamada recursiva, y se libera después). ¿Qué ocurre si en

algún momento en la recursión ocurre un fallo severo, y tenemos que dar el programa por terminado? Cierto, al dejar el programa no podríamos (en un lenguaje convencional) liberar todos los recursos reservados en niveles de recursión inferiores, debido a que todos los punteros a tales áreas de memoria se encuentran en variables locales inalcanzables. En E, simplemente lanzamos una excepción, y lanzamos otra al final del manejador, llamando de esta forma recursivamente a los manejadores, y liberando todos los recursos. Ejemplo:

```
CONST SIZE=100000
ENUM NOMEM  /* ,... */

RAISE NOMEM IF AllocMem()=NIL

PROC main()
  reserva()
ENDPROC

PROC reserva() HANDLE
  DEF mem                 -> veamos cuantos bloques de
  mem:=AllocMem(SIZE,0)   -> memoria se pueden obtener.
  reserva()               -> realiza la recursión.
EXCEPT DO
  IF mem THEN FreeMem(mem,SIZE)
  ReThrow()        -> llama recursivamente a los manejadores.
ENDPROC
```

Por supuesto, esto es una simulación de un problema de programación natural, que normalmente es bastante más complejo, y que por ello hace más evidente la necesidad del manejo de excepciones. Hecha un vistazo al código fuente de la utilidad "D.e" para ver un programa real de ejemplo, en el que el manejo de errores se complicaría sin manejo de excepciones.

El "DO" que va después de EXCEPT indica que en lugar de saltar a ENDPROC, el código principal simplemente continúa la ejecución en el manejador cuando llega a ese punto. También fija la excepción a 0. Puede ser útil si en el manejador liberas recursos locales al PROC.

1.127 ch_13d

```
Uso de ID de excepciones.
-------------------------
```

En realidad un ID de excepción es un valor de 32-bits normal (por supuesto), y puedes pasar lo que quieras al manejador de excepciones: por ej., algunos los usan para pasar cadenas de descripción de errores

```
Raise('¡No pude abrir la "gadtools.library"!')
```

Sin embargo, si quieres usar las excepciones de una forma expandible y deseas poder usar módulos futuros que lanzen excepciones sin definir en tu programa, sigue las siguientes normas:

- Usa y define el ID 0 como "no error" (finalización normal).
- Usa los IDs 1-10000 para excepciones específicas de tu programa.
  Defínelos como siempre con ENUM:

```
ENUM OK,NOMEM,NOFILE,...
```

(OK será 0, y los demás 1+)

- Los ID's 12336 a 2054847098 (todos ellos consisten de letras mayúsculas o minúsculas y dígitos, y tienen longitud 2, 3 o 4, entre comillas) están reservados para excepciones comunes. Una excepción común es una excepción que no es necesario definir en tu programa, y que puede ser usada por los implementadores de módulos (con funciones) para lanzar excepciones: por ejemplo, si diseñas un conjunto de procedimientos que realizan cierta tarea, puede ser que desees lanzar excepciones. Dado que podrías querer usar esas funciones en varios programas, no sería práctico tener que coordinar los IDs con el programa principal, es más, si usas más de un conjunto de funciones (en un módulo, en el futuro) y cada módulo tuviera un identificador diferente para 'sin memoria', las cosas se harían impracticables. Ahí es donde entran las excepciones comunes: el ID común para sin-memoria es "MEM" (incluyendo las comillas): cada implementador debería escribir simplemente

  Raise("MEM")

  desde cualquier procediemiento, y el programador que use el módulo sólo necesita proporcionar un manejador de exepciones que entienda "MEM".

  FUturos módulos que contengan conjuntos de funciones especificarán las excepciones que pueden alcanzar cada uno de esos procedimientos, y si éstas coinciden con los IDs de otros procedimientos, la tarea del programador que tiene que tratar con excepciones se verá simplificada significativamente.

  Ejemplos:

  (sistema)

"MEM"	sin memoria
"FLOW"	pila (prácticamente) llena
"STCK"	recolector de residuos con problemas de pila
"^C"	Ruptura por Control-C
"ARGS"	args erróneos

  (exec/librerias)

"SIG"	no se pudo reservar señal
"PORT"	no se pudo crear puerto de mensajes
"LIB"	library no disponible
"ASL"	no asl.library
"UTIL"	no utility.library
"LOC"	no locale.library
"REQ"	no req.library
"RT"	no reqtools.library
"GT"	no gadtools.library (similar para las demás)

  (intuition/gadtools/asl/gfx)

"WIN"	fallo al abrir ventana
"SCR"	fallo al abrir pantalla
"REQ"	no se pudo abrir requester
"FREQ"	no se pudo abrir requester de ficheros
"GAD"	no se pudo crear gadget
"MENU"	no se pudo crear menu(s)
"FONT"	problema obteniendo fuente

  (dos)

```
        "OPEN"        no pude abrir fichero/fichero no existe
        "OUT"         problemas escribiendo
        "IN"          problemas leyendo
        "EOF"         fin de fichero no esperado
        "FORM"        error en formato de entrada
        "SEG"         problemas con loadseg

  La tendencia general es:
  * todas las letras mayúsculas para excepciones generales del sistema,
  * mezcladas para excepciones usadas por >1 app, pero no suficientemente
    generales,
  * todas minúsculas para excepciones lanzadas dentro  varios  módulos de
    tu propia aplicación.

- Todas los demás, (incluyendo los IDs negativos) estan reservados.
```

1.128 ch_14a

```
Programación Orientada a Objetos en E.
--------------------------------------
```

Las características que se describen en esta sección están agrupadas como tal ya que constituyen lo que generalmente se conoce como los tres principales componentes esenciales que hacen a un lenguaje 'Orientado a objetos' (es decir, herencia - ocultamiento de datos - polimorfismo). Sin embargo, en E no son un 'tema aparte' ya que cada uno de ellos se puede usar de todas formas junto con otras características del E.

```
        Herencia de objetos
        Ocultamiento de datos               (EXPORT/PRIVATE/PUBLIC)
        Métodos y métodos virtuales
        Constructores, Destructores y Super-Métodos (NEW,END,SUPER)
```

1.129 ch_14b

```
Herencia de Objetos.
--------------------
```

Siempre resulta incómodo el no poder expresar dependencias entre OBJECTos, o reusar código que funciona en un OBJECTo particular con otro mayor que encapsula al primero. La herencia de objetos te permite justamente hacer eso en E. Cuando tienes un objeto 'a':

```
            OBJECT a
              sig, indice, term
            ENDOBJECT
```

puedes crear un nuevo objeto 'b' que tenga las mismas propiedades de 'a' (y es compatible con el código que utilice 'a'):

```
            OBJECT b OF a
              bla, x, burp
            ENDOBJECT
```

y que equivale a:

```
            OBJECT b
```

```
              sig, indice, term         /* de a */
              bla, x, burp
            ENDOBJECT
```

de forma que con DEF p:b podrás acceder directamente no sólo a 'p.bla' como siempre, sino también a 'p.sig'.

Por ejemplo, si se tiene un módulo con un OBJECTo para implementar un cierto tipo de datos (por ejemplo, una lista doblemente enlazada) con procedimientos de apoyo, se puede simplemente heredar de ella, añadiendo datos propios al objeto, y usar las funciones _existentes_ para manipular la lista. Sin embargo la herencia sólo ofrece toda su potencia cuando se combina con métodos.

1.130 ch_14c

Ocultamiento de datos (EXPORT/PRIVATE/PUBLIC).

E posee un mecanismo de ocultamiento de datos bastante controlable. Otros lenguajes, como el C++, usan el ocultamiento de datos en clases, lo cual nos lleva a la necesidad de trampas (como 'friends'), y hace el ocultamiento de datos inseguro (Eiffel). El ocultamiento de datos de E funciona a nivel de módulos, lo cual permite modelar el ocultamiento de datos a nivel de clases, pero también permite esquemas más inteligentes.

PUBLIC y PRIVATE declaran visible o no al mundo exterior una sección de un objeto; en este caso, el mundo exterior es todo el código ajeno al módulo. Por otra parte, todo es visible para el código del módulo.
Ejemplo:

```
            OBJECT misdatos PRIVATE          -> todo el objeto es privado
              bla:PTR TO misdatos, burp, grrr:INT
            ENDOBJECT

            OBJECT aaargh
              blerk:PTR TO aaargh                    -> público
            PRIVATE
              x:INT, y:INT, z:INT                    -> privado
            PUBLIC
              hmpf[10]:ARRAY OF misdatos             -> públic de nuevo
            ENDOBJECT
```

Un objeto es público por omisión, la aparición de PRIVATE o PUBLIC actua como interruptor para la visibilidad del objeto en ese momento. En el primer objeto todo es privado. El segundo sólo tiene (x,y,z) como privado. Las palabras clave PRIVATE y PUBLIC pueden aparecer:
- En la línea de cabecera del objeto.
- Como línea en si misma dentro de la definición del objeto.
- Precediendo declaraciones en la definición del objeto.
(es decir, prácticamente en cualquier lugar)

¿Por que ocultamiento de datos?
Si quieres saber por qué el ocultamiento de datos es una buena técnica, es probable que quieras leer algún buen libro de OO. Pero en resumen: por lo general se supone que gran cantidad de los problemas de mantenimiento y mejora de fragmentos grandes de código se deben al hecho

de que resulta dificil cambiar las cosas, debido a que gran cantidad de código empieza a depender de ciertas estructuras del código. Si ocultas un objeto, sólo el código dentro del módulo dependerá del formato de los objetos, y podrás modificar facilmente la representación de un objeto y el código que trabaja con él (por ejemplo, podrías cambiar fácilmente la implementación de una pila de ARRAY a una lista). Si gran parte del código de una gran aplicación depende del hecho de que la pila es un ARRAY, no podrás cambiarlo con facilidad, lo cual acarreará problemas. En resumen, intenta ocultar tanto como puedas sin llegar a ser demasiado restrictivo en el uso de tu objeto. El uso de métodos te permitirá a menudo mantener todo el objeto privado.

1.131 ch_14d

Métodos y métodos virtuales.

Un método es muy parecido a un PROC, con la diferencia de que forma parte de un OBJECTo. Además, también permite explotar el polimorfismo en los objetos, como veremos a continuación.
Definición de un método:

```
OBJECT blerk PRIVATE
  x:PTR TO blerk, y:INT, z
ENDOBJECT

PROC getx() OF blerk IS self.x
```

la parte 'OF blerk' le dice al compilador que pertenece al objeto 'blerk'. Fíjate en que, aparte de 'OF', la sintaxis es completamente igual a la de un 'PROC', sin embargo, no se puede invocar como tal, y también funciona de forma diferente.

'self' es una variable local que está disponible en todo método, y es un puntero al objeto al que pertenece el método (en este caso 'self:PTR TO blerk'). Esta función sólo retorna el valor de campo x de blerk, lo cual tiene sentido, ya que esto te permite modificar más tarde lo que representa x.

Podemos llamar a los métodos de forma similar a como se hace la selección de objetos ".":

```
DEF a:PTR TO blerk
NEW a
...
a.getx()            -> invoca método getx() en el objeto a
```

en este ejemplo, sobre la invocación "a" toma el valor de 'self' durante la jecución de getx().

Como ves, el uso de métodos es interesante, aunque no hemos visto su poder real, el cual sólo aparece cuando se usa junto con la herencia.

Si heredo un objeto que tiene métodos, de forma automática obtengo esos métodos en el nuevo objeto:

```
OBJECT burp OF blerk PRIVATE   -> como blerk, + campo extra
  prut:INT
ENDOBJECT
```

```
DEF b:PTR TO burp
NEW b
...
b.getx()                        -> mismo método
```

Lo interesante viene ahora, ya que en lugar de heredar un método, también puedes redefinirlo:

```
PROC getx() OF burp IS self.x+1
```

(no hace falta decir que también puedes añadir nuevos métodos)

Así que, cuando sea apropiado, puedes optar por modificar el funcionamiento de los métodos que obtenemos de otros objetos, mientras que el interface a él (en este caso 'getx()') se mantiene igual. Esto no sólo permite la reutilización del código de forma selectiva, también podemos hacer uso del polimorfismo:

```
PROC hazalgo(o:PTR TO blerk)
  ...
  o.getx()
  ...
ENDPROC

hazalgo(a)
hazalgo(b)
```

podemos llamar a este PROC tanto con 'a' como con 'b', ya que ambos son compatibles con un objeto blerk. Aunque, ¿Cual de las dos implementaciones de getx() se invoca con o.getx()? Repuesta: las dos. Las llamadas a métodos en E son lo que se llaman métodos virtuales en otros lenguajes: actúan dinámicamente en el tipo real del objeto (o) y llaman al método apropiado.

Un ejemplo más claro:

```
-> Ejemplo clásico de polimorfismo OO

OBJECT loc
  PRIVATE posx:INT, posy:INT
ENDOBJECT

OBJECT punto OF loc
  PRIVATE color:INT
ENDOBJECT

OBJECT circulo OF punto
  PRIVATE rados:INT
ENDOBJECT

PROC muestra() OF loc IS WriteF('¡Soy una Localización!\n')
PROC muestra() OF punto IS WriteF('¡Soy un Punto!\n')
PROC muestra() OF circulo IS WriteF('¡Soy un Círculo!\n')

PROC main()
  DEF x:PTR TO loc,l:PTR TO loc,p:PTR TO punto,c:PTR TO circulo
  ForAll({x},[NEW l,NEW p,NEW c],`x.muestra())
ENDPROC
```

aquí 'x' es un PTR TO loc, por lo que podrías esperar que 'x.muestra()' escribiera 'Soy una Localización' tres veces, pero en su lugar escribe la cadena correcta para cada objeto.

Si quisieramos escribir esto en un lenguaje no OO, necesitaríamos un SELECT para cada operación como muestra(), comprobando cierto valor presente en el objeto para ver de qué tipo es. Si quisiera añadir una nueva forma a las anteriores, como:

```
OBJECT elipse OF circulo
```

necesitaría cambiar todos los SELECTs a lo largo de la aplicación para que lo tengan en cuenta. Con el polimorfismo de objetos, simplemente escribiría un método muestra(), y TODO el código de la aplicación que llame a x.muestra() actuaría correctamente cuando x sea un PTR TO elipse, ¡incluso sin tener que recompilar!

Es dificil mostrar el poder de esto en unos pocos ejemplos, la mejor forma de descubrirlo es usándolo en aplicaciones reales, y: como dije antes, leyendo un libro sobre ello.

¿Cómo funciona el polimorfismo?

En los ejemplos anteriores, está claro que el compilador no sabe a qué método llamará. Ese es el motivo por el cual E usa un 'objeto clase', y cada objeto creado obtiene un puntero a este objeto. En el objeto clase se guarda toda la información que es común a todos los objetos de ese tipo, tales como punteros a métodos. Cuando el compilador detecta una llamada como x.muestra(), en lugar de mirar directamente en la muestra() que pertenece al tipo de x (es decir loc), generará código que llama automáticamente a la muestra() de punto, cuando x es realmente un objeto punto. Esto recibe a veces el nombre de enlace en tiempo de ejecución.

Los objetos que tienen métodos son, por tanto, 4 bytes mayores de lo esperado, ya que contienen un puntero al objeto clase. El puntero lo inicializa automáticamente NEW, razón por la que _de momento_ NEW es la única forma de crear tales tipos de objetos.

Si un método se declara con el único propósito de permitir que las subclases lo redefinan (este tipo de clases se conocen en algunos lenguajes como clases bases virtuales), se puede usar EMPTY:

```
PROC bla() OF obj IS EMPTY
```

entonces se puede redefinir en las subclases. Dado que puede que el programador no implemente todos los métodos de una vez, no es un error que se ejecute el método anterior. Simplemente retornará 0 o NIL.

Se pueden añadir métodos a los OBJECTos del sistema:

```
OBJECT migadget OF gadget       -> ¡de intuition!
  -> campos extra aquí
ENDOBJECT
PROC creategadget() OF migadget IS ...
```

un puntero a un objeto como éste es compatible con el puntero normal a gadget, es decir, se puede añadir directamente a un ventana, etc...

1.132 ch_14e

Constructores, Destructores y Super-Métodos.

El constructor puede tener un nombre cualquiera, aunque suele darsele el mismo nombre que la clase. Incluso se pueden tener varios constructores para una misma clase. El constructor se llama directamente en un

```
objeto creado con NEW:
            NEW obj.pila()

   Sin  embargo,  los  destructores deben llamarse "end".  Y el objeto se
destruye con:
            END obj
y si 'obj' tiene un método end(), se le llama de forma automática.  end()
no debe tener argumentos, y no tiene sentido que retorne un valor.

   El  super-método  de  un  método es el método de su super-clase con el
mismo nombre.  Algunas veces es útil llamar a este método para incluir su
comportamiento  al  de  tus clases,  sin  embargo,  debido  a  que lo has
redefinido,  el  llamar  al  super-método  por  su  nombre simplemente se
llamaría  a  sí  mismo  (¡!).  La palabra clave SUPER te permite llamar a
cualquier método de tu super-clase (o la super-clase de algun otro):
            SUPER obj.metodo()
este  fragmento  de código se puede usar como expresión y como sentencia.
Sin embargo, hay que tener cuidado, ya que si el supermétodo llama a otro
método de ese objeto, llamará a la versión redefinida, no la de su propio
'nivel'  (aunque  eso  suele ser lo que queremos).  Además, el compilador
mira  el  tipo estático de 'obj' para encontrar su superclase, no el tipo
dinámico (aunque pueda tener ese resultado).
```

1.133 ch_15

```
Ensamblador en-línea.
---------------------

         Compartición de identificadores
         Ensamblador en-línea comparado a un macroensamblador
         Utilización de datos binarios (INCBIN/CHAR..)
         OPT ASM
         Ensamblador en-línea y variables registro
```

1.134 ch_15a

```
Compartición de identificadores.
--------------------------------

   Como  habrás apreciado en el ejemplo de Mnemotécnicos ensamblador, las
instrucciones  en  ensamblador se pueden mezclar libremente con código E.
El gran secreto es que el compilador incluye un ensamblador completo.

   Además  de los  modos  de direccionamiento  normales del  ensamblador,
puedes usar los siguientes identificadores del E:
         mietiqueta
         LEA mietiqueta(PC),A1 /* etiquetas */

         DEF a              /* variables */
         MOVE.L (A0)+,a     /* fíjate que <var> es <offset>(A4) (o A5) */

         MOVE.L dosbase,A6     /* identificadores de llamada a library */
```

```
         JSR    Output(A6)

         MOVEQ  #TRUE,D0       /* constantes */
```

   El ensamblador de EC permite las siguientes construcciones,
donde
```
         n       = numregistro
         x       = índice
         lab     = etiqueta, de: "etiqueta:" o "PROC etiqueta()"
         abs     = dirección absoluta
         s       = tamaño. L, W o B, el que sea apropiado.
```

- modos de direccionamiento que permite EC:
  Dn, An, (An), (An)+, -(An), x(An), x(An,Dn.s),
  lab(PC), lab(PC,Dn.s), abs, abs.W
  (nota: escribe abs.W en hexadecimal, para no confundirlo con un valor real, es decir escribe MOVE.L $4.W,A6)

- permitidos parcialmente:
  #<expconst>

- no permitidos:
  lab (igual que abs), #lab
  usa LEA lab(PC),An en su lugar.

- modos extra:
  var.s (transfiere contenido de var. s puede ser ".W" o ".B", por omisión es ".L")

  LibraryFunction(A6)

  ejemplo:

```
         ...
         MOVE.W mivar.W,D0     -> mueve palabra baja de 'mivar'
         ...
         MOVE.L dosbase,A6
         JSR    Write(A6)
```

   Además, E permite que las funciones retornen directamente un registro:
```
         ENDPROC D0
```
y también puedes interpretar esto cómo valores de retorno mútiples, es decir D0/D1/D2.

1.135 ch_15b

El ensamblador en línea comaparado con un macro ensamblador.

   El ensamblador en-línea se diferencia del macro-ensamblador de siempre de alguna forma, debido principalmente al hecho de que es una extensión de E, y por tanto sigue la sintaxis de E. Principales diferencias:

- los comentarios se hacen con /* */ y no ";", con significado distinto.
- las palabras clave y los registros van en mayúsculas, todo depende de si va en mayúsculas o minúsculas.
- no hay macros ni otros lujos de los ensambladores (bueno, está todo el lenguaje E para cubrir esto ...)

- debes tener en cuenta que los registros A4/A5 no se deben ensuciar con el código en ensamblador, ya que los utiliza el código E, si tu código se puede llamar desde código que reserva registros, debes preservar D3-D7. La siguiente instrucción puede ayudar en caso de problemas:
  MOVEM.L D3-D7,-(A7); /* ensam en línea */; MOVEM.L (A7)+,D3-D7
- _TODAVIA_ no se permiten model LARGE/hunk-reloc en ensamblador. Esto prácticamente significa que de momento debes usar direccionamiento relativo a PC (que de todas formas es más rápido).

1.136 ch_15c

```
Utilización de datos binarios (INCBIN/CHAR..).
----------------------------------------------

INCBIN

Sintaxis:         INCBIN <fichero>

incluye el fichero binario en el punto exacto de la sentencia, por tanto
debe separase del código. Ejemplo:
            mitab: INCBIN 'df1:data/blabla.bin'

LONG, INT, CHAR

Sintaxis:         LONG <valores>,...
                  INT <valores>,...
                  CHAR <valores>,...

te permite colocar datos binarios directamente en el programa. Funciona
como DC.x en ensamblador. Señalar que la sentencia CHAR también recibe
cadenas, y siempre estarán alineados a direcciones pares. Ejemplo:
            misdatos: LONG 1,2; CHAR 3,4,'¡Que hay amigos!',0,1
```

1.137 ch_15d

```
OPT ASM.
--------

   OPT ASM también se discute en el apartado sobre OPT. Permite operar
con 'EC' como si fuera un ensamblador. No hay ninguna razón para usar EC
en lugar de otros macro-ensambladores, a excepción de que es significati-
vamente más rápido que por ejemplo A68k, así como DevPac, y pierde con
respecto a AsmOne (sniff 8-{). También será tedioso el pasar los viejos
discos con fuentes de Seka por EC, debido a las diferencias descritas con
anterioridad. Si quieres escribir programas en ensamblador con EC, y
quieres mantener tus fuentes compatibles con otros ensambladores, simple-
mente precede todos los elementos específicos de E con un ";", EC los
usará, y cualquier otro ensamblador los tomará como comentarios. Empieza
los comentarios normales con un smiley (";->").
Ejemplo:
            ; OPT ASM
            comienzo:  MOVEQ   #1,D0           ;-> haz algo tonto
```

```
                        RTS                         ;-> y sal
```
esto lo ensamblará cualquier ensamblador, incluyendo EC.

1.138 ch_15e

```
Ensamblador en línea y variables registro.
------------------------------------------
```

Las variables registro son una buena ayuda al ensamblador en línea, ya que funcionan como registros, aunque al mismo tiempo tienen un identificador claro en lugar de Dx, y además, el código de E los guarda y recupera de forma automática. Ejemplo:

```
            PROC bla()
              DEF count:REG
              MOVEQ #10,cuenta
            bucle: WriteF('cuenta=\d\n',cuenta)
              DBRA cuenta,bucle
            ENDPROC
```
todas las instrucciones que funcionan con Dx EA funcionan con variables registro. Ejemplos:

```
            MOVEQ #1,a
            MOVEM.L D0/D1/a/b/A0,-(A7)
            LSL.L a,b
            ....
```
como se sabe, EC usa D3-D7 para estas variables registro. Si quieres escribir código que mezcle libremente ensamblador con E, es recomendable mantener variables usadas con frecuencia en variables registro, y temporales en D0-D2/A0-A3/A5.

1.139 ch_16

```
Notas técnicas y de implmentación.
----------------------------------
```

```
        Palabra clave OPT
        Modelo pequeño/grande (SMALL/LARGE)
        Organización de la pila (STACK)
        Límites fijos (hardcoded)
        Mensajes error, advertencias y comprobación no-referencia
        Organización y reserva del buffer del compilador
        Reserva de registros
```

1.140 ch_16a

```
La palabra clave OPT.
---------------------
```

```
OPT, LARGE, STACK, ASM, NOWARN, DIR, OSVERSION, MODULE, EXPORT, RTD, REG
```

```
Sintaxis:         OPT <opciones>,...

Te permite cambiar algún funcionamiento del compilador:

LARGE           Fija el modelo de código y datos en grande  (LARGE).  Por
                omisión es  pequeño (SMALL);  El compilador suele generar
                código relativo a PC, con un tamaño máximo de 32k.  LARGE
                no pone tales límites, y genera bloques (hunks) reloc.
STACK=x         Fija el tamaño de la pila a 'x' bytes.  Úsalo sabiendo lo
                que haces.  Normalmente  el  compilador  hace  una  buena
                estimación del espacio de pila que se necesita.
ASM             Pone el compilador en modo  ensamblador.  A partir de ese
                momento  sólo se  permiten instrucciones  en ensamblador,
                y no se genera código de inicialización.
NOWARN          Elimina  las  advertencias.  El compilador  te avisará si
                *piensa* que  tu  programa  es  incorrecto,  aunque  sea
                sintácticamente correcto.  (mira NOWARN, -n)
DIR=dirmodulos  Indica el directorio en  el que el compilador buscará los
                módulos,  por omisión='EMODULES:'.
OSVERSION=vers  Por  omisión=33  (v1.2).  Fija  la  versión de  KickStart
                mínima  (como  37 para  v2.04) bajo  la que  funcionará tu
                programa.  De esta forma, tu programa fallará al abrir la
                dos.library  en el  código  de  inicialización cuando  tu
                código  se  ejecuta  en  máquinas  anteriores.  De  todas
                formas, es  de más utilidad  al usuario si  compruebas la
                versión tú mismo y das un mensaje de error apropiado.
MODULE          Indica que este código es un módulo.
EXPORT          Exporta automáticamente las declaraciones del módulo.
RTD             Genera RTDs en vez de RTS en fuente principal. Sólo 020+.
                [optimización experimental]
020,881,040     Genera código para esa CPUs. Aun no utilizable.
REG=n           Usa 'n' registros para reserva de registros.

Ejemplo:    OPT STACK=20000,NOWARN,DIR='df1:modules',OSVERSION=39,REG=3
```

1.141 ch_16b

```
Modelo SMALL/LARGE.
-------------------

   AmigaE te permite  elegir entre modelo de datos/código SMALL y  LARGE.
Fíjate que la  mayoría de los programas que  escribirás (especialmente si
acabas de empezar con E) entrarán en 32KB al compilarlos:  no tendrás que
preocuparte  de poner  algún modelo  de  generación de  código.  Te darás
cuenta de la  necesidad de un modelo  LARGE tan pronto como  EC empieze a
quejarse de  que ya  no puede poner tu código en  32KB.  Para compilar un
fuente con modelo LARGE:
            1> ec -l sizy.e
o mejor incluso, incluye la sentencia
            OPT LARGE
al principio de tu código.
```

1.142 ch_16c

```
Organización de la pila.
-------------------------
```

Para guardar las variables locales y globales, el sistema en tiempo de ejecución de un ejecutable generado por AmigaE reserva un bloque de memoria, del cual tomará una parte fija para almacenar todas las variables globales. El resto se usará dinámicamente según se vayan llamando las funciones. Cuando se llama a una función en E, se reserva espacio en la pila para almacenar todos sus datos locales, el cual se libera al salir de la función. Ese es el motivo por el que tener arreglos grandes de datos locales puede se peligroso cuando se usa recursivamente: todos los datos de las llamadas anteriores a la misma función siguen estando en la pila y utilizan partes grandes del espacio libre de la pila. Sin embargo, si los procedimientos se llaman de una forma lineal, no hay motivo para que la pila se llene.
Ejemplo:

```
            global data:            10k (arreglos, etc.)
            local data PROC #1:      1k
            local data PROC #1:      3k
```

El sistema en tiempo de ejecución siempre reserva 10k extra por encima de esto para una recursión normal (por ejemplo con arreglos locales pequeños) y buffers y espacio de sistema adicionales, por lo que reservará 24k de espacio de pila.

1.143 ch_16d

```
Límites fijos.
--------------
[ Significado: (+-)   mas o menos, depende de la situación.
               (s.l)  sin límite claro, pero este parece razonable.]

-------------------------------------------------------------------------
OBJETO/ELEMENTO                                  TAMAÑO/CANTIDAD/MAX
-------------------------------------------------------------------------

valor CHAR                                       0 .. 255
valor INT                                        -32 Kb .. +32 Kb
valor LONG/PTR                                   -2 Gb .. +2 Gb

logitud identificador                            100 bytes (s.l.)
logitud línea de código                          2000 tokens léxicos (+-)
logitud código                                   2 Gb (teoricamente)
listas constantes                                algunos cientos elem (+-)
cadenas constantes                               1000 caracteres (s.l.)
anidamiento max. de bucles (IF, FOR etc.)        500 niveles
anidamiento max. de comentarios                  infinito

num. de vars. locales por procedimiento          8000
num. de vars. globales                           7500
num. de argumentos a func. propias               8000 (junto con locales)
num. args. funciones varargs de E (WriteF())     64 (v2.1) / 1024 (v2.5)
```

```
un  objeto (reservado local/global o dinám.)    8 Kb
un  arreglo, lista o cadena (local o global)    32 Kb
una cadena (dinámicamente)                      32 Kb
una lista (dinámicamente)                       128 Kb
un  arreglo (dinámicamente)                     2 Gb
obejtos con NEW                                 64Kb elem.
CHAR/INT/LONG con NEW                           2 Gb

datos locales por procedimiento                 250 Mb
datos globales                                  250 Mb

tamaño código de un prodeciemiento              32 Kb
tamaño código del ejecutable                    SMALL 32 k, LARGE 2 Gb
límite práctico actual (puede cambiar)          2-5 (v2.1) / 10 Mb (v2.5)

tamaño buffer de cdogio generado e identific.   relativo al fuente
tamaño buffer de etiquetas/saltos e intermed.   (re)reservado independ.
```

1.144 ch_16e

```
Mensajes de error, advertencias y comprobación de no referencia.
----------------------------------------------------------------
```

Al compilar tu código fuente con EC, algunas veces obtienes un mensaje del tipo UNREFERENCED: <ident>, <ident>, ... Esto sucede en casos en los que declaras variables, funciones o etiquetas, pero no las usaste. Esto es un servicio extra que te ofrece el compilador para encontrar todos esos errores difíciles de detectar.

Hay varias advertencias que el compilador ofrece para notificar que hay algo que puede ir mal, aunque no es realmente un error.

- 'A4/A5 used in inline assembly'
  'A4/A5 usado en ensamblador en línea'
  Esta es la advertencia que obtendrás cuando usas los registros A4 o A5 en tu código ensamblador. El motivo de esto es que esos registros los usa E internamente para direccionar las variables globales y locales respectivamente. Por supuesto, puede haber una buena razón para utilizarlos, como hacer un MOVEM.L A4/A5,-(A7) antes de una parte grande de ensamblador en línea.

- 'keep an eye on your stacksize'
  'hecha un ojo al tamaño de la pila'
- 'stack is definitely too small'
  'la pila es definitivamente muy pequeña'
  Ambas pueden ser provocadas por el uso de OPT STACK=<tamaño>. El compilador simplemente contrastará tu <tamaño> con su estimación, y escribirá la primera advertencia si piensa que está bien aunque un poco ajustado, y la última si probablemente es muy pequeño.

- 'suspicious use of "=" in void expressions (s). (line %d)'
  'uso sospechoso de "=" en expersiones void (s). (línea %d)'
  Esta advertencia aparece si escribes expresiones como 'a=1' como una sentencia. La razón de esto es que una comparación no tiene mucho

sentido como sentencia, aunque la razón principal es que puede ser motivo del frecuente error tipográfico de 'a:=1'. Puede ser dificil de encontrar el olvidode ":", y puede tener consecuencias desastrosas.

- 'module changed OPT settings'
  'el módulo cambió las opciones OPT'
  Si usas un módulo que usa OPT OSVERSION=37, ésta también afectará al programa principal. La advertencia te avisa de ésto. Para eliminarla sólo tienes que poner un OPT del mismo tipo en el programa principal.

- 'variable used as function'
  'variable usada como función'
  En v3, cualquier variable se puede usar como una función. Esta advertencia está para _avisarte_ por si lo haces de forma accidental.

- 'code outside PROCs'
  'código fuera de PROCs'
  Has escrito código entre PROCs, lo cual es de rara utilidad.

Errores.
El compilador escribirá la línea de código fuente que causó el error a continuación de éste, y un cursor en el punto exacto del error. El cursor denota el punto en el que se hallaba el compilador cuando _descubrió_ el error, por lo que parece lógico pensar que el símbolo que provocó el error es el que está justo _antes_ del cursor.

- 'Sintaxis error'
  'Error de sintaxis'
  El error más común. Este error aparece tanto cuando no hay otro error apropiado, como cuando la estructura de tu código es demasiado anormal.

- 'unknown keyword/const'
  'palabra clave/constante desconocida'
  Has utilizado un identificador en mayúsculas (como "IF" o "TRUE"), y el compilador no pudo encontrar una definición de él. Causas:
  * palabra clave mal escrita
  * usaste una constante, pero olvidaste definirla en una sentencia CONST
  * te olvidaste de indicar el módulo en el que se define la constante

- '":=" expected'
  'se esperaba ":="'
  Has escrito una sentencia FOR o de asignación, y pusiste otra cosa diferente de ":=" en su lugar.

- 'unexpected characters in line'
  'caracteres en la línea no esperados'
  Usaste caracteres que no tienen significado sintáctico fuera de una cadena. Por ejemplo: "@!&\~"

- 'label expected'
  'se esperaba una etiqueta'
  En algunos puntos, por ej. después de la palabra clave PROC o JUMP, se pide un identificador de etiqueta, y escribiste algo diferente.

- '"," expected'
  'se esperaba ","'

Al especificar una lista de elementos (por ejemplo una lista de parámetros), escribiste algo que no era una coma.

- 'variable expected'
  'se esperaba una variable, ejemplo: FOR <var>:= ... etc.

- 'value does not fit in 32 bit'
  'el valor no entra en 32 bits'
  Al especificar un valor constante diste un número demasiado grande, ej: $FFFFFFFFF, "abcdef". También ocurre cuando defines un SET con más de 32 elementos.

- 'missing apostrophe/quote'
  'falta apóstrofe/comilla'
  Te olvidaste del ' al final de una cadena.

- 'incoherent program structure'
  'estrutura de programa incoherente'
  * empezaste un nuevo PROC antes de acabar el anterior,
  * no anidas los bucles correctamente, por ejemplo:

```
FOR
  IF
  ENDFOR
ENDIF
```

- 'illegal command-line option'
  'opción de línea de comandos ilegal'
  Al especificar 'EC -opt fuente' escribiste algo para '-opt' que no es una opción legal de EC.

- 'division and multiplication 16bit only'
  'división y multiplicación sólo con 16 bit'
  El compilador detectó que estabas a punto de usar 32 bits para * o /. Lo cual no tendría el resultado deseado en tiempo de ejecución.
  (mira Mul() y Div())

- 'superfluous items in expression/statement'
  'elementos superfluos en expresión/sentencia'
  Después de haber compilado tu sentencia, el compilador aún encontró tokens léxicos en vez del final de línea. Es probable que te hallas olvidado del <lf> o ";" para separar dos sentencias.

- 'procedure "main" not available'
  'no se dispone del procedimiento "main"'
  ¡Tu programa no incluye un procedimiento main()!

- 'double declaration of label'
  'doble declaración de una etiqueta'
  Declaraste una etiqueta dos veces, por ejemplo:

```
etiqueta:
PROC etiqueta()
```

- 'unsafe use of "*" or "/"'
  'uso de "*" o "/" no seguro'
  Esto tiene que ver de nuevo con * y / de 16 bits en lugar de 32.
  Mira 'división y multiplicación sólo con 16 bit'.

- 'reading sourcefile didn't succeed'
  'sin éxito al leer el fichero fuente'
  Comprueba la especificación de fichero fuente que ediste con 'ec mifuente', asegúrate de que el fichero acaba en '.e'

- 'writing executable didn't succeed'
  'sin exito al escribir el ejecutable'
  Al escribir el código generado como ejecutable produjo un error de DOS. Por ejemplo, el ejecutable que ya existía no se pudo reescribir.

- 'no args'
  'faltan argumentos'
  "USAGE: ec [-opts] <sourcecodefilename> ('.e' is added)"
  "USO: ec [-opts] <nombrefichfuente> (se añade '.e')"
  Obtienes esto simplemente con escribir 'ec' sin ningún argumento.

- 'unknown/illegal addressing mode'
  'modo de direccionamiento desconocido/ilegal'
  Este error sólo es del ensamblador en línea. Las posibles causas son:
  * usaste algún tipo de direccionamiento que no existe en el 68000
  * el modo de direccionamiento existe, pero no para esa instrucción. No todas las instrucciónes en ensamblador permiten todas las combinaciones de dirección efectiva tanto para origen como destino.

- 'unmatched parentheses'
  'discordancia de paréntesis'
  Tu sentencia tienen más "(" que ")" o la revés.

- 'double declaration'
  'declaración doble'
  Un identificador se usa en dos o más declaraciones.

- 'unknown identifier'
  'identificador desconocido'
  Un identificador no se usa en ninguna declaración; es desconocido. Probablemente te olvidaste de ponerlo en una sentencia DEF.

- 'incorrect #of args or use of ()'
  'número de args o uso de () incorrecto'
  * Te olvidaste poner "(" o ")" en el punto correcto,
  * pasate un número de argumentos incorrecto a alguna función.

- 'unknown e/library function'
  'función de E/biblioteca desconocida'
  Utilizas un identificador con el primer caracter en mayúscula y segundo en minúscula, pero el compilador no le encontró ninguna definición. Posibles causas:
  * Nombre de función mál escrito.
  * Se te olvidó incluir el módulo que define esa llamada.

- 'illegal function call'
  'llamada ilegal a función'
  Raramente aparece. Puede aparecer si intentas construir llamadas a extrañas, como WriteF()s anidados. Ejemplo: WriteF(WriteF('¡hola!'))

- 'unknown format code following "\"'
  'código de formato que sigue a "\" desconocido'

```
  Especificaste un código de formato ilegal en una cadena.

- '/* not properly nested comment structure */'
  '/* estrutura de comentarios incorrectamente anidada */'
  Número de '/*' no es igual al de '*/', o están en orden inapropiado.

- 'could not load binary'
  'no he podido leer binario'
  No se pudo leer <especfichero> en INCBIN <especfichero>.

- '"}" expected'
  'se esperaba "}"'
  Empezaste una expresión con "{<var>", pero se te olvidó el "}".

- 'immediate value expected'
  'se esperaba un valor inmediato'
  Algunas construcciones  requieren un  valor inmediato  en lugar  de una
  expresión.  Ejemplo:
         DEF s[x*y]:STRING  -> mal: sólo es legal algo como s[100]:STRING

- 'incorrect size of value'
  'tamaño de valor incorrecto'
  Especificaste un valor inaceptablemente  grande (o pequeño) para alguna
  construcción.  Ejemplos:
         DEF s[-1]:STRING, t[1000000]:STRING    /* debe ser 0..32000  */
         MOVEQ #1000,D2                         /* debe ser -128..127 */

- 'no e code allowed in assembly modus'
  'no se permite código E en modo ensamblador'
  Pusiste  el compilador  en modo  ensamblador con 'OPT  ASM',  pero  por
  accidente escribiste algo de código en E.

- 'illegal/inappropriate type'
  'tipo ilegal/no apropiado'
  En algún lugar donde se necesitaba una especificación <tipo> escribiste
  algo no apropiado.  Ejemplos:
         DEF a:PTR TO ARRAY       /* no hay tal tipo */
         [1,2,3]:STRING

- '"]" expected'
  'esperaba "]"'
  Empezaste con "[" pero nunca lo acabaste con "]"

- 'statement out of local/global scope'
  'sentencia fuera del alcance local/global'
  Un punto de  ruptura de alcance es la primera sentencia PROC.  Antes de
  ella sólo se permiten definiciones globales (DEF,CONST,MODULE,etc.), no
  código.  En la segunda parte sólo son legales código  y definiciones de
  funciones, no definiciones globales.

- 'could not read module correctly'
  'no se pudo leer el módulo correctamente'
  Ocurrió un error DOS intentando leer un módulo en una sentencia MODULE.
  Causas:   * EModules: no ha sido asignado correctamente.
            * Nombre del módulo escrito de forma incorrecta, o no existe.
            * Escribiste MODULE 'bla.m' en lugar de MODULE 'bla'.
```

- 'workspace full!'
  '¡espacio de trabajo lleno!'
  Raras veces aparece. Si lo hace, deberás usar la opción '-m' (ADDBUF) para forzar a EC a que haga una estimación superior de la cantidad de memoria que necesita. Intenta compilar con -m2, luego con -m3, ... hasta que desaparezca el error. Es probable que hayas escrito una aplicación enorme con cantidades de datos gigantes antes de poder tener la oportunidad de obtener este error.

- 'not enough memory while (re-)allocating'
  'memoria no suficiente al (re-)reservar'
  Simplemente eso. Posibles soluciones:
  1. Estabas corriendo otros programas en multitarea. Abandónalos e intentalo de nuevo.
  2. Tenías, de todas formas, poca memoria y tu memoria estaba fragmentada. Prueba reiniciando el sistema.
  3. Ninguna de 1-2. Compra una expansión de memoria (ejem!).

- 'incorrect object definition'
  'definción de objeto incorrecta'
  Escribiste alguna torpeza en las definiciones entre OBJECT y ENDOBJECT.

- 'illegal use of/reference to object'
  'uso ilegal de/referencia a objeto'
  Si usas expresiones como ptr.miembro, miembro debe ser un miembro legal del objeto al que apunta ptr.

- 'incomplete if-then-else expression'
  'expresión IF-THEN-ELSE incompleta'
  Si usas IF como operador la parte ELSE es imprescindible: una expresión con un IF en ella siempre necesita retornar un valor, mientras que una sentencia con un IF en ella puede, simplemente, no hacer nada si no está presente una parte ELSE.

- 'unknown object identifier'
  'identificador de objeto desconocido'
  Usaste un identificador reconocido por el compilador como parte de algún objeto, pero se te olvidó declararlo. Motivos:
  * nombre mal escrito
  * falta el módulo
  * el identificador del módulo no se escribe como esperas de los RKRMs. Compruebalo con ShowModule. Tienes que tener en cuenta que el sistema de objetos del Amiga se hereda de los identificadores del ensam -blador, no de C. Y además siguen la sintaxis de E.

- 'double declaration of object identifier'
  'redeclaración de identificador de objeto'
  Se usa un mismo identificador en la definición de dos objetos.

- 'reference(s) out of 32k range: switch to LARGE model'
  'referencia(s) fuera del rango de 32k: cambia a modelo LARGE'
  Sólo tienes que poner 'OPT LARGE' en tu código fuente y seguir.

- 'reference(s) out of 256 byte range'
  'referencia(s) fuera del rango de 256 bytes'
  Es probable que hayas escrito un BRA.S o Bcc.S a gran distancia.

- 'too sizy expression'
  'expresión muy grande'
  Usaste cadenas '' o listas [], quizas [[]] recursivas, de gran tamaño.

- 'incomplete exception handler definition'
  'definición de manejador de excepciones incompleto'
  Probablemente usaste EXCEPT sin HANDLE, o al revés.

- 'not allowed in a module'
  'no permitido en un módulo'
  Estas haciendo una de las pocas cosas que no se pueden hacer en un módulo, como son variables globales con inicialización.

- 'allowed in modules only'
  'sólo se permite en módulos'
  Es probable que hayas usado EXPORT en tu código fuente principal.

- 'this doesn't make sense'
  'esto no tiene sentido'
  Error general.

- 'you need a newer version of EC for this :-)'
  'necesitas una versión más reciente de EC para esto :-)'
  Es probable que estes usando un módulo que fue compilado con una versión más reciente de la que tienes.

- 'no matching "["'
  '"[" sin emparejar'
  Se encontró un "]" en una sentencia sin un "[" que lo empareja.

- 'this instruction needs a better CPU/FPU (see OPT)'
  'esta instrucción necesita una CPU/FPU mejor (mira OPT)'
  Usas una construcción (probablemente una instucción ensamb.) que requiere un OPT 020 o parecido.

- 'object doesn't understand this method'
  'el objeto no entiende este método'
  Invocaste un método que no fué definido para ese objeto.

- 'method doesn't have same #of args as method of baseclass'
  'el método no tiene el mismo número de args que el de la clase base'
  Si redefines un método, debes asegurarte de que el nuevo tienen el mismo número de argumentos que el original.

- 'too many register variables in this function'
  'demasiadas variables registro en esta función'
  Estas declarando variables registro con :REG, y de momento no se pueden usar más de 5.

- 'Linker can't find all symbols'
  'el enlazador no encuentra todos los símbolos'
  Si usas un módulo A que usa a su vez un módulo B, B también debe ser enlazado. A depende de ciertos PROCs que estan en B, y si B fue recompilado sin esos PROCs, entonces el enlazador tendrá problemas al crear tu ejecutable.

- 'could not open "mathieeesingbas.library"'

```
  'no pude abrir "mathieeesingbas.library"'
  Si usas código con reales, el propio compilador necesitará funciones de
  reales para poder generar el código.

- 'illegal destructor definition'
  'definición ilegal del desctructor'
  Definiste un método end() con argumentos (o con valor de retorno).

- 'implicit initialisation of private members'
  'inicialización implícita de miembros privados'
  Escribiste una expresión [...] o NEW [...] que tiene partes privadas.

- 'double method declaration'
  'redefinición de método'
  Definiste un método dos veces para el mismo objeto.

- 'object referenced by other object not found'
  'objeto hace referencia a otro que no se encuentra'
  Es probable  que heredaras de  algún otro objeto  en otro módulo,  y lo
  cambiaste  (por ejemplo  su  nombre) sin  cambiar/recompilar los  demás
  módulos que dependen de él también.

- 'unknown preprocessor keyword'
  'palabra clave de preprocesador desconocida'
  El PP de EC sólo reconoce #define, #ifdef, #ifndef y #endif.

- 'illegal macro definition'
  'definición de macro ilegal'
  Provocaste un error de sintaxis al escribir tu #define.

- 'incoherent #ifdef/#ifndef nesting'
  'anidamiento de #ifdef/#ifndef no coherente'
  Se te olvidó cerrar con #endif, o algo parecido.

- 'macro redefinition'
  'redefinición de macro'
  No puedes usar el mismo identificador de nombre de macro dos veces.

- 'Sintaxis error in #ifdef/#ifndef/#else/#endif'
  'error de sintaxis en #ifdef/#ifndef/#else/#endif'
  (mira correcta de compilación de código condicional).

- 'macro(s) nested too deep'
  'macro(s) anidada(s) a un nivel muy profundo'
  Obtendrás esto si tienes macros que  se expanden a otras, las cuales se
  expanden a otras, ...,  de forma que  la cantidad de  memoria necesaria
  para esto se empieza a desmadrar.  Lo más seguro es  que hayas definido
  una macro recursiva (que intentará expandirse indefinidamente).

- 'method definition out of object/module scope'
  'defínción de método fuera del alcance del objeto/módulo'
  Sólo puedes definir métodos para un objeto en el mismo módulo/fuente en
  el que se define el objeto.
```

1.145 ch_16f

Organización del buffer del compilador y reserva.

Es útil saber cómo organiza EC sus buffers cuando obtienes un error de 'workspace full' (raras veces), o quieres saber qué es lo que sucede realmente cuando se compila tu programa.

Un compilador, y en este caso EC, necesita buffers para mantener todo tipo de cosas, como identificadores, etc..., y necesita un buffer para almacenar el código que genera. EC desconoce el tamaño que deben tener esos buffers. Esto no representa ningún problema para la mayoría de los buffers, como el que se utiliza para varias estructuras, si el buffer se llena durante la compilación, EC sólo tiene que reservar un nuevo fragmento de memoria y continuar. Sin embargo, otros buffers, como el que contiene el código generado, debe estar en un bloque de memoria contíguo que no cambien durante la compilación: EC necista hacer una buena estimación sobre el tamaño de este buffer para poder compilar código fuente ya sea grande o pequeño. Para ello EC calcula la memoria que necesitará dependiendo del tamaño del código fuente, y le suma una cantidad apropiada. De esta forma, en el 99% de los casos, EC habrá reservado suficiente memoria para compilar practicamente cualquier código fuente, en los demás casos, se te ofrecerá un error y tendrás que especificar más memoria con la opción '-m' (ADDBUF).

Para ver cómo funciona esto en la práctica prueba con fuentes de diferentes tipos y tamaños en combinación con la opción '-b' (SHOWBUF).

1.146 ch_16g

Reserva de registros.

E v3 permite la reserva de registros, que es una técnica para mantener variables en registros en lugar de la pila. En código normal que use rutinas del SO no notarás mucho la diferencia, aunque para bucles de computación pequeños, esta optimización puede marcar una gran diferencia.

Hay dos formas de utilizar reserva de registros:

- Con la opción REG.
  Si, por ejemplo, escribes EC REG=3 bla.e, (de momento max=5), EC buscará para cada PROC las tres variables más usadas y las pondrá en registros. La resreva de registro es una técnica que intenta ser inteligente: calculará un peso para cada variable, y usará heurísticos para incrementar tal peso, por ejemplo, una variable usada en un bluche FOR obtiene un peso relativamente mayor que una usada fuera de él, y una en un IF obtendrá un peso incluso menor. Estos pesos se combinan, de forma que un WHILE en un FOR obtiene un peso bastante mayor.

- HTM (Hazlo Tu Mismo).
  Puedes poner la palabra clave REG delante de cualquier tipo en una declaración, por ejemplo:

```
      DEF x:REG, s[4]:REG LIST
```

puedes utilizarlo si no crees en el reservador de registros, o si quieres realizar un ajuste fino en un sólo PROC. Incluso puedes usar las dos técnicas juntas: si en un PROC tienes una var con ':REG', compilar con REG=5 permitirá a EC repartir los 4 restantes por sí mismo.

Por omisión REG=0, haciendo que EC opere como en versiones anteriores.

Sólo se PUEDEN reservar variables locales que no son parámetros. Ademas, si tomas la dirección de una variable con {}, no la puedes poner en un registro (adivina por qué). No se pueden reservar registros en PROCs que tienen manejador de excepciones, de momento.

Estas son algunas cosas a tener en cuenta al usar registros:

- Esta parte de EC (E v3.0a) se ha comprobado ser bastante fiable, pero debes seguir comprobando que el funcionamiento sea igual que en código sin reserva.
  Debe funcionar bien, pero es demasiado pronto para garantizarlo :-)
  Resumiendo: ten cuidado desde ahora al aplicar estas técnicas.
- EC usa los registros D7..D3 para variables, de modo que si usas ensamblador en línea, debes procurar que los PROCs que usan reserva de registros o :REG no los ensucien. El código generado por un PROC guarda de forma automática los registros que usa para proteger el código desde el que fué llamado.
- Pista: compilar con REG=5 no tiene porqué ser más rápido, ya que el guardar las variables en llamadas a función/librería implica más código a ejecutar. Además, si _todo_ el código en cuestión opera con llamadas de librería en lugar de computación pura, no esperes ninguna ventaja con los registros.

```
      PROC main()                        -> este programa será dos veces
         DEF a,b=10,c=20,d               -> más rápido fácilmente si se
         FOR a:=1 TO 1000000 DO d:=b+c -> reservan registros.
      ENDPROC

      PROC main()                       -> este será como mucho un 5% más
         DEF a,s[100]:STRING,t          -> rápido reservando registros.
         t:='poner "a" en un reg no creo que acelere esto mucho.'
         FOR a:=1 TO 100000 DO StrCopy(s,t)
      ENDPROC
```

1.147 ch_17

```
Utilidades imprescindibles para E.
----------------------------------

        ShowModule
        ShowHunk
        Pragma2Module/Iconvert
        ShowCache/FlushCache
        ecompile.rexx
        o2m
        E-Yacc
        SrcGen
        EBuild
```

```
        EE/Aprof
        EDBG
        Prepocesador de EC
```

1.148 ch_17a

```
bin/showmodule.
---------------
```

Como ya habrás notado, el equivalente de E para los "includes", los módulos, son ficheros binarios, muy parecidos a los que se ofrecen con los compiladores de Modula2 por ejemplo. Para ver el contenido de tales ficheros en forma ASCII legible, puedes utilizar ShowModule:

```
           ShowModule <especmodulo>

Ejemplos:      1> showmodule EModules:intuition/intuition
               1> showmodule >gadtools.txt EModules:gadtools
```

Fíjate que por omisión ShowModule escribe a stdout, y se puede interrumpir en cualquier momento con <CtrlC>.

1.149 ch_17b

```
sources/utilities/showhunk.e, bin/showhunk.
-------------------------------------------
```

Muestra todo tipo de ficheros ejecutables, además de ficheros objeto ".o" generados por (otros) compiladores/ensambladores. Muestra la estructura (muy simple) de los ejecutables generados por EC, aunque tambien permite ficheros overlay complejos. También muestra las etiquetas (como XREFs y XDEFs).

Y lo más importante de todo, ShowHunk ofrece un desensamblador para los bloques de código. Usa la opción 'DISASM/S'.

```
ShowHunk <ficheroeje>

Ejemplos:      1> ShowHunk holamundo
               1> ShowHunk DISASM dpaint
```

1.150 ch_17c

```
bin/iconvert, bin/pragma2module.
--------------------------------
```

Estas dos utilidades son sólo para programadores de E avanzados. Sáltate este apartado si crees que (todavía) no eres uno de ellos.

[NOTA: al igual que la utilidad ShowModule, ya no se incluye el código fuente de estas utilidades en la distribución, ya que la gente utilizaba

de forma incorrecta su conocimiento del formato de los módulos '.m'. Es PRIVADO. Contacta conmigo primero si quieres hacer algo con él.]

IConvert convierte definiciones de estructuras y constantes de los ficheros ".i" en ensamblador a módulos E, y Pragma2Module hace lo mismo con ficheros pragma de SAS/C de definición de librería. Por supuesto, ya se han convertido de esta forma todos los includes de Commodore, pero digamos que encuentras una librería de PD que te gustaría usar con E, entonces necesitarás estas utilidades.

La mayoría de las librerías vienen con varios includes definiendo, obviamente, las llamadas a librería de la librería, así como constantes y estructuras (OBJECTos en E) que utiliza. Digamos que se llama "tools.library", entonces, es probable que incluya:

```
pragmas/tools_pragmas.h
includes/tools.i
```

entonces tienes que hacer:

```
1> Pragma2Module tools_pragmas.h
```

renombra el "tools_pragmas.m" resultante a "tools.m" y ponlo en EModules:, comprueba con ShowModule si todo fue correcto.

Ahora, puedes usar la tools.library en tu programa:

```
MODULE 'tools'

PROC main()
  IF (toolsbase:=Openlibrary('tools.library',37))=NIL THEN error()
....
  ToolsFunc()
....
```

Convierte "tools.i" con IConvert a otro "tools.m", el cual puedes colocar en EModules/libraries/, por ejemplo. IConvert necesita un ensamblador como el A68k de DP para hacer la mayor parte del trabajo de traducción del ensamblador.

```
1> IConvert tools.i
```

mira con ShowModule todo que se obtuvo del fichero ".i". Y lo podrás usar en tu programa con:

```
MODULE 'libraries/tools'

DEF x:toolsobj, y=TOOLS_CONST
```

La conversión con IConver puede requerir alguna experiencia con ensamblador, ya que IConvert depende del formato correcto del fichero ".i", justo como los includes de ensamblador de Commodore. Es necesario corregir mano cerca del 10% de los ficheros ".i" para ser "convertibles". Las expresiones que IConvert piensa que son correctas son, entre otras:

```
<etiqueta> EQU <expresión_cualquiera>

STRUCTURE <nombrees>,0  ; si <>0, entonces <estructura>_SIZEOF
ULONG <nombrees>_<etiqueta>
BPTR <nombrees>_<etiqueta>
   ; etc.
LABEL <nombrees>_SIZEOF  ; o "_SIZE"
```

Para tener una idea del tipo de expresiones en ensamblador con las que puede operar IConvert, hecha un vistazo a los includes ensamblador de

Commodore y compáralos con sus módulos equivalentes (ej. intuition.i).

1.151 ch_17d

```
bin/ShowCache, bin/FlushCache.
------------------------------
```

El Caché de Modulos de E es un fragmento de memoria que permite mantener módulos (.m) entre compilaciones. La primera vez que usas un cierto módulo, EC lo leerá del disco y lo pondrá en el caché. Las siguientes veces EC lo encontrará en el caché, y no tendrá que leer nada de disco. Si EC compila un módulo del cual hay una versión anterior en el caché lo sacará del caché. Puedes imaginar que esto supone una enorme mejora en el tiempo de compilación cuando se usan gran cantidad de módulos y se recompila a menudo, incluso para gente con disco duro,

Para ver qué es lo que está almacenado en el caché en un momento determinado (y cuanta memoria se está empleando :-), teclea:

```
1> ShowCache
```

Una segunda utilidad, FlushCache, permite eliminar selectivamente los módulos del caché. Las razones de esto pueden ser:

- No te puedes permitir prescindir de la memoria que utiliza,
- Creaste un nuevo ".m", con una herramienta distinta de EC, por lo que nececitas vaciar el caché a mano.

El argumento que recibe es la cadena que debe aparacer en el nombre de un módulo para eliminarlo del caché. El no dar un argumento implica el vaciado completo.

```
1> FlushCache            ; vacía todo el caché
1> FlushCache intuition/ ; elimina módulos de intuition
```

Puedes usar la opción de EC 'IGNORECACHE/S' para compilar un fuente sin usar el caché. Ya esté el caché lleno o vacío EC lo leerá todo de disco y no pondrá nada en éste.

Si dos ECs intentan acceder al caché simultáneamente en multitarea, el segundo EC actuará como si se le hubiera dado la opción IGNORECACHE.

1.152 ch_17e

```
rexx/ecompile.rexx.
-------------------
```

[¿quien mantiene los otros scripts de ARexx?]

Este es un script ARexx para CygnusEd (tm), y te permite compilar programas desde el editor. Simplemente asigna este script a una tecla de función en el editor con "Install Dos/Arexx Command ..." (comprueba tu manual de CED si no estas seguro de como hacer esto). Ahora, escribe tu programa, y pulsa Fx si quieres compilar. El código fuente se guardará si es necesario, se invocará al compilador en una ventana consola separada, y se ejecutará el programa en una ventana consola diferente.

Cuando el programa acabe, puedes pulsar <return> para volver al editor (el script realiza de forma automática el cambio de la pantalla de CED adelante y detrás). Si ocurre un error durante la compilación, el script permitirá a CED saltar a la línea del error después de pulsar <return>.

Nota: en el script hay un nombre de camino que indica donde se puede encontrar el compilador. Es probable que tengas que cambiarlo. Además, el script copia EC a RAM: para sistemas con un dispositivo SYS: lento, puede que quieras deshabilitar esto si tienes un disco duro rápido.

1.153 ch_17f

bin/o2m.

Si tienes bloques de fuente en ensamblador grandes que te gustaría usar, sería tedioso tener que, como poco, convertirlos todos a mano a ensamblador en línea de E. o2m te permite hacer, simplemente, que tu macro-ensamblador favorito te lo ensamble todo a un fichero ".o", y entonces o2m te convertirá ese fichero ".o" a un fichero ".m" para ser usado con E. Si tienes un fichero bla.o:

```
1> o2m bla
```

producirá bla.m. Sin embargo, el fichero ".o" tendrá que seguir ciertas reglas. Debe consistir de un sólo bloque de código con definiciones externas (XDEFs) para cada símbolo al que desees hacer referencia desde E, y ningún XREF. Por lo general, tu fuente se parecerá a:

```
                XDEF suma__ii

        suma__ii:
                move.l  4(a7),d0
                add.l   8(a7),d0
                rts
```

en este caso es un fragmento de ensamblador que recibe dos argumentos (de ahí las dos "i" de entero). Los argumentos pueden encontrarse en la pila, donde 4(a7) es el último arg, 8(a7) el anterior, ...

ShowHunk muestra esto como:

```
                hunk_unit:
        HUNK -1 hunk_name:
                hunk_code: 12 bytes
                hunk_ext
                  add__ii = $0
```

este tipo de fichero ".o" se puede convertir facilmente a ".m" con o2m:

```
        /* this module contains 12 bytes of code! */

        PROC add(a,b)
```

Hay que destacar algunas cosas:

- si tu código ensam. usa D3-D7/A4/A5 es probable que debas guardarlos.
- si una etiqueta no tienen el "__" con un "i" para cada función, se convierte en una función sin parámetros. No te preocupes de si la etiqueta en realidad hace referencia a datos, puedes obtener la dirección de ese 'PROC' con {}, y usarla como un puntero a tus datos.

En teoría, o2m se podría usar para enlazar código C a programas en E,

sin embargo, a menudo, en la práctica esto no es factible. Si tu compilador de C te permite 'ajustar' los ficheros ".o" resultantes un poco, podría funcionar.

Algunos problemas son:
- referencia a funciones de C, por ejemplo _printf().
- referencia a variables globales creadas por código inicialización de C. El código en C podría hacer referencia a "DOSBase" como XREF, mientras que el código inicialización de E dispone este valor en algún lugar de la pila.
- convenios de llamada/registros.

He logrado enlazar con E pequeñas funciones de C que sólo realizan algunos cálculos simples, además de lograr utilizarlas con éxito (usando MaxonC++, cuyo enlazador usa el convenio __ii para los parámetros).

1.154 ch_17g

bin/EYacc.

Esta es una conversión de la famosa utilidad de Unix Yacc, el cual produce ahora código E en lugar de código C. Esta es sólo una primera versión, por lo que no esperes mucho de ella. Si no tienes idea de lo que hace Yacc, lee algún texto sobre ello (yo no voy a explicarlo aquí).

Básicamente, puedes escribir ficheros ".y" como siempre, sólo que donde las acciones se solían escribir en C, ahora puedes escribirlas en E. Mira el ejemplo Src/Yacc/bcalc.y.

```
1> eyacc bcalc.y
```
produce un fichero 'yyparse.e'
```
1> ec yyparse
```
genera un módulo, el cual sólo contiene la función yyparse(). El resto del 'como mediar con Yacc' debe ser análogo al C.

[nota: tengo a medias una conversión de Lex a E-Lex, aunque no acabada.]

Más información.
E-Yacc es un modificación del Yacc 1.8 de Berkeley, originario de corbett@berkeley.edu. La inclusión de esta versión modificada en la distribución de E es totalmente legal, ya que el autor dice en el README original de BYacc1.8:

" Berkeley Yacc is in the public domain. The data structures and algorithms used in Berkeley Yacc are all either taken from documents available to the general public or are inventions of the author. Anyone may freely distribute source or binary forms of Berkeley Yacc whether unchanged or modified. Distributers may charge whatever fees they can obtain for Berkeley Yacc. Programs generated by Berkeley Yacc may be distributed freely. "
[No veo interés en realizar la traducción de esto, si no piensas igual, dímelo: u0868551@oboe.etsiig.uniovi.es Antonio J. Gomez Glez.]

1.155 ch_17h

```
bin/SrcGen.
-----------
```

[nota: esta utilidad no ha sido actualizada para funcionar mejor con el sistema de módulos de E, todavía genera el código fuente llano (el cual se incorpora facilmente en cualquier módulo). Sólo la actualizaré si hay mucha demanda ello. Si quieres incluir un GUI decente en tu aplicación echa un vistazo también a modules/tools/EasyGUI.m, o a kits más recientes como BGUI de Jan van den Baard, el autor de GadToolBox.]

SrcGen, generador de fuente GadToolBox de para E: versión beta

Necesitarás GadToolBox v2.0 o superior, y tener la gadtoolsbox.library que viene con él en tu LIBS:. Ahora, cre con GTB algún ejemplo simple (una ventana con unos pocos gadgets/menus etc.), grábalo como "bla" (el nombre del fichero será "bla.gui"), y teclea:

```
1> SrcGen bla
1> EC bla
1> bla
```

"bla.e" tendrá las rutinas de apertura de tu interface, así como algunas rutinas para manejar mensajes IDCMP, errores, ..., y un "main" sencillo que sólamente espera por una selección. Ahí puedes poner tu propio código. Mira el patrón de la línea de comandos para ver como se evita que SrcGen genere esas rutinas.

Eso es para todo lo que sirve. Si tienes problemas, simplemente comprueba el código que se generó.

1.156 ch_17i

```
bin/EBuild.
-----------
```

EBuild es una copia de "Make", y funciona de forma similar. EBuild es una utilidad que te ayuda a recompilar las partes necesarias de una aplicación grande después de modificarla. Escribes un fichero ".build" en el directorio que contiene los fuentes de tu proyecto. El fichero contiene información sobre qué fuentes dependen de qué otros, y qué acciones se deben llevar a cabo si es necesario reconstruir un módulo o un ejecutable. EBuild comprueba las fechas de los ficheros para ver si se ha modificado un fuente después de la útima compilación, y si el fuente usa módulos que también han sido modificados, compilando esos últimos primero.

La sintaxis es la misma que el make de Unix. En resumen, "#" precede líneas con comentarios, y:

```
objetivo: dep1 dep2 ...
  acción1
  acción2
  ...
```

objetivo es el fichero resultado del que estamos hablando, en la mayoría de los casos un ejecutable o un módulo, aunque puede ser cualquier cosa.

Tras los ":" puedes escribir todos los ficheros de los que depende, probablemente su fuente, y otros módulos. Las acciones de las líneas que le siguen son comandos de AmigaDos normales, y es necesario precederlos de al menos un espacio o un tabulador para distinguirlas de objetivos.

```
bla: bla.e defs.m
     ec bla quiet
```

este sencillo ejemplo recompilará "bla.e" si ha sido modificado, o si el defs.m que usa ha sido modificado.

Si tecleas 'EBuild' sin argumentos, EBuild se asegurará de que el primer objetivo esté actualizado.

TARGET,FROM/K,FORCE/S:

Si das un TARGET (objetivo), EBuild empezará con otro objetivo. FROM te permite usar otro fichero que no sea ".build", y FORCE reconstruirá todo, independientemente de si era necesario o no.

Ejemplo: # fichero build de prueba

```
all:    bla burp
defs.m: defs.e
        ec defs quiet
bla:    bla.e defs.m
        ec bla quiet
burp:   burp.e
        ec burp quiet
limpia:
        delete defs.m bla burp
```

este fichero build es sobre dos programas, bla y burp, de los cuales bla también depende de un módulo defs.m. Incorpora un objetivo extra no real 'limpia' de forma que puedas escribir 'EBuild limpia' para borrar todos los ficheros generados.

Se pueden añadir otras dependencias y acciones con facilidad. Por ejemplo, si tu proyecto usa un analizador generado por E-Yacc:

```
yyparse.m: parser.y
           eyacc parser.y
           ec yyparse quiet
```

O si incorpora código de macro-ensamblador como módulo de herramienta usado a menudo:

```
blerk.m: blerk.s
         a68k blerk.s
         o2m blerk
         copy blerk.m emodules:tools
         flushcache tools/blerk
```

En el momento en el que logres conocer EBuild, descubrirás que lo puedes usar para más propósitos a parte de éste. Míralo como una herramienta de scripts inteligente.

Si quieres descubrir en detalle de lo que puede hacer EBuild, lee la documentación de algún make de Unix, ya que EBuild es, de alguna forma, compatible con él. Lo que de momento no hace es:

- eliminar dependencias cíclicas

```
        - permitir "\" al final de una línea para reglas más largas
        - definiciones constantes
```

Jason Hulance ha actualizado EBuild para la versión v3.1 del compilador de E, para corregir el error que ejecutaba acciones en orden inverso. También cambió la ejecución de la acción a un script (transparente). En ese script a la variable 'target' se le da el valor del objetivo actual.

```
Ejemplo:    test:   test.e
                    ec "$target"
                    if warn
                            echo "Error: falló la compilación"
                    else
                            echo "Compilado OK... ejecutando"
                            "$target"
                    endif
```

que equivale al siguiente código antiguo, aunque permite más cosas.

```
            all:    test
                    echo "ok, ejecutando:"
                    test

            test:   test.e
                    ec -q test
```

1.157 ch_17j

```
EE / Aprof
---------------
```

Estos se describen en su propia documentación en el directorio tools.

1.158 ch_17k

```
EDBG.
-----
```

EDBG es el depurador e nivel de fuente de E. Para usarlo compila tu fuente con la opción DEBUG (esto funciona tanto para el programa principal como para los módulos), lo cual añadirá información de depuración en tu ejecutable/módulo.

NOTA: NO distribuyas un programa en el que alguna parte ha sido compilada con DEBUG/S (puedes comprobarlo con ShowHunk, no debe contener ningún hunk_debug). Los programas con información de depuración se compilan con NOPs extra para facilitar la depuración, lo cual no es deseable en el código final.

Asegúrate siempre de que tanto el código fuente como el compilado están en el directorio actual, entonces lanza EDBG con:

```
            1> EDBG nombreejecutable

patrón:     EXECUTABLE/A,PUBSCREEN/K:
```

con PUBSCREEN puedes hacer que EDBG corra en cualquier lugar (por ej., PUBSCREEN=Workbench para que funcione en el Workbench). Por omisión EDBG abre su propia pantalla, la cual es una copia del Workbench en tamaño y modo de pantalla. EDBG funciona bien en la tarjeta gráfica Picasso, ...

EDBG abre una ventana para cada fuente en tu proyecto, y empezará con el que contiene a 'main'. Desde la que puedes recorrer tu código, y se abrirán ventanas automáticamente cuando sean necesarias. Cuando el código no tiene un fuente adjunto, sólo se puede ejecutar (lo cual es útil a veces, si no es necesario depurarlo).

Desde ese momento todo es bastante intuitivo. Los botones más importantes son las dos primeras imágenes, que son el de recorre sobre/en (step over/in). Recorre en sigue el código paso a paso según se ejecuta, recorre sobre hace lo mismo pero no entra en las subrutinas.
[Pruébalo, el depurador es bastante intuitivo].

Otras funciones sacan ventanas de memoria o registros, y algunas otras funciones (algunas sin implementar). Una importante es hacer un doble click en nombres de variables: ésto nos mostrará su contenido en la ventana temporal (ya lo sé, esto será algo más cómodo en el futuro).

PEGAS:
- El programa que se depura corre en la misma tarea que EDBG. Esto quiere decir que todo código que haga cosas especiales en la tarea deberá ser cuidadoso. Un ejemplo es Forbid(), etc...
- Un caso especial es ReadArgs(). Dado que EDBG ya leyó los argumentos, una llamada del programa que se depura causará una lectura de la consola. De forma que puedes teclear convenientemente los argumentos de tu programa en la línea de comandos y pulsar <return>.

NOTE: EDBG todavía es algo beta, aunque ya bastante útil. Le falta gran cantidad de funciones, y tendrás que esperar algo antes de que se lleguen a implementar. [por lo tanto, no vengas a advertirme que de "X no funciona" o "EDBG necesita X", ya que lo _se_.]

Las opciones LINEDEBUG y SYM.
La opción LINEDEBUG añade información 'linedebug' a tu ejecutable (para cada línea de código dentro de un PROC). EDBG necesita esta opción, aunque la opción DEBUG la activa de forma automática. LINEDEBUG es parcialmente compatible con el HUNK_DEBUG "LINE" producido por otros compiladores/ensambladores, por lo que también puede ser útil con otras herramientas de depuración. La opción SYM no es necesaria para EDBG, pero puede ser útil para otros, como AProf o desensambladores.

Errores más graves conocidos:
- no puedes desplazar la línea actual fuera de la parte visible en una ventana de fuente porque EDBG intenta mantenerla a la vista.
- muchos otros..., probablemente.

1.159 ch_17l

PreProcesador de EC.

EC tiene un peroporcesador interno que ofrece substituciones de macros y compilación condicional. Estas no son características del lenguaje E, sino que se han integrado en EC por velocidad y flexibilidad.

Activando el preprocesador.
Hasta que no escribas:

```
OPT PREPROCESS
```

EC se comportará como siempre. Esta OPT es necesaria para cualquier característica relacionada con el preprocesador.

Macros.
El macropreprocesador es compatible con Mac2E y el PP del C. Puedes definir macros con:

```
#define NOMBREMACRO
#define NOMBREMACRO CUERPO
#define NOMBREMACRO(ARG,...) CUERPO
#define NOMBREMACRO(ARG,...) CUERPO \
  RESTO DEL CUERPO
```

NOMBREMACRO y ARG pueden tener tanto mayúsculas como minúsculas, y pueden conter "_" y 0-9 como siempre. Se pueden añadir espacios en cualquier lugar menos entre NOMBREMACRO y "(", ya que sino EC no podría distinguir entre argumentos y el cuerpo.
El CUERPO puede contener cada uno de los argumentos tantas veces como quiera. Una macro puede continuar en la línea siguiente si se precede el fina_de_línea con un "\".
[un nombremacro sin cuerpo puede ser útil en combinación con
compilación condicional.]

Los identificadores de macro tienen precedencia sobre los demás.

Las macros definidas en un módulo sólo se guardan en el módulo si se activa OPT EXPORT (#define no se puede preceder con EXPORT). Si eso representa un problema, mantén las macros juntas en su propio módulo. Las macros de los módulos se pueden usar en código diferente simplemente con importarlas con MODULE, y usando OPT PREPROCESS.

Usando una macro.
El uso de NOMBREMACRO en cualquier punto del programa hará que se inserte el CUERPO de la macro en ese punto. Ten en cuenta que ésta es una substitución de texto, y tiene poco que ver con la sintaxis de E en sí. Si las macros tienen argumentos, éstos se insertarán en sus respectivos lugares del cuerpo de la macro. Si el cuerpo (o los argumentos) contienen más macros, éstos se expanderán más tarde.

Ejemplo:

```
#define MAX(x,y) (IF x>y THEN x ELSE y)
WriteF('el mayor = \d\n',MAX(10,a))
```

es lo mismo que escribir:

```
WriteF('el mayor = \d\n',(IF 10>a THEN 10 ELSE a))
```

Esto en seguida nos muestra el peligro de las macros: dado que simplemente copia textualmente, escribir MAX(computacion_grande(),1)

```
generará código  que ejecuta computación_grande() dos veces.  Ten cuidado
con eso.

Compilación condicional.
   Esto puede ser útil si quieres decidir en el momento de la compilación
qué parte de tu código quieres usar.  Sintaxis:
        #ifdef NOMBREMACRO
o
        #ifndef NOMBREMACRO
el fragmento de código  que lo sigue se, o no se compilará dependiendo de
si NOMBREMACRO fué definido.  Puedes hacer esto simplemente con:
        #define MIBANDERA
o algo parecido.  Finaliza el bloque compilado condicionalmente con:
        #endif

   Este tipo de bloques se puede anidar.
Ejemplo:    #define DEPURA
            ->#define DEPURABIEN

            #ifdef DEPURA
              WriteF('entrando en bla() con x = \d\n',x)
            #ifdef DEPURABIEN
              WriteF('volcando memoria...\n')
              /* ... */
            #endif
            #endif
```

1.160 ch_18

```
Apéndices.
----------

        Descripción de la gramática de E
        Tutorial
        Correspondencia de E con C/C++/Pascal/Ada/Lisp ...
        FAQ de Amiga E
```

1.161 ch_18a

```
Descripción de la gramática de E.
---------------------------------

   Esta es  la gramática de E  para aquellos a  quienes le  interese.  No
esperes que  esté actualizada o completa/correcta  (aunque  debería estar
bien para E hasta, por lo menos, v2.1b).

Sintaxis lex: expresiones regulares
Sintaxis analizador: adaptación de ASF/SDF propia;

        nombre   = identificador gramatical
        "nombre" = constante
        ()       = agrupamiento
```

```
        |        = o
        e*       = 0 o más de e
        e+       = 1 o más de e
        {e s}*   = 0 o más de e separados por s
        {e s}+   = 1 o más de e separados por s
        [e]      = e es opcional
        ; e      = e es comentario :-)


LEX
---

espacioblanco   = [ \t] ; también \n si último token es [,+-*/] o similar
                  cualquier cosa entre "/*" y "*/"
                  desde "->" hasta \n
eol             = [;\n]

constante       = [A-Z] ( [A-Z] [A-Za-z0-9_]* )?
interno         = [A-Z] [a-z] [A-Za-z0-9_]*
ident,identobj  = [a-z] [a-zA-Z0-9_]*

num             = [0-9]+        ; "-" es un token aparte
                  $[0-9A-Fa-f]+
                  %[01]+
numr            = [0-9]*.[0-9]*

cadenaconst     = cualquier cosa entre ''
charconst       = cualquier cosa entre ""


PARSE
-----

programa        = opcs parteglobal partelocal

parteglobal     = ( sentmodulo | sentdef | declobj | declconst | declexcep )*
partelocal      = ( declproc | declconst )+

sentmodulo      = "MODULE" { cadenaconst "," }+ eol
sentdef         = "DEF" listadeclvar eol
declobj         = "OBJECT" ident [ "OF" ident ] eol
                     ( listadeclvar eol )+
                  "ENDOBJECT" eol
declconst       = "CONST" { ( constante "=" expconst ) "," }+ |
                  "ENUM" { ( constante | constante "=" expconst ) "," }+ |
                  "SET" { constante "," }+
declproc        = [ "EXPORT" ] "PROC" ident "(" listadeclarg ")"
                  [ "OF" ident ] [ "HANDLE" ]
                  ( ( "RETURN" | "IS" ) { exp "," }* |
                     eol sentdef* sents
                     [ "EXCEPT" eol sents ]
                     "ENDPROC" { exp "," }* eol )
declexcep       = "RAISE" { ( constante "IF" interno "()" compop num ) "," }+
opcs            = ( "OPT" { setting "," }+ )*           ; depende máquina

listadeclvar    = { declvar "," }+
declvar         = ident [ "=" num ]
```

```
                    [ ":" ( "LONG" | "REAL" | "PTR" "TO" tipoptr ) ] |
                  ident ":" tipoobj |
                  ident "[" num "]" ":"
                    ( "ARRAY" |
                      "ARRAY" "OF" tipoptr |
                      "STRING" |
                      "LIST" )
listadeclarg    = { declarg "," }+
declarg         = ident [ "=" argomision ]
                    [ ":" ( "LONG" | "REAL" | "PTR" "TO" tipoptr ) ]
tipoptr         = tipoobj | tiposimple
tiposimple      = CHAR | INT | LONG
tipoobj         = ident

sents           = ( ( sentunalinea | sentmaslineas ) eol )*
sentunalinea    = exp |
                  vali ":=" exp |
                  { var "," }+ ":=" exp |
                  "IF" exp "THEN" sentunalinea "ELSE" sentunalinea |
                  "FOR" var ":=" exp "TO" exp [ "STEP" num ]
                     "DO" sentunalinea |
                  "WHILE" exp "DO" sentunalinea |
                  "RETURN" { exp "," }* |
                  "JUMP" ident |
                  ( "INC" | "DEC" ) var |            ; casi obsoleto
                  mnemonico_asm { operando "," }* |  ; depende máquina
                  "INCBIN" cadenaconst |             ; apoyo asm en línea
                  tiposimple { num "," }+ |
                  "VOID" exp                         ; obsoleto
sentmaslineas   = "IF" exp eol sents
                     [ ( "ELSEIF" exp eol sents )* ]
                     [ "ELSE" eol sents ]
                     "ENDIF" |
                  "FOR" var ":=" exp "TO" exp [ "STEP" num ] eol
                     sents "ENDPROC" |
                  "WHILE" exp eol sents "ENDWHILE" |
                  "REPEAT" eol sents "UNTIL" exp |
                  "SELECT" var eol
                     ( "CASE" exp eol sents )+
                     [ "DEFAULT" eol sents ]
                     "ENDSELECT" |
                  "LOOP" eol sents "ENDLOOP"

listaexp        = { exp "," }+
exp             = [ "-" ] { elem opbin }+ |
                  exp "BUT" exp
elem            = num | numr | vali | cadenaconst | charconst |
                  "SIZEOF" identobj |
                  "IF" exp "THEN" exp "ELSE" exp |
                  "[" listaexp "]" [ ":" tipoptr ] |
                  ( interno | ident ) "(" listaexp ")" |
                  var ":=" exp |
                  "{" ident "}" |
                  "`" exp |
opbin           = opmat | opcomp | oplog
opmath          = "+" | "-" | "*" | "/"
opcomp          = "=" | "<>" | ">" | "<" | ">=" | "<="
```

```
oplog           = "AND" | "OR"
expconst        = [ "-" ] { num ( "+" | "-" | "*" | "/" ) }+
vali            = var ( "[" [ exp ] "]" | "." ident )* [ "++" | "--" ] |
                  "^" var [ "++" | "--" ] |
var             = ident
argomision      = num
```

1.162 ch_18b

```
Tutorial.
---------

NOTA: el tutorial original de E v2.1b ya  no se incluye, ya que no era un
tutorial demasiado útil.  Es más, en su lugar, Jason Hulance  ha hecho un
tutorial de E intensivo, al que deberías hechar un vistazo.
[N.T.: está en inglés :-(]
```

1.163 ch_18c

```
Correspondencia de E con C/C++/Pascal/Ada/Lisp ...
--------------------------------------------------
[Se han añadido algunas cosas nuevas aquí]

   En las  dos primeras columnas  aparecerán las equivalencias entre E  y
AnsiC/C++, la  tercera columna  está reservada  para un  tercer lenguaje.
Usaré principalmente Pascal en esa  columna, aunque si una característica
lo pide, usaré otros (LISP, por ejemplo, con expresiones entrecomilladas,
Ada con excepciones, etc...)

apunta bien: no te tomes muy en serio estas tablas. Intentaré denotar las
equivalencias sintácticas,  y las  propiedades semánticas siempre que sea
posible, pero cada lenguaje necesita una evaluación propia.

equivalencia de signos y abreviaturas:

-       = característica no disponible en el lenguaje en cuestión.
?       = el autor no tiene idea a qué se traduce esa característica.
          (o por lo menos no está seguro).
...     = puede estar disponible, pero no en relación 1:1 que la haga
          interesante.
x,y,z   = indetificadores arbritarios.
e,f,g   = expresiones arbritarias.
s,t,u   = sentencias arbritarias.
i,j,k   = enteros arbritarios.
etc.


-------------------------------------------------------------------------
ESTRUCTURA/SENTENCIAS

E                       C/C++                   Pascal
----------------------- ----------------------- -----------------------
```

```
PROC x()                int x() {               FUNCTION x:INTEGER;
PROC x(y,z)             int x(y,z) {            FUNCTION x(y,z:INTEGER):INTEGER;
PROC x(y=1)             int x(y=1) {            -
ENDPROC                 return 0; };            x:=0; END;
ENDPROC e               return e; };            x:=e; END;
ENDPROC e,f,g           -                       -
RETURN e                return e;               ?

IF e                    if(e) {                 IF e THEN BEGIN
ELSEIF e                } else if(e) {          END ELSE IF e THEN BEGIN
ELSE                    } else {                END ELSE BEGIN
ENDIF                   };                      END;
IF e THEN s             if(e) s;                IF e THEN s;
IF e THEN s ELSE t      if(e) s else t;         IF e THEN s ELSE t;

FOR x:=e TO f           - (1)                   FOR x:=e TO f DO BEGIN
FOR x:=e TO f STEP i    -                       - (2)
EXIT e                  if(e) break;            -
ENDFOR                  -                       END;
FOR x:=e TO f DO s      -                       FOR x:=e TO f DO s;

WHILE e                 while(e) {              WHILE e DO BEGIN
EXIT e                  if(e) break;            -
ENDWHILE                };                      END;
WHILE e DO s            while(e) s;             WHILE e DO s;

s; WHILE e              for(s;e;u) {            s; WHILE e DO BEGIN
  t; u                    t;                      t; u
ENDWHILE                };                      END;

REPEAT                  do {                    REPEAT
UNTIL e                 } while(!e);            UNTIL e;

LOOP                    for(;;) {               WHILE TRUE DO BEGIN (?)
ENDLOOP                 };                      END;

SELECT x                switch(x) {             CASE x OF
SELECT x OF y           switch(x) {             CASE x OF
CASE 1; s...            case 1: s...; break     1: BEGIN s... END
CASE a+1                -                       -
CASE 1,2,3              case 1: case 2: case3:  1,2,3:
CASE "a".."z"           -                       -
ENDSELECT               };                      END

INC x                   x++;                    x:=x+1; (INC())
DEC x                   x--;                    x:=x-1; (DEC())
JUMP lab                goto lab;               GOTO lab;
x:=e                    x=e;                    x:=e;

/* */                   /* */                   { }
->                      //                      -
```

(1) mira WHILE; C no tiene FOR, "for" es otra forma de escribir "while"
(2) sólo STEP -1 con DOWNTO

VALORES

E	C/C++	Pascal
1	1	1
1.0	1.0	1.0
$1	0x1	?
%1	?	?
"a"	'a'	chr(97) (?)
'blabla'	"blabla"	'blabla'
[1,2,3]	- (1)	-
[1,2,3]:INT	-	-

(1) al traducir de E a C, puedes simularlo con:

```
            mifunc([1,2,3])
    se convierte en:
            int temp [] = {1,2,3};
            mifunc(temp);
```

OPERADORES

E	C/C++	Pascal
+ - * /	+ - * /	+ - * DIV
= <> > < >= <=	== != > < >= <=	= <> > < >= <=
AND OR (log)	&& ||	and or
AND OR (bit)	& |	?
SIZEOF x	sizeof(x)	-
`e	-	- (1)
^x (4)	*x	...
{x}	&x	...
x++	x++	-
x--	--x	-
-x	-x	-x
IF e THEN f ELSE g	e ? f : g	-
x.y	x->y	x^.y
-	x.y	x.y
x.y.z	x->y->z	x^.y^.z
x:=e	x=e	-
e BUT f	(e,f)	-
x[]	x[0] *x (2)	x[0]
x[1]	x[1]	x[1]
x[1] (3)	&x[1]	?
x[1].y	x[1]->y	x[1]^.y
x[]++	*x++	-
x[1].y++	*(x+1)++	-
x::y.a	((y *)x)->a	-
x.y::z.a	((z *)x->y)->a	-

(1) mira EXPRESIONES ENTRECOMILLADAS
(2) también para otros, se mantiene la equivalencia entre *(x+e) y x[e].
(3) con ARRAY OF <objeto>
(4) ONLY sólo para pasar por referencia. En otro caso: "[]"

CONSTANTES/TIPOS

```
E                       C/C++                   Pascal
----------------------- ----------------------- -----------------------
CONST X=1               #define X 1             CONST X=1;
                        const int X=1;
ENUM X,Y,Z              #define X 0 (etc.)      TYPE x=(X,Y,Z);
                        enum x{X,Y,Z};
SET X,Y,Z               -                       TYPE x=SET OF (X,Y,Z);

DEF                                             VAR
x                       int x; (or: long x;)    x:INTEGER;
x:LONG                  int x;                  x:INTEGER;
x:PTR TO y              struct y* x;            x:^y;
x:y                     struct y x;             x:y;
x[10]:ARRAY OF y        struct y x[10];         x:ARRAY [0..9] OF y;
x[10]:STRING            - (1)                   x:STRING[10]; (2)
x[10]:LIST              - (1)                   - (1)

x:REG                   register int x;

OBJECT x                struct x {      (3)     TYPE x = RECORD
  y:CHAR,z:INT            char y; short z;        y:CHAR; z:INTEGER;
ENDOBJECT               };                      END;
```

(1) al traducir de E a C, simúlalo con un arreglo de char/int respect., y realiza tu propia comprobación de rango, etc.
(2) no en Wirth Pascal, pero disponible en todos los dialectos populares.
(3) o clase pública.

EXPRESIONES ENTRECOMILLADAS

```
E                       LISP                        MIRANDA
----------------------- --------------------------- -------------------
`e                      (QUOTE e)  'e               (3)
                        (LAMBDA () e)     (1)
`x+y                    '(+ x y)
Eval(`e)                (EVAL `e)
ForAll(v,l,`e)          - (2)
MapList(v,l,l,`e)       (MAPCAR (LAMBDA (V) E) L)   map (\v->e) l
```

ejemplo:

```
E:              MapList({x},[1,2,3,4],a,`x*x)
MIRANDA:        map (\x->x*x) [1,2,3,4]
LISP:           (MAPCAR (LAMBDA (X) (* X X) `(1 2 3 4))
```

(1) realmente QUOTE, pero usado a veces donde en LISP se usaría LAMBDA, como en MapList()
(2) ni siquiera en ProLog, mira en otros lenguajes de lógica.
(3) en su lugar se usaría 'lazyness'

UNIFICACION Y CELDAS LISP

```
E                       LISP                    PROLOG
----------------------- ----------------------- -----------------------
<1|2>                   (1 . 2)                 [1|2]
<1,2,3>                 (1 2 3)                 [1,2,3]
<1,2|3>                 (1 2 . 3)               [1,2|3]

E                       HASKELL                 PROLOG
----------------------- ----------------------- -----------------------
e <=> <x|y>             (x:y) = e               e = [X|Y]
e <=> <1,2,x>           [1,2,x] = e             e = [1,2,X]
e <=> [1,x]             -                       -


-------------------------------------------------------------------------
EXCEPCIONES

E                       C++                     ADA
----------------------- ----------------------- -----------------------
PROC x() HANDLE         int x() { try {         function x is begin
EXCEPT                  } catch (exc) {   (1)   exception
EXCEPT DO               -                       -
ENDPROC                 }};                     end x;

Raise(e)                throw e;                raise e;
Throw(e,f)              ?                       -
ReThrow()               throw e;                raise e;
RAISE "MEM" IF New()=0  -                       - (2)


(1) catch sólo controla un excepción concreta, es bastante diferente de
    como se usan los manejadores de excepciones generales en E.
(2) el sistema en tiempo de ejecución no lanza algunas excepciones, pero
    no estoy seguro si se pueden _definir_ excepciones lanzadas
    automáticamente en ADA.


-------------------------------------------------------------------------
PROGRAMACION ORIENTADA A OBJETO

E                               C++
------------------------------- -----------------------
OBJECT x                        class x {
OBJECT x OF y                   class x : y {
self.i                          this->i
PROC a OF x IS self.i           virtual int x::a() { return i; }
-                               int x::a() { return i; }
PROC a OF x IS EMPTY            virtual int x::a() =0
PUBLIC                          public:
a.method(1)                     a->method(1)

[mira también en el siguiente apartado en NEW]

-------------------------------------------------------------------------
FUNCIONES PREDEFINIDAS Y RESERVA DE MEMORIA
(sólo algunas a forma de ejemplo)

E                       C/C++                   Pascal
```

WriteF(fs,...)	printf(fs,...); cout << a << b ... ;	WriteLn(a,b,...);
ReadStr(f,s) Val(s)	scanf(fs,...) cin >> s;	ReadLn(s) Val()
StrCopy(s,s,n) (1)	strcpy(s,s)	s:=s; (2)
Mod(e,e) Shl(e,n) Long(e)	e%e e<<n -	e MOD e Shl() -

p:=New(e)
NEW p
NEW p.constr()
NEW [e,f,g]
Dispose(p)
END p
	p=malloc(e);
	p=new type;
	p=new constr()
	-
	free(p);
	delete p;
		New(p);
		
		-
		-
		Dispose(p);

(1) al traducir de C, asegúrate de convertir los arreglos de char en STRINGs de verdad.
(2) no se que función se necesita en el caso de punteros.

1.164 ch_18d

```
FAQ de Amiga E.
---------------
[1_10_94..1_12_94]
```

Esta lista-FAQ (Frequently Asked Questions = Preguntas realizadas con Frecuencia) ha sido recogida mirando en el emil viejo y escongiendo aquellas preguntas que se repetían una y otra vez.

Compilador/Enlazador/Ejecutables

- ¿Cómo puedo enlazar código E con otros lenguajes/usar un formato de objeto estandard?

  La conversión entre .m y .o no es en realidad un gran problema, como podemos ver con la utilidad o2m, que permite la cooperación entre E y macro-ensambladores bastante bien. El problema con el C radica en el código compilado, no el formato del fichero. A no ser C trivial, cualquier código compilado en C hará referencia a cosas como _DOSBase, librerías de enlace, bases de pila, y otras cosas del código de inicialización que E proporciona de una forma algo diferente, por ejemplo, el código de inicialización de E es bastante diferente de el de los compiladores de C. Estos problemas pueden aparecer incluso entre dos compiladores de C. Si puedes conseguir que tu compilador de C genere un .o sin referencias externas, etc, entonces podrás enlazarlo con E (yo ya lo he hecho una vez con MaxonC++).

- ¿Puedo enlazar con amiga.lib?

De momento no. Aunque en menor medida, esto tiene los mismos problemas que los ficheros .o, aunque con la ayuda de un ensamblador y o2m se podría traducir un fichero .lib a .m, de hecho, ya se ha realizado para algunas. Sólo es cuestión de tiempo que alguien lo haga para (partes de) amiga.lib.

- ¿Se puede hacer residente el código de E?

  Practicamente TODO el código de E ya se puede hacer residente sin la ayuda del programador. Según creo, la única construcción E que puede violar ésto es un lista estática [] con una expresión en ella (como [a] por ejemplo. [1] y NEW [a] van bien).

- Tengo una aplicación con fragmentos de ensamblador en línea, y después de OPTI/S no funciona bien. ¿Qué está pasando?

  El ensamblador en línea puede usar los mismos registros a la vez que el opimizador de registros. Busca en otra parte del documento sobre eso.

- He escrito X tanto en C como en E, y la versión en E es Y veces más rápida/lenta. ¿A qué se debe?

  No es fácil hacer una comparación de forma totalmente objetiva. A menudo, se puede dar el caso que alguno de los dos tenga rutinas más optimizadas para algo (manejo de cadenas, E/S, etc..). El uso de un profiler puede ayudar a descubrirlo. Si el código en cuestión sólo realiza cálculos, no hay ninguna razón por la que alguno de los dos sea significativamente más lento que el otro, si se usan correctamente.

- He escrito un buen 'misutils.m', y con sólo utilizar una función de él, EC lo enlaza todo. Eso no me gusta.

  El estilo de escritura de módulos en E es mantenerlos como unidades relativamente pequeñas, sólo con código y datos relacionados. (Esto tiene más sentido si sabes que el módulo es la unidad de ocultamiento de datos de E). ¡Divídelo!

- ¿Puedo dejar EC residente?

  No, de momento no. El propio EC esta escrito en un estilo de ensmablador anticuado, y no se puede hacer residente ;-)

Recursos

- En la documentación no se explican la funciones de intuition, ...
  ¿Qué pasa?

  Esas funciones no son parte de E, sino que pertenecen al SO del Amiga. Como tales, se describen en los documentos de Commodore, no en los documentos de E. "The AMIGA ROM Kernel Reference Manuals" son una serie de libros publicados por Addison Wesley. Un buen lugar por donde empezar es el libro de "Libraries" de esa serie, ISBN 0-201-56774-1. También hay otros libros que merecen la pena ser considerados, como el "The Amiga Guru Book" de Ralph Babel.

- Me gustaría leer más código fuente a parte del que viene con la distribución, ¿Donde debo buscar?

  Hay cantidad de lugares en los que mirar, aunque el mejor sin duda alguna es Aminet (Una colección de sites de FTP en Internet), por ejemplo ftp.luth.se. En el directorio '/pub/aminet/dev/e' encontrarás todo tipo de cosas relacionadas con E. Algunas BBSs fuera de Internet también tienen Aminet, e incluso hay disponibles CDROMS. Otro buen lugar para recoger fuentes y hablar con buenos programadores en E, es la lista de correo de E. Envía un email con 'HELP' en el cuerpo a amigae-request@bkhouse.cts.com para escuchar todo sobre ello. También hay discusiones sobre E en otras redes (FidoNet, AimgaNet), series de discos de dominio público (EPD en europa/alemania, la misma dirección que el lugar de registro alemán), además, hay gran cantidad de clubs de usuarios, BBS que apoyan E. Simplemente echa un vistazo en tu zona.

- He oido hablar de esa lista de correo. ¿Qué tal está?

  Compruebalo tu mismo. Si vas en serio sobre E, es definitivamente imprescindible, además de que es el primer lugar en el que aparecen las noticias sobre E.

- He tenido problemas para introducirme en la lista. ¿Podrías ocuparte de eso por mi?

  Yo no tengo, personalmente, nada que ver con la administración de la lista, de forma que no te puedo ayudar más que cualquier otro. Recuerda que el servidor es un proceso automátizado, de forma que debes ser preciso en lo que le envias. No envíes correo administrativo a la lista, en su lugar, intenta contactar con el administrador.

- Soy nuevo en Internet. ¿Podrías ayudarme a conseguir la lista de correo, los ficheros FTP de Aminet, etc...?

  Cosas de este tipo van más allá de mi cometido. Por favor, intenta contactar localmente con alguien.

- ¿Cómo se cual es la última versión, y cómo la recibo?

  Si estas en la lista de correo serás el primero en saberlo. Aminet es el primer lugar en el que aparecen las distribuciones y actualizaciones. Los usuarios registrados obtendrán las actualizaciones y/o notas de forma automática en su mbox.

- ¿Cómo registrarse?

  Por favor mira en otra parte de este documento.

- ¿Existe E para otras plataformas?

  No todavía. Me he comprometido personalmente con un proyecto para realizar compiladores/traductores, y algunos otros tienen proyectos de compilador E para otras plataformas, aunque de momento aún no hay nada.

- ¿Puedo convertirme en un lugar de registro para el país X?

  Generalmente los lugares los elijo yo cuando creo que es necesario y

conozco muy bien a alguien en ese país.

- ¿Puedo apoyar E con una BBS / con un club de programación ...?

Siempre puedes hacerlo, sin falta de preguntarme. Aunque por supuesto, me gusta tener noticias sobre esfuerzos de ese tipo...

Programación

- ¿Puedo hacer X en E? (donde X = {juegos,autoedición,...})

E es un lenguaje de programación de propósito general, por lo que no debería haber ningún tipo de programa que no se pueda hacer con E (con alguna rara excepción, que está cubierta con el ensamblador en línea de E). Esto no significa que E esté equipado especialmente para ciertos tipos de programas, es decir, E no tiene funciones especiales para juegos (aunque su extensibilidad permite incorporarlas facilmente).

- ¿Cómo creo X en E? (donde X = {ventana, manejador de interrup.,...})

Al igual que en la pregunta anterior, E sólo abre la posibilidad de escribir lo que quieras, y no siempre tiene una función específica. En el peor de los casos eso significará el usar funciones difíciles de alguna librería, aunque merece la pena meterte en detalle con eso. Si tienes suerte algún otro programador ya habrá realizado algo parecido de lo que puedas aprender.

- Por favor ecríbeme un ejemplo de como hacer X en E.

Por favor, intente imaginartelo tú mismo, o pregunta a otros programadores de E que te ayuden.

- El compilador no quiere compilar cosas como mipantalla.rastport.bitmap, incluso cuando en general x.y.z es posible. ¿Por qué?

Para poder hacer referencia al último bitmap, el compilador necesita saber el tipo de la expresión mipantalla.rastport. Y si miras en el módulo 'intuition/screens.m' verás que el campo rastport no tiene tipo.

- ... ¿por qué no tenemos nuevos módulos con tipos corregidos?

Porque los includes en ensamblador (.o) de los cuales se convierten los módulos E no contienen esa información. No se pueden usar los ficheros .h de C en su lugar, porque usan de alguna forma otros identificadores, y rompería la compatibiladad anterior.

- ... entonces, ¿qué hago?

El método clásico es poner x.y en un puntero con tipo, y entoces hacer referencia a .z desde ése. Con la mutación de punteros en v3, también puedes hacer referencia a él directamente (míralo en otra parte de este documento).

- Escribo código como este:
```
DEF s[100]:STRING
s:='mi bonita cadena'
```
que el compilador permite, aunque me dá problemas más tarde. ¿Que puede

estar mál esto?

Si estas acostumbrado al BASIC, por ejemplo, estarás acostumbrado a usar las cadenas de la misma forma que los enteros, ya que ambos son _valores_. Sin embargo en E no hay variables cadena reales en el sentido del BASIC, el DEF anterior crea un trozo de memoria para almacenar la cadena, luego fija la variable como un puntero a esa memoria. El puntero y la memoria son, aparte de la implementación, dos entidades no relacionadas. A partir del DEF, todas las operaciones que acceden directamente a 's' están accediendo al puntero. La asignación pone la direccion de 'mi bonita cadena' en 's', sobreescribiendo el valor anterior. La memoria de cadena creada por el DEF se mantiene inalterada, y ahora inaccesible ya que no hay ningún puntero que apunte ella. Las funciones como StrCopy() pueden usar ese puntero para encontrar la memoria real, y rellenarla:

```
StrCopy(s,'mi bonita cadena')
```

es correcto. Por supuesto, la asiganción de cadenas como punteros también es de utilidad, por ejemplo, si sólo quieres leer los datos de la cadenas, y no hacer nada más con ellos, entonces:

```
DEF s:PTR TO CHAR
```

no reserva ninguna memoria para la cadena, sólo el puntero. La asignación anterior tendría sentido ahora. Todo esto nos lleva a la diferencia entre punteros y valores (o semántica por valor/referencia), y es importante que lo entiendas para programar en E con éxito.

- ¿Como retorno un STRING, OBJECT, etc. de una función?

Depende. Si sólamente quieres utilizarlo como valor de retorno temporal, pasa una cadena como argumento y deja que la rutina la rellene. Si necesitas crear una cadena/objeto nuevo, reserválo dinámicamente dentro de la función, y retorna el puntero.

<se añadirán más cuestiones específicas de este tipo>

Errores

- Recuerdo cuando estaba programando algo que de repente el compilador rompió/funcionó mal/produjo un error interno/generó mensajes de error erróneos. ¿No deberías hacer algo sobre esto?

Si no tengo pistas sobre donde buscar un error, no podré corregirlo. Si te encuentras con algo que estas seguro de que es un error del compilador, haz una copia del fuente que reproduce el error (si puedes reduce el fuente al mímino manteniendo el error) y envíamelo con tanta información como sea posible. Por ejemplo, los hits de Enforcer de EC son bastante útiles.

- He escrito un programa X, pero rompe. Estoy seguro de que no tiene errores, por lo que debe ser un error del compilador.

Debido a la falta de tipos de E, nunca puedes estar completamente seguro de que tu código es correcto. La mayor parte del código que se me envió con comentarios como el anterior se comprobó más tarde que eran errores en el código, casi siempre debidos a una falta de conocimiento de E. ¡Lee la documentación, y usa EDGB!

Caracteristicas Futuras

¿Tendrá E...
- argumentos en registros?

  Estoy seguro de que es posible, pero como muchas otras cosas, no es prioritario para mí.

- enlazador de librería/dispositvo (Library/Device)?

  Ya lo habría hecho si no fuera por la forma en que E guarda la variables globales (en la pila), lo cual complica las cosas bastante. Esto simplemente necesitará algo de tiempo.

- herencia múltiple?

  No es posible en E. La herencia múltiple rompe la compatibilidad estructural entre objetos y necesita de una comprobación de tipos bastante severa para resolverlo.

- PROCs dentro de PROCs?

  No creo.

- arreglos multidimensionales?

  Tampoco lo creo. Realmente en E no hay nada parecido a un arrreglo, sólo punteros que apuntan a grandes cantidades de objetos del mismo tamaño. Para tener arreglos de dos dimensiones, sería necesario tener el concepto de 'tamaño' de un arreglo, lo cual E no tiene.

- más aopoyo a 020/881?

  Si, eventualmente. Aunque no está entre las primeras cosas a realizar.

- sintaxis compatible con C?

  A menudo la gente está acostumbrada a algún tipo de lenguaje (principalmente C), y no entiende porque el diseño de un lenguaje de programación no es tan configurable como una utilidad de directorio o un editor de texto:

```
'Prefiero "=="/"=" más que "="/":="'
'¿No puedes intercambiar """ y "'" ?'
'¿Por qué "--" no funciona como en C?'
'¡Odio escribir las palabras clave en mayúsculas!'
'¡La falta de precedencia es odiosa!'
```

  Las características 'referidas' arriba jamás cambiarán, y será mejor que te acostumbres a ellas. Todas las decisiones de diseño tienen buenas razones, y al contrario de lo que dicen algunos rumores, no se hicieron por razones de velocidad del compilador (de hecho, el implementarlas de una forma tradicional no haría más rápido al compilador).

- ¿Por qué la caracteristica X en E no es como en el lenguaje Y? (que yo encuentro mejor).

  Lo mismo que antes, el diseño de un lenguaje no se puede cambiar debido

a una preferencia personal. Si tienes ideas serias de cómo debe ser un lenguaje según tus preferencias, y esas ideas provienen del lenguaje Y, entonces utiliza Y. Si no encuentras tus ideas en ningún lenguaje existente, entonces ¡es el momento de diseñar e implementar el tuyo propio! (No me estoy burlando, ¡merece la pena!, y siempre puedes empezar con E... :-)

- quizás X?

Después de haber visto prácticamente cientos de lenguajes de programación, las posibilidades que tienes de sugerir una nueva característica para E en la que no haya pensado, no son muy grandes. Muchas cosas no son apropiadas para E, y en general el tiempo está limitado para añadir cosas más interessante. (Si te fijas bien, la cantidad de características de E ya es bastante alta para un lenguaje de programación medio). Tengo una lista bastante grande de posibles características para E, y se implementarán eventualmente algunas de ellas. Sólo espera y veras...