Amiga E v3.1a
Compiler for The E Language
By Wouter van Oortmerssen

Contents:

0. compiler and introduction
   A. introduction
   B. the distribution
   C. demo restrictions, registration and sites
   D. using the compiler
   E. changes from v2.1b to v3.0a, and from v3.0a to v3.1a
   F. additional information

1. format
   A. tabs,lf etc.
   B. comments
   C. identifiers and types

2. immediate values
   A. decimal (1)
   B. hexadecimal ($1)
   C. binary (%1)
   D. float (1.0)
   E. character
   F. strings ('bla')
   G. lists ([1,2,3]) and typed lists
   H. lisp-cells (<a|b>)

3. expressions
   A. format
   B. precedence and grouping
   C. types of expressions
   D. function calls

4. operators
   A. math (+ - * /)
   B. comparison (= <> > < >= <=)
   C. logical and bitwise (AND OR)
   D. unary (SIZEOF ` ^ { } ++ -- -)
   E. triple (IF THEN ELSE)
   F. structure (.)
   G. array ([])
   H. float operator (!)
   I. assignments expressions (:=)
   J. sequencing (BUT)
   K. dynamic memory allocation (NEW)
   L. unification (<=>)
   M. pointer typing (::)

5. statements
   A. format (;)
   B. statement labels and gotos (JUMP)
   C. assignment (:=)
   D. assembly mnemonics
   E. conditional statement (IF)

F. for-statement (FOR)
G. while-statement (WHILE)
H. repeat-statement (REPEAT)
I. loop-statement (LOOP)
J. select-case-statement (SELECT)
K. increase statement (INC/DEC)
L. void expressions (VOID)
M. memory deallocation (END)

6. function definitions and declarations
A. proc definition and arguments (PROC)
B. local and global definitions: scope (DEF)
C. endproc/return
D. the 'main' function
E. built-in system variables
F. default arguments to functions
G. multiple return values
H. function values

7. declaration of constants
A. const (CONST)
B. enumerations (ENUM)
C. sets (SET)
D. built-in constants

8. types
A. about the 'type' system
B. the basic type (LONG/PTR)
C. the simple type (CHAR/INT/LONG)
D. the array type (ARRAY)
E. the complex type (STRING/LIST)
F. the compound type (OBJECT)
G. initialisation
H. the essentials of the E typesystem

9. built-in functions
A. io functions
B. strings and string functions
C. lists and list functions
D. intuition support functions
E. graphics support functions
F. system support functions
G. math and other functions
H. string and list linking functions
I. lisp-cells and cell functions

10. library functions and modules
A. built-in library calls
B. interfacing to the amiga system with the v39 modules
C. compiling own modules
D. the modulecache

11. quoted expressions
A. quoting and scope
B. Eval()
C. built-in functions

12. floating point support
A. float values
B. computing with floats
C. builtin float functions
D. float implementation issues

13. Exception handling
    A. defining exception handlers (HANDLE/EXCEPT)
    B. using the Raise() function
    C. defining exceptions for built-in functions (RAISE/IF)
    D. use of exception-ID's

14. OO programming
    A. OO features in E
    B. object inheritance
    C. data hiding (EXPORT/PRIVATE/PUBLIC)
    D. methods and virtual methods
    E. Constructors, Destructors and Super-Methods (NEW,END,SUPER)

15. inline assembly
    A. identifier sharing
    B. the inline assembler compared to a macro assembler
    C. ways using binary data (INCBIN/CHAR..)
    D. OPT ASM
    E. Inline asm and register variables

16. technical and implementation issues
    A. the OPT keyword
    B. small/large model
    C. stack organisation
    D. hardcoded limits
    E. error messages, warnings and the unreferenced check
    F. compiler buffer organisation and allocation
    G. register allocation

17. Essential E Utilities / Applications
    A. ShowModule
    B. ShowHunk
    C. Pragma2Module / Iconvert
    D. ShowCache / FlushCache
    E. ecompile.rexx
    F. o2m
    G. E-Yacc
    H. SrcGen
    I. Build
    J. EE / Aprof
    K. EDBG
    L. EC PreProcessor

18. Appendices
    A. E Grammar description
    B. Tutorial
    C. Mapping E to C/C++/Pascal/Ada/Lisp etc.
    D. Amiga E FAQ

0. COMPILER AND INTRODUCTION

0A. introduction

E is an object oriented / procedural / unpure functional higher programming language, mainly influenced by languages such as C++, Ada, Lisp etc. It is a general-purpose programming language, and the Amiga implementation is specifically targeted at programming system applications. The number of features of the language is really too great to sum up entirely (see 0E where some of the features new in v3 are listed), and include: speed of >20000 lines/minute on a 7 Mhz amiga, inline assembler and linker integrated into compiler, large set of integrated functions, great module concept with v39 includes as modules, flexible type-system, quoted expressions, immediate and typed lists, low-level and object polymorphism, exception handling, inheritance, data-hiding, methods, multiple return values, default arguments, register allocation, fast memory management, unification, LISP-Cells, (macro-) preprocessor, source-level debugger, and much more...

This is what 'HelloWorld' looks like in E:

```
/* nominated for Most Boring Example */

PROC main()
  WriteF('Hello, World!\n')
ENDPROC
```

0B. the distribution

The distribution includes:

bin/	contains the compiler EC and the support utilities
modules/	Directory containing all Amiga v39 E modules and useful tools as linkable modules
docs/	documentation on E
src/	example sources in E
tools/	optional tools for use with E

This distribution of Amiga E v3.0 including the demo version of the compiler is FreeWare, and may be freely copied. The compiler archive some of you may have received together with this archive, however, is NOT FreeWare and should NOT be copied for anyone but yourself.

Distributing copies of E for more than the price of disk+postage (generally <3$), or any other way of distribution with only the slightest aim of making profit, is not allowed. I explicitly forbid Serge Hammouche or his company "France Festival Distribution" to distribute E in any shape or form. He has done so in the past without my permission, misusing my name and stealing money from me on the backs of french E programmers. If you paid him money, claim it back and spread the word. A french translation of this document authorized by me (made by Olivier Anh) is available on aminet.

This distribution should always be distributed as a whole, i.e. in the form of the original .lha archive. No additions, modifications, translations, partial distributions or whatever are allowed without my explicit permission.

No guarantees, no warranty. If you manage to drown your pet goldfish using E, or E fails to be suitable for ordering pizzas, that's entirely your problem. Whatever happens, don't blame it on me.

Fred Fish has special permission to distribute E on his CD-ROM's.

0C. demo restrictions, registration and sites

The Amiga E distribution is PD, and contains the demo version of the compiler. registered programmers obtain the full compiler as a separate archive, with only "EC" in it.

If you have received the registered EC already, you may wish to put it in the bin directory. The distribution archive may be freely copied for anyone (see 0B on that), but please make sure you do not distribute your registered copy of EC to anyone, not even to friends. I haven't serialised these copies because of the confidence I have in my registered users, so don't break it. Registration is very affordable, and v2.1b is still available, so there's no excuse left for pirating E.

So what's "demo" about the included compiler?
The compiler will only spit out executables <8k, above that it will present you with a "workspace full" error. Other than that it's fully functional.

You are allowed to evaluate this demo for _two weeks_, after which you decide either to register or quit using this demo version, even if you only write programs <8k. [this used to be 12k, yes. guess why it changed.]

updates
distribution updates can be found in the PD as normal, and updates to the registered EC will be distributed as patches, also in the PD.

ordering a copy
The basic price is 40$.

- subtract 5$ if you are prepared to receive your registered EC per uuencoded E-mail instead of on disk. This is the fastest way to get v3.
- add 10$ for each major update you wish to receive (if they are released) . This is only for people living in Siberia or similar, since updates are available in the PD.
- contact me for large quantity orders.

I will allow payment in some other currencies as well, to simplify payment for some people. No rare currencies please.

examples (US$):	40
Dutch Guilders (highly preferred)	65
German DeutschMark	60
Etc...	

NOTE: these are calculated from transaction prices on 30 june 94, if your currency starts to plummet, you'll have to adjust. making sure you're 1$ above the guilder-price seems ok.

Getting the money to me.
This is not simple, since most banks charge a lot for foreign money transfers. Some options:

- Sending cash (preferred). This may sound spooky, but it actually is the most succesful way of getting money across. Changing your money to dutch paper-money is always a good idea. If you feel insecure about mailing money, most post-offices offer a service that will report back to you that everything safely arrived at my place.
  [40$]
- Send a "Mandat the poste international". This has worked from for example France and Italy without problems
  [40$]
- within Europe, send a EuroCheque (in guilders). I believe there are no costs for this, but check locally.
  [make sure you mark "DFL" as currency!]
  [40$]
- transferring directly to my bank account:
  427875951, ABN-AMRO bank, the netherlands or my post-giro account:
  6030387, PostBank, the netherlands
  make sure you cover any transfer costs your bank may charge you, and/or may charge me. for example postbank transfer from outside the netherlands costs 4$, bank transfers can be up to 10$
  [44$ to 50$, netherlands: 40$]
- Do **NOT** send CHEQUES, MONEY ORDERs or similar methods of payment. Cheques (other than the EuroCheques above) banks here ask 15$ tranfer costs for to cash (if I can cash them at all).
  [50$ to 55$]

You may choose any (other) method, but in general realise that it's YOUR task to make sure I am able to cash the full amount of money _easily_. If you do not take transfer costs into account and the final amount is less than 40$, you'll have to do yet another transfer before you can get your registrated version.

Make sure you send as complete info with the money as possible, i.e. full address, internet E-mail, maybe even Amiga-config etc. I'm not going to write out one of those silly registration forms here :-)

my address: (see 0F)

You can also register at one of the authorized E registration sites in Australia, England, the U.S.A. or Germany (Italy soon):

Australia:	Rob Nottage	(information below)
England:	Jason R. Hulance	(information below)
U.S.A.:	Barry Wills	(information below)
Germany:	Jörg Wach	(no full information yet)
Italy:	-	(no confirmation yet)

Below is some information from the various sites, for more info contact them personally. Residents in nearby countries may choose at their option to registrate at these sites as well. Note that generally the same rules go for sites as well: whatever your method of payment, make sure the correct amount is received by the site.

GERMANY

name:	Jörg Wach
address:	Waitzstr. 75, 24105 Kiel, Deutschland/Germany
email:	JCL_POWER@freeway.sh.sub.de
price:	60dm postal delivery, 54dm via email

AUSTRALIA

name:	Rob Nottage
address:	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.
price:	AUS$ 55 (postage or fidonet crashmail), AUS$ 50 (internet or fidonet email), depending on current exchange rates
payment:	- Postal or money order (preferred) - personal or bank cheque (you pay the cost) - direct transfer to bank account (National Australia Bank, fill out a deposite slip for the correct amount, account# 86131 435770223, 'Robert K. Nottage') - Sending cash (your risk)

ENGLAND

name:	Jason R. Hulance (Dept. E)
address:	Formal Systems (Europe) Ltd, 3 Alfred Street, Oxford OX1 4EH, ENGLAND
telephone:	(0869) 324350, out of office hours.
email:	m88jrh@ecs.ox.ac.uk
price:	26 UK pounds (on disk) or 23 UK pounds (via E-mail) "Beginner's Guide to Amiga E" (182 pages) is available in AmigaGuide, TeX, PostScript and printed forms with a large index. This costs 5 UK pounds (non-printed forms) or 10 UK pounds (printed form).
payment:	For safety, please make payment by cheque (payable to "Jason Hulance"), although postal orders/cash/bank transfers will also be accepted

U.S.A.

name:	Barry Wills
address:	5528D Pryor Dr., SAFB, IL 62225
email:	el269@cleveland.freenet.edu
price:	40$ (US mail), 35$ (internet email)
payment:	- money order or cashier check: preferred method. - cash - Okay, but AT YOUR OWN RISK. - personal check: Okay, but expect delays of up to 3 weeks for shipment of your registered compiler.

0D. using the compiler

To install Amiga E on your system, just copy the whole distribution to some place in your system, extend your path to the BIN directory, and assign EMODULES: to the MODULES directory.

syntax of the compiler (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:
```

for 1.2 to 2.03:

```
EC [-opts] <sourcefile>
```

As an example we'll compile the program 'HelloWorld.e'. The compiler will produce an executable 'HelloWorld'.

```
E:>  ec helloworld
Amiga E Compiler/Assembler/Linker v2.4f (c) 91/92/93 $#%!
lexical analysing ...
parsing and compiling ...
no errors
E:>  helloworld
Hello, World!
E:>  list
HelloWorld.e                  89 ----rwed Oggi      17:37:00
helloworld                   656 ----rwed Oggi      17:37:00
2 files - 4 blocks used
E:>
```

note: the compiler cannot be made resident

Last note on compiling the examples: if a program uses module files for library definitions like:

```
MODULE 'GadTools', 'Reqtools'
```

the compiler needs to know where to find them. Two possible solutions:
1. you make the assignment "emodules:" to the modules directory (best).
2. you state in the source code where to look for modules, like:

```
OPT DIR='dh0:src/e/modules'
```

Options.
These are standard AmigaDos options. You can see them at any time by typing 'EC ?'. For 1.2 options the old unix-like switches are used (need to be written together, preceded by a "-"):

LARGE	-l	compiles with large code/data model (see 16A, OPT LARGE).
ASM	-a	puts EC into assembler mode (see 15D, OPT ASM).
NOWARN	-n	suppresses warnings (see 16A, OPT NOWARN)
SYM	-s	adds a symbolhunk to the executable, for use with profilers, debuggers, disassemblers etc.
LINEDEBUG	-L	adds a "LINE" debughunk to the executable, for use with profilers, debuggers etc. (see 17K).
REG=N	-rN	uses N regs per PROC for register-allocation. (default is 0 for the time being, read elsewhere about how to use this!)

QUIET	-q	if there are no errors or warnings, EC won't output anything.
WB	-w	puts wb to front (for scripts)
SHOWBUF	-b	shows buffer memory usage information
ADDBUF=X	-mX	forces EC to allocate more memory for its buffers. X ranges 1..9, the minimum number of 100k blocks to allocate. default is 1 (never needed).
ERRLINE	-e	give the line# the error was on as returnvalue
ERRBYTE	-E	give the byte-offset in the file the error was on as returnvalue
IGNORECACHE	-c	don't use the module-cache in any way
HOLD	-h	wait for a <return> on the commandline before exiting EC.
OPTI		enable all optimisations (currently equals REG=5)
DEBUG		attach debug infos to executable/module (see 17K)

example: ec large blabla
compiles blabla.e with large model.
NOTE: in most standard cases you won't need to use any of these options

0E. changes from v2.1b to v3.0a, and from v3.0a to v3.1a

What's new in v3.0a (for v3.1a, see further below):
[separated in four sections. for the description of these refer to the actual chapter]

1. BIG NEW FEATURES.

- compilation to modules [!] (see 10C)
- oo: object inheritance, private/public, methods etc. (see 14A)
- Default arguments for PROCs (see 6F) (and some builtins, too).
- Multiple return values (see 6G)
- Register variables / allocation (see 16G, 15E, ...).
- builtin REALs (see 12)
- NEW operator (see 4K), and END (see 5M) (with superfast memory manegement)
- powerful syntactic variation of SELECT (see 5J)
- unification (see 4L)
- allows for more complex types in OBJECTs, PTR TO <type> etc. (see 8F) together with more powerful dereferencing (see 4F, x.y[a].z etc.)
- garbage collected Lisp-Cells (see 9I)
- function values (call ptrs to PROCs) (see 6H)
- module caching (see 17D, showcache)
- error reconstruction: pinpoint error at exact spot in line
- whole bunch of new builtin functions (see 9A,...)
- whole bunch of useful modules (will get loads more soon)
- various compiler optimisations to make EC significantly faster
- several code optimisations
- intelligent linker / module system (see 10C, amongst others)
- compiler uses a lot less memory.
- symbol and linedebug hunks (see 0D above for switches).
- various utilities, such as EE (Barry Wills great E Editor), Build (a make), o2m (converts assembly .o files to modules), E-Yacc etc.
- new docs, GREAT tutorial by Jason Hulance (known as 'The Guide').

2. SMALLER NEW FEATURES.

- pointertyping "::" operator (see 4M)
- EXCEPT DO: continue at handler (see 13C)
- EXIT <boolexp> in WHILE/FOR (see 5F)
- single line comments (see 1B)
- for some: os v39 modules.
- character constants may contain "\n\t\e\a" "\0\\\b\q" now. (\q = ")

- warning for flakey "assignments" now with linenum.
- dereferencing a PTR TO <object> with a just a [] results in a pointer now (for example x[1] = next object)
- variety of smaller code-optimisations.
- a lot of string/list/io functions now return useful returnvalues (see 9A,...).
- EC's command line is now ReadArgs() style for 2.04+ users (see 0D).
- better line-endings (see 5A)
- several new builtins (see 9A,...)
- expressions with *2 *4 /2 and /4 get optimized into adds and shifts.

3. BUGS / PROBLEMS FIXED.

- recognises '.e' also on the command line.
- various bugs fixed (and new ones introduced, obviously)
- now also a 'stdin' variable is available
- dereferencing a PTR TO INT with [] would result in an unsigned integer instead of a signed one (NOTE: this may change behaviour!)
- EC could crash on missing ENDIFs: fixed.
- CHAR would align incorrectly.
- RAISE accepts negative constants.
- Div() didn't handle negative numbers.
- inline asm bugs: ADD.L ...,Ax became ADDX, various silly little problems.
- StringF documentation was missing, InStr() incorrect.
- SIZEOF handles CHAR/INT/LONG now too
- WaitIMessage() did something silly
- EC wouldn't consume some very large sources (>500k)
- a WHILE or a UNTIL with an IF-exp as boolean could generate wrong code.
- [exp]:CHAR would generate wrong code
- some syntactical bugs could previously go unnoticed
- console would open in READWRITE mode

4. OTHER CHANGES ONE NEEDS TO BE AWARE OF.

- v2.1b "-s" option deleted.
- 'OPT' now needs to be first thing in a source; other constructs (like 'CONST', 'RAISE') have stricter checking on their position in a source.
- EC may generate 'internal errors' for some cases. If this happens, report, with source what cause it.
- keyword "IS" is equivalent to "RETURN" directly after a "PROC"
- The inline-assembly register conventions have changed (see 15B)

Changes from v3.0a to v3.1a:

Bugs fixed in v3.0b (nearly all were tiny problems, with great consequences!)
- internal errors on 68000 systems
- problems with inherited methods
- modules could have 4 superfluous bytes of code in them
- default arguments in modules could get lost
- method redefinition could cause double declaration errors
- ptr's to <object> in objects in modules would get lost [now ec will try to find the correct object, else type = LONG]

Bugs fixed in v3.0e:
- huge bug: linker wouldn't link modules >32k
- enforcer hit in linker
- String() could trash a register
- errors within an OBJECT were reported on the first line
- wrong code generated for AND.L Dn,regvar and similar instructions

- .B size on An eas wasn't detected for various instructions
- methods could be defined outside of object/module scope
- object declarations could start as private for no reason

New features in v3.1a:
- SourceLevel debugging!!! option DEBUG/S and external debugger EDBG (see 17k)
- Built-in preprocessor!!! (conditional compilation & macro preprocessing), Mac2E and Cpp compatible (see 17L)
- The SUPER keyword allows calling of superclass methods (see 14E)
- Some new example sources, new tool modules (FilledVector, EasyGUI etc.)
- new versions of Aprof and EE
- docs updated: new features, FAQ etc...

Bugs fixed in v3.1a:
- obj.unknownmember could give weird error reports
- ShowModule could display arguments wrong
- Lists [] didn't accept negative floating point numbers
- small problems in E module definitions fixed
- E-Build executed actions in reverse order :-)
- FlushCache wouldn't allow a selective flush
- Unification on complex nested LISP-lists could fail
- NEW-allocated list deallocation routine was missing (see 9F)
- missing closing brackets could cause fuzzy error messages
- some constructions of nested method-calls and NEW with constructors could result in wrong code
- Both RealF() and RealVal() contained bugs that could give wrong results on machines without a FPU
- in some situations method names could clash with members in other modules

Hint for users of older versions: do a diff over both versions of reference.doc

0F. additional information

The Amiga E Compiler was developed over the course of more than three and a half years, after the author's idea of a good programming language, and a quality amiga-specific compiler for it. It was programmed (as you might have guessed) in 100% assembly, using the AsmOne assembler v1.02. All other support programs were written in E itself.

Special thanks go to the following people:

Barry Wills - for being the best best betatester ever.
Jason Hulance - for his work on 'The Beginners Guide', and betatesting.
Rob Verver - for comments/inspiration.

I also would like to thank the following people for various reasons (no particular order):

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...

This compiler was programmed with great care towards reliability, and even more so the code it generates, additionally it has been tested and debugged for a long period. However, it is not impossible that it contains bugs. if you happen to find one, or have other comments/questions, write me at the address below: I _strongly_ prefer E-mail above conventional mail.

NOTE WELL: due to the immense popularity of the previous version of Amiga E, I get an almost unreplyable amount of Email, some of which (>90%) are questions that would not have been necessary if people read all the docs carefully. What I mean to say is that I like to receive Email, and I don't mind answering questions and helping people out with programming problems, but be sure to check all other information at your disposal (like the Amiga E docs or the RKRM's) to see if your question is relevant, before mailing me. Especially

questions that are not E specific but Amiga specific should not be directed to me. And try and be gentle on me with comments like "I think E should have feature X..." (see 18D, the FAQ)

registrations (see 0C) and donations are welcome at the following address:

Wouter van Oortmerssen
Levendaal 87
2311 JG Leiden
HOLLAND

or better if you have access to Email:

wouter@mars.let.uva.nl
wouter@alf.let.uva.nl (if mars bounces)

1. FORMAT

1A. tabs,lf etc.

E-sources are pure ascii format files, with the linefeed <lf> and semicolon ";" being the separators for two statements. Statements that have particularly many arguments, may be spread over several lines by ending a line with a comma (or any other lexical element that can normally never occur at the end of a line), thus ignoring the following <lf> (see 5A for line endings). Any lexical element in a source code file may be separated from another by any number of spaces, tabs etc.

1B. comments

comments may be placed anywhere in a source file where normally a space would have been correct. They start with '/*' and end with '*/' and may be nested infinitely. The same goes for one-line comments, which start with a '->' and end at the first <lf>.

```
/* this, is a comment */
-> this one too.
```

1C. identifiers and types

identifiers are strings that the programmer uses to denote certain objects, in most cases variables, or even keywords or function names predefined by the compiler. An identifier may consist of:

- upper and lowercase characters
- "0" .. "9" (except as first character)
- "_" (the underscore)

All characters are significant, but the compiler just looks at the first two to identify the type of identifier it is dealing with:

both uppercase:	- keyword like IF, PROC etc.
	- constant, like MAX_LENGTH
	- assembly mnemonic, like MOVE
first lowercase:	- identifier of variable/label/object etc.
first upper and second lower:	- E system function like: WriteF()
	- library call: OpenWindow()

Note that all identifiers obey this syntax, for example:

`WBenchToFront()` becomes `WbenchToFront()`

2. IMMEDIATE VALUES

Immediate values in E all evaluate to a 32bit result; the only difference among these values (A-G) is either their internal representation, or the fact that they return a pointer rather than a value.

2A. decimal (1)

A decimal value is a sequence of characters "0" .. "9", possibly preceded by a minus "-" sign to denote negative numbers.
Examples: 1, 100, -12, 1024

2B. hexadecimal ($1)

A hexadecimal value uses the additional characters "A" .. "F" (or "a" .. "f") and is preceded by a "$" character.
Examples: $FC, $DFF180, -$ABCD

2C. binary (%1)

Binary numbers start with a "%" character and use only "1" and "0" to form a value.
Examples: %111, %1010100001, -%10101

2D. float (1.0)

Floats differ only from normal decimal numbers in that they have a "." to separate their two parts. Either one may be omitted, not both. Note that floats have a different internal 32bit (IEEE) representation (see 12A for more information on floats).
Examples: 3.14159, .1 (=0.1), 1. (=1.0)

2E. character

The value of a character (enclosed in double "" quotes) is ist ascii value, i.e. "A" = 65. In E, character immediate values may be a short string up to 4 characters, for example "FORM", where the first character "F" will be the MSB of the 32bit representation, and "M" the LSB (least significant byte).

2F. string ('bla')

Strings are any ascii representation, enclosed in single " quotes. he value of such a string is a pointer to the first character of it. More specific: 'bla' yields a 32bit pointer to a memory area where we find the bytes "b", "l" and "a". ALL strings in E are terminated by a zero byte. Strings may contain format signs introduced by a slash "\", either to introduce characters to the string that are for some reason not displayable, or for use with string formatting functions like WriteF(), TextF() and StringF(), or kick2 Vprintf().

\n	a linefeed (ascii 10)
\a or "	an apostrophe ' (the one used for enclosing the string)
\q	a doublequote: "
\e	escape (ascii 27)
\t	tab (ascii 9)
\\	a backslash
\0	a zero byte. Of rare use, as ALL strings are 0-terminated
\b	a carriage return (ascii 13)

Additionally, when used with formatting functions:

\d	print a decimal number
\h	print a hexadecimal
\s	print a string
\c	print a character
\z	set fill byte to '0' character
\l	format to left of field
\r	format to right of field (these last two act as toggles)

Field specifiers may follow the \d,\h and \s codes:

[x]	specify exact field width x
(x,y)	specify minimum x and maximum y (strings only)

Example: print a hexadecimal number with 8 positions and leading zeroes:

```
WriteF('\z\h[8]\n',num)
```

A string may be extended over several lines by trailing them with a "+" sign and a <lf>:

```
'this specifically long string ' +
'is separated over two lines'
```

2G. lists ([1,2,3]) and typed lists

An immediate list is the constant counterpart of the LIST datatype, just as a 'string' is the constant counterpart for the STRING or ARRAY OF CHAR datatype. Example:

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

is an expression that has as value a PTR to an already initialised list, a list as a representation in memory is compatible with an ARRAY OF LONG, with some extra length information at a negative offset. You may use these immediate lists anywhere a function expects a PTR to an array of 32bits values, or a list. Examples:

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

(see 9C on list-functions for a discussion on typed-immediate lists and detailed information).

2H. lisp-cells (<a|b>)

(see 9I to read about cells in detail)

3. EXPRESSIONS

3A. format

An expression is a piece of code held together by operators, functions and parentheses to form a value.
They mostly consist of:

- immediate values (see 2A,...)
- operators (see 4A,...)
- function calls (see 3D)
- parentheses () (see 3B)
- variables or variable-expressions (see 3C)

examples of expressions:

```
1
'hello'
$ABCD+(2*6)+Abs(a)
(a<1) OR (b>=100)
```

3B. precedence and grouping

The E language has _no_ precedence whatsoever. This means that expressions are evaluated left to right. You may change precedence by parenthesizing some (sub-)expression:

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

3C. types of expressions

There are three types of expressions that may be used for different purposes:

- <var>, consisting of just a variable
- <varexp>, consisting of a variable, possibly with unary operators with it, like "++" (increment) "." (member selection) or [] (array operator). (see 4D, 4G). It denotes a modifiable expression, like Lvalues in C. Note that those (unary) operators are not part of any precedence.
- <exp>. This includes <var> and <varexp>, and any other expression.

3D. function calls

A function call is a temporary suspension of the current code for a jump to a function, this may be either a self-written function (PROC), or a function provided by the system. The format of a function call is the name of the function, followed by two parentheses () enclosing zero to unlimited arguments, separated by commas ",". Note that arguments to functions are again expressions. (see 6A) on how to make your own functions, (see 9A and 10A) on built-in functions.
Examples:

```
foo(1,2)
Gadget(buffer,glist,2,0,40,80+offset,100,'Cancel')
Close(handle)
```

4. OPERATORS

4A. math (+ - * /)

These infix operators combine an expression with another value to yield a new value.
Examples:

```
1+2, MAX-1*5
```

(see 12A on how to overload these operators for use with floats). "-" may be used as the first part of an expression, with an implied 0, i.e. -a or -b+1 is legal.
Note that * and / are by default 16bit operators: (see 9G, Mul())

4B. comparison (= <> > < >= <=)

Equal to math operators, with the difference that they result in either TRUE (32bit value -1), or FALSE. These can also be overloaded for floats.

4C. logical and bitwise (AND OR)

These operators either combine truth values to new ones, or perform bitwise AND and OR operations.
Examples:

```
(a>1) AND ((b=2) OR (c>=3))           /* logical */
a:=b AND $FF                          /* bitwise */
```

4D. unary (SIZEOF ^ {} ++ -- `)

- SIZEOF <objectident>
  simply returns the size of a certain object or CHAR/INT/LONG.
  Example:

  ```
  SIZEOF newscreen + SIZEOF INT
  ```

- {<var>}
  Returns the address of a variable or label. This is the operator you would use to give a variable as argument to a function by reference, instead of by value, which is default in E. See "^".
  Example:

  ```
  Val(input,{x})
  ```

- ^<var>
  The counterpart of {}, writes or reads variables that were given by reference, examples: ^a:=1 b:=^a it may also be used to plainly "peek" or "poke" LONG values from memory, if <var> is pointer to such a value.

  Example for {} and ^: write your own assignment function:

  ```
  PROC set(var,exp)
    ^var:=exp
  ENDPROC
  ```

  and call it with:

  ```
  set({a},1)                    /* equals a:=1 */
  ```

- <varexp>++ and <varexp>--
  Increases (++) or decreases (--) the pointer that is denoted by <varexp> by the size of the data it points to. This has the effect that that pointer points to the next or previous item. When used on variables that are not pointers, these will simply be changed by one.

Note that ++ always takes effect _after_ the calculation of <varexp>, -- always _before_.

Examples:

```
a++                          /* return value of a, then increase by one */
sp[]--                       /* decrease pointer sp by 4 (if it were an array of long),
                                and read value pointed at by sp */
```

- `<exp>
  This is called a quoted expression, from LISP. <exp> is not evaluated, but instead returns the address of the expression, which can later be evaluated when required. More on this special feature in chapter 11

4E. triple (IF THEN ELSE)

The IF operator has quite the same function as the IF statement, only it selects between two expressions instead of two statements or blocks of statements. It equals the x?y:z operator in C.

```
IF <boolexp> THEN <exp1> ELSE <exp2>
```

returns exp1 or exp2, according to boolexp. For example, instead of:

```
IF a<1 THEN b:=2 ELSE b:=3
IF x=3 THEN WriteF('x is 3\n') ELSE WriteF('x is something else\n')
```

write:

```
b:=IF a<1 THEN 2 ELSE 3
WriteF(IF x=3 THEN 'x is 3\n' ELSE 'x is something else\n')
```

4F. structure (.)

<ptr2object>.<memberofobject> builds a <varexp>
The pointer has to be declared as PTR TO <object> or ARRAY OF <object> (see 8D for these), and the member has to be a legal object identifier. Note that reading a subobject in an object this way results in a pointer to that object. Examples:

```
thistask.userdata:=1
rast:=myscreen.rastport
```

If the member you select is of type PTR TO <type>, you may use "." and "[]" to dereference further values. If you select an ARRAY or substructure in an OBJECT, the result is again a PTR.

4G. array ([])

<var>[<indexexp>] (is a <varexp>)
This operator reads the value from the array <var> points to, with index <indexexp>. The index may be just about any expression.
Note1: "[]" is a shortcut for "[0]"
Note2: with an array of n elements, the index is 0 .. n-1
Examples:

```
a[1]:=10             /* sets second element to 10 */
x:=table[y*4+1]      /* reads from array */
x.y[3]               /* accesses array in an object */
```

4H. float operator (!)

<exp>!<exp>
Converts expressions from integer to float and back, and overloads operators + - * / = <> < > <= >= with float equivalents. (see 12A to read all about floats and this operator).

4I. assignments expressions (:=)

Assignments (setting a variable to a value) exist as statement and as expression. The only difference is that the statement version has the form <varexp>:=<exp> and the expression <var>:=<exp>. The latter has the value of <exp> as result. Note that as <var>:= takes on an expression, you will often need parentheses to force correct interpretation, like:

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

is interpreted like:

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

which is not what you mean: mem should be a pointer, not a boolean. but you should write:

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

it's a good habit to parenthesize any assignment expression that is part of another one, if not already disambiguated by other constructs such as "bla(a:=1)", "b:=a:=1" etc.

4J. sequencing (BUT)

The sequencing operator "BUT" allows two expressions to be written in a construction that allows for only one. Often in writing complex expressions/function calls, one would like to do a second thing on the spot, like an assignment. Syntax:

```
<exp1> BUT <exp1>
```

this says: evaluate exp1, but return the value of exp2.
Example:

```
myfunc((x:=2) BUT x*x)
```

assign 2 to x and then calls myfunc with x*x. The () around the assignment are again needed to prevent the := operator from taking (2 BUT x*x) as an expression.

4K. dynamic memory allocation (NEW)

the NEW operator is a powerful operator for dynamic memory allocation. it has various forms:

(assuming DEF p:PTR TO whateverobj, q:PTR TO INT)

```
NEW p
```

is an expression that will allocate zero-ised memory for the object-size p points to. the resulting pointer will be put in p, and is also the value of the expression. if NEW fails to get memory, it raises a "NEW" exception. 'NEW p' is therefore roughly equivalent with:

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

with the difference that p never gets any value if an exception is raised, and that the former is of course an expression, not a statement

but there's more: one can also allocate an array dynamically:

```
NEW p[10]                       /* array of 10 objects */

NEW q[a+1]                      /* array of INT, size is runtime computed */
```

(this doesn't work when instantiating classes)
A problem with list [1,2,3] expressions sometimes is that they are static, and when using these to build large datastructures they would need to be created at run-time. one may then use the dynamic equivalent to static lists:

```
p:=[1,2,3]:whateverobj          /* static structure */

p:=NEW [1,2,3]:whateverobj      /* dynamicly allocated! */
```

this works with both lists and typed-lists, and also with arrays:

```
NEW [1,2,3]                     /* constant-list, dyn. (note: not a like a listvar!) */
NEW [1,2,3]:obj                 /* object */
NEW [1,2,3]:INT                 /* array of INTs */
```

freeing memory allocated with any variations of NEW is done automatically at the end of the program, or by hand with END or FastDispose(). (note: the only exception is NEW <list>, use FastDisposeList())

If you use NEW [...]:obj, and the number of fields is less than the one present in the object, additional 0/NIL/"\0" whatever fields will be added. this allows one to extend objects without getting into trouble with allocations like these. (note that this is different from the static [...]:obj)

when you use NEW as a statement, multiple pointers may follow, i.e.:

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

is valid.
(see 5M, END)
(see 9F for the fast memory allocation functions NEW makes use of)

4L. unification (<=>)

Unification allows a totally different style of programming that will be familiar to you if you're used to either Logic (ProLog), equational or functional (Miranda/Gofer/Haskell) programming. Most of us when using structures/arrays whatever are used to get values from it by selection ("." and "[]"), in these languages however pattern matching is used.

in E:

```
exp <=> uni_exp
```

exp can be any expression, but in v3 it's only really useful if it somehow is a pointer to a list. uni_exp is the pattern that is used to match exp. All constants in uni_exp much match to values in exp, variables are set to their respective values. if something doesn't match, no variables get a value, and the expression has the result FALSE. otherwise TRUE.
example:

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

this will succeed with x=2 and y=3. If list a were to be another lenght than 3, the match would fail too. The fact that a is a list in the first place is something you need to assure by yourself, EC simply tries to fit the uni_exp on whatever value it gets as exp.
examples of FALSE:

```
a <=> [1,x]               -> wrong list-len
a <=> [1,4,x]             -> 4=2 fails
'bla' <=> [1,2]           -> unpredictable result / crash ?
```

The fun thing with unification is that you can do very complex matches, and that if you take the first field or so as a constant telling what the structure is, you have a nice form of dynamic typing. And, all the time without using PTRs!
a slightly nicer example:

```
[BLA,[1,'burp'],['bla',"bla"]] <=> [BLA,[1,x],y]
```

binds:

```
x='burp', y=['bla',"bla"]
```

or even:

```
IF myexp <=> [PLUS,[MUL,a,1],[SUBS,[PLUS,c,d],e]] THEN RETURN a+c+d-e
```

maybe a silly example, but one could imagine doing complex stuff like this in a compiler's code-optimizer. it equals this traditional code:

```
IF ListLen(myexp)=3
```

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

only then a bit more optimal. As you see there's a lot of expressive power involved in unification as compared to traditional selection-based programming.

For now, the only thing allowed in unification are untyped lists, (integer) constants, variables and LISP-Cells (see 9I for that). The future will see this expanded with strings, typed-lists/objects, and even expressions [!]

4M. pointer typing (::)

when you write an expression like 'p' where p is a PTR TO object, this allows you to dereference it as such. The pointer typing operator allows you to change such a type on the fly:

```
DEF p:PTR TO mp            -> message port

p.sigbit                   -> access it
p::ln.name                 -> access pointer as if it were a node
```

this is very useful for pointers that can point to different sorts of things or objects that are part of each other, where of course only one declaration is possible.

A second handy use is in OBJECTs that have members declared as LONG, which are actually pointers. You may type it on the fly to dereference it anyway:

```
mywindow.userport::mp.sigbit
```

here the window OBJECT has userport defined as LONG, so you wouldn't be able to dereference it normally.

5. STATEMENTS

5A. format (;)

As suggested in chapter 1A, a statement generally stands in ist own line, but several of them may be put together on one line by separating them with semicolon, or one may be spread over more than one line by ending each line in a comma ",". Examples:

```
a:=1; WriteF('hello!\n')
DEF a,b,c,d,                        /* too many args for one line (faked) */
    e,f,g
```

statements may be:

- assignments
- conditional statements, for statements and the like, (see 5E-5K)
- void expressions
- labels
- assembly instructions

The comma is the primary character to show that you do not wish to end the statement with the next linefeed, but from v3 on, any token that cannot legally be the end of a line causes the statement to continue. Furthermore, if not all "[" and "(" occuring in a statement have been closed off, a statement will continue also. examples of such tokens:

```
+ - * / =
< >= <=
:= . <=> ::
{ [ (
AND OR BUT THEN IS                  -> others too, but these are most useful
```

example of bracketing:

```
a:=[
    [1,2],
    [3,4]
   ]                                -> assignment ends here.
```

5B. statement labels and gotos (JUMP)

Labels are global-scoped identifiers with a ':' added to them, as in:

```
mylabel:
```

they may be used by instructions such as JUMP, and to reference static data. They may be used to jump out of all types of loops (although this technique is not encouraged, (see 5F, EXIT), but not out of procedures. In normal E programs they are mostly used with inline assembly. Labels are always globally visible.

Usage of JUMP: JUMP <label>
continues execution at <label>. You are not encouraged to use this instruction, it's there for situations that would otherwise increase the complexity of the program. Example:

```
IF done THEN JUMP stopnow

/* other parts of program */

stopnow:
```

5C. assignment (:=)

The basic format of an assignment is: <var> := <exp>
Examples:

```
a:=1
```

```
a:=myfunc()
a:=b*3
```

In E one can assign multiple variables at once, when <exp> is a function that returns multiple returnvalues. (see 6G)

5D. assembly mnemonics

In E, inline assembly is a true part of the language, they need not be enclosed in special "ASM" blocks or the like, as is usual in other languages, nor are separate assemblers necessary to assemble the code. This also means that it obeys the E syntax rules, etc.
(see 15A to read all about the inline assembler). Example:

```
DEF a,b
b:=2
MOVEQ  #1,D0                    /* just use some assembly statements */
MOVE.L D0,a                     /* a:=1+b  */
ADD.L  b,a
WriteF('a=\d\n',a)              /* a will be 3 */
```

5E. conditional statement (IF)

IF, THEN, ELSE, ELSEIF, ENDIF

syntax:

```
IF <exp> THEN <statement> [ ELSE <statement> ]
```

or:

```
IF <exp>
    <statements>
[ ELSEIF <exp>                  /* multiple elseifs may occur */
    <statements> ]
[ ELSE ]
    <statements>
ENDIF
```

builds a conditional block. Note that there are two general forms of this statement, a single-line and a multiple-line version.

5F. for-statement (FOR)

FOR, TO, STEP, DO, ENDFOR

syntax:

```
FOR <var> := <exp> TO <exp> STEP <step> DO <statement>
```

or:

```
FOR <var> := <exp> TO <exp> STEP <step>
    <statements>
ENDFOR
```

builds a for-block, note the two general forms. <step> may be any positive or negative constant, excluding 0, and is optional. Example:

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

The body of FOR may contain EXIT statements with the following syntax:

```
EXIT <boolexp>
```

Which allows you to exit the loop if boolexp is true.

5G. while-statement (WHILE)

WHILE, DO, ENDWHILE

syntax:

```
WHILE <exp> DO <statement>
```

or:

```
      WHILE <exp>
                  <statements>
      ENDWHILE
```

builds a while-loop, which is repeated as long as <exp> is TRUE. Note the one-line/one-statement version and the multiple line version.

WHILE may also contain EXIT statements, like FOR.

5H. repeat-statement (REPEAT)

REPEAT, UNTIL

syntax:

```
      REPEAT
      UNTIL <exp>
```

builds a repeat-until block: it will continue to loop this block until <exp>=TRUE. Example:

```
   REPEAT
     WriteF('Do you really, really wish to exit this program?\n')
     ReadStr(stdout,s)
   UNTIL StrCmp(s,'yes please!')
```

5I. loop-statement (LOOP)

LOOP, ENDLOOP

syntax:

```
      LOOP
        <statements>
      ENDLOOP
```

builds an infinite loop.

5J. select-case-statement (SELECT)

SELECT, CASE, DEFAULT, ENDSELECT

syntax:

```
      SELECT <var>
      [ CASE <exp>
        <statements> ]
      [ CASE <exp>
        <statements> ]   /* any number of these blocks */
      [ DEFAULT
        <statements> ]
      ENDSELECT
```

builds a select-case block. Various expressions will be matched against the variable, and only the first matching block executed. If nothing matches, a default block may be executed.

```
   SELECT character
     CASE 10
       WriteF('Gee, I just found a linefeed\n')
     CASE 9
       WriteF('Wow, this must be a tab!\n')
     DEFAULT
       WriteF('Do you know this one: "\c" ?\n',character)
   ENDSELECT
```

next to the old SELECT <var> which works on expressions in the CASE statements, E has a SELECT <maxrange> OF <exp> which works with constants and or ranges of constants in a CASE. not only is this for many applications more powerful, it is also much faster for large numbers of cases (>5 usually), or if cases are equally probable. It assumes however, that all CASEs lie within a small integer range from 0 TO n-1, where n is something reasonable, for example 10, or 255 for characters.

example:

```
SELECT 127 OF FgetC(stream)
  CASE "\n","\b"
    WriteF('line ending\n')
  CASE "\t"," "
    WriteF('whitepace\n')
  CASE "0" TO "9"
    WriteF('Integer\n')
  CASE "A" TO "Z", "a" TO "z", "_"
    WriteF('Identifier\n')
  DEFAULT
    WriteF('some other character\n')
ENDSELECT
```

DEFAULT will be hit not only for those for which there is no CASE, but also for the chars that fall out of the range, i.e. 128 TO 255 (and >255, and <0).
note that speed costs: because this SELECT uses a jump-table to quickly get at the right CASE, it'll use 2*<maxrange> bytes, 256 in this case.

5K. increase statement (INC/DEC)

INC, DEC

syntax:

```
INC <var>
DEC <var>
```

short for <var>:=<var>+1 and <var>:=<var>-1. Only difference withvar++ and var-- is that these are statements, and do not return a value.

5L. void expressions (VOID)

VOID

syntax:

```
VOID <exp>
```

calculates the expression without the result going anywhere. Only useful for a clearer syntax, as expressions may be used as statements without VOID in E anyway. This may cause subtle bugs though, as "a:=1" assigns a the value 1, but "a=1" as a statement will do nothing. E will give you a warning if this happens.

5M. memory deallocation (END)

END is the complement to NEW. any pointer obtained should be deallocated (and only!) with END

```
END a
```

or even

```
END a,b,c
```

where the arguments are PTRs to some type. END frees the amount of space that is being pointed to, so if a is PTR TO LONG it will only free 4 bytes. so if you allocated a with NEW a[NUM], free it with

```
END a[NUM]
```

NIL-pointers may safely be passed to NEW. Pointers are also nuked to NIL afterwards. If a is a PTR to an object that is an instance of a class, END will dynamically get the size to be freed from the class-object, so if you have an object of class b which is 32 bytes, but the pointer you're freeing with is a baseclass pointer (only 24 bytes), END will correctly deallocate 32 bytes. this only works for classes.

You can imagine END p as a macro for:

```
IF p
```

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

note that the NEW and END make use of the FastNew() and FastDispose() functions (described elsewhere) which work quite differently from NewR() and Dispose(). (see 9F for details).

6. FUNNCTION DEFINITIONS AND DECLARATIONS

6A. proc definition and arguments (PROC)

You may use PROC and ENDPROC to collect statements into your own functions. Such a function may have any number of arguments, and several return values.

PROC, ENDPROC

syntax:

```
PROC <label> ( <args> , ... )
ENDPROC <returnvalue>, ...
```

defines a procedure with any number of arguments. Arguments are of type LONG or optionally of type PTR TO <type> (see 8B) and need no further declaration. The end of a procedure is designated by ENDPROC. If no return value is given, 0 is returned. Example: write a function that returns the sum of two arguments:

```
PROC add(x,y)          /* x and y are local variables */
ENDPROC x+y            /* return the result  */
```

a short version:

```
PROC add(x,y) IS x+y
```

6B. local and global definitions: scope (DEF)

You may define additional local variables besides those which are arguments with the DEF statement. The easiest way is simply like:

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

declares the identifiers a, b and c as variables of your function. Note that such declarations should be at the start of your function.

DEF

syntax:

```
DEF <declarations>,...
```

declares variables. A declaration has one of the forms:

<var>	
<var>:<type>	where <type>=LONG,<objectident>,...
<var>[<size>]:<type>	where <type>=ARRAY,STRING,LIST

(see 8A for more examples, as that is where the types are introduced). For now, we'll use the <var> form.
Arguments to functions are restricted to basic types; (see 8B). A declaration of a basic type can have an initialisation, in the current version this must be an integer (not an expression):

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

A program consists of a set of functions, called procedures, PROCs. Each procedure may have Local variables, and the program as a whole may have Global variables. At least one procedure should be the PROC main(), as this is the module where execution begins. A simple program could look like:

```
DEF a, b                        /* definition of global vars */

PROC main()                     /* all functions in random order */
  bla(1)
ENDPROC

PROC bla(x)
  DEF y,z                       /* possibly with own local vars */
ENDPROC
```

To summarize, local definitions are the ones you make at the start of procedures, and which are only visible within that function. Global definitions are made before the first PROC, at the start of your source code, and they are globally visible. Global and local variables (and of course local variables of two different functions) may have the same name, local variables always have priority.

6C. endproc/return

As stated before, ENDPROC marks the end of a function definition, and may return a value. Optionally RETURN may be used at any point in the function to exit, if used in main(), it will exit the program. (see 9F, CleanUp()).

```
RETURN [<returnvalue>]                /* optional */
```

Example:

```
PROC getresources()
  /* ... */
  IF error THEN RETURN FALSE          /* something went wrong, so exit and fail */
  /* ... */
ENDPROC TRUE                          /* we got this far, so return TRUE */
```

a very short version of a function definition is:

```
PROC <label> ( <arg> , ... ) RETURN <exp>
```

(or "IS" instead of "RETURN")
These are function definitions that only make small computations, like faculty functions and the like: (one-liners :-)

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

6D. the main function

The PROC called main is only of importance because it is called as first function; it behaves exactly the same as other functions, and may also have local variables. Main has no arguments: the command-line arguments are supplied in the system-variable "arg", or can be checked with ReadArgs() (dos.library function)

6E. built-in system variables

Following global variables are always available in you program, they're called system variables.

arg	As discussed above, contains a pointer to a zero-terminated string, containing the command-line arguments. Don't use this variable if you wish to use ReadArgs() instead.
stdout	Contains a file-handle to the standard output (and input). If your program was started from the workbench, so no shell-output is available, WriteF() will open a CON: window for you and put its file handle here.
stdin	file-handle of standard input
conout	This is where that file handle is kept, and the console window will be automatically closed upon exit of your program. (see 9A, WriteF(), on how to use these two variables properly).
execbase, dosbase, gfxbase, intuitionbase	These four variables are always provided with their correct values.
stdrast	Pointer to standard rastport in use with your program, or NIL. The built-in graphics functions like Line() make use of this variable.
wbmessage	Contains a ptr to a message you got if you started from wb, else NIL. May be used as a boolean to detect if you started from workbench, or even check any arguments that were shift-selected with your icon. See WbArgs.e in the Src/Args dir how to make good use of wbmessage.

6F. default arguments to functions

default arguments allows you to specify for one or more arguments of a procedure which is the default value, if the procedure is called with less args than parameters. for example, a procedure like:

```
PROC bla(a,b=1,c=NIL)
```

can be called like:	is equivalent with:
`bla(a,b,c)`	`bla(a,b,c)`
`bla(a,b)`	`bla(a,b,NIL)`
`bla(a)`	`bla(a,1,NIL)`

This can be useful and also express something about the procedures function, i.e. that most of the time one would call it with NIL anyway, so why not leave it out for clarity. That's also why you should not overdo it with D.A.: do not start specifying non-sensical values for procedures out of pure laziness, if you feel a certain parameter reallyhas no default value.

to make calls with fewer args unambiguous, D.A. declarations can only apply to the last 0..n parameters of a PROC of n parameters.

for example, illegal is:

```
PROC bla(a,b=1,c)
```

(you should then simply reorder the parameters, of course). arguments supplied in a call are filled in from left to right, missing arguments are added with D.A.'s as needed.

6G. multiple return values

In E you can return any number of return values (max 3 in Amiga E because of implementation reasons). How?

```
RETURN <exp>,<exp>,<exp>         (or ENDPROC, of course)
```

example:

```
PROC sincos(rad)
  DEF sin,cos
  /* whatever computation is needed */
ENDPROC sin,cos
```

call with:

```
s,c:=sincos(3.14)
s:=sincos(1.00)
```

as you can see, there's a new statement of the form:

```
<var> , ... := <exp>
```

where <exp> makes mostly only sense as function call. note two things:

- you can decide yourself how many values you wish to receive. this makes sense when the first retval is the main one, and the second/third optional infos, which might only be important to some callers.
- this form is a _statement_. this means that when you would call sincos() as part of another expression, only the first (the regular) return value is used: fun(sincos(1.0))

6H. function values

With v3, you can also have functions as values, and pass these freely around. they're different from quoted expression, since they're called just like normal PROCs. example:

```
fun:={myproc}              -> get PROC ptr
...
fun(1,2,3)                 -> apply to args, just like normal PROC
```

notes:

- you have to be sure that the PROC you have a ptr to and the number of args are the same. the compiler can't check this for you.
- even worse: you have to be sure the ptr is a PROC at all. there is a compiler warning that may help you with this.

7. DECLARATION OF CONSTANTS

7A. const (CONST)

syntax:
```
CONST <declarations>,...
```

Enables you to declare a constant. A declaration looks like:

```
<ident>=<value>
```

constants must be uppercase, and will in the rest of the program be treated as <value>. Example:

```
CONST MAX_LINES=100, ER_NOMEM=1, ER_NOFILE=2
```

You cannot declare constants in terms of others that are being declared in the same CONST statement: put these in the next.

CONST, ENUM and SET declarations are always global, i.e. it is not possible to declare constants local to a PROC. The best place for constant declarations is at the top of your source, but EC also allows you to put them between two PROCs, for example.

7B. enumerations (ENUM)

Enumerations are a specific type of constant that need not be given values, as they simply range from 0 .. n, the first being 0. At any given point in an enumeration, you may use the '=<value>' notation to set or reset the counter value. Example:

```
ENUM ZERO, ONE, TWO, THREE, MONDAY=1, TUESDAY, WEDNESDAY

ENUM ER_NOFILE=100, ER_NOMEM, ER_NOWINDOW
```

7C. sets (SET)

Sets are again like enumerations, with the difference that instead of increasing a value (0,1,2,...) they increase a bitnumber (0,1,2,...) and thus have values like (1,2,4,8,...). This has the added advantage that they may be used as sets of flags, as the keyword says. Suppose a set like the one below to describe properties of a window:

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

to initialise a variable to properties DEPTH and SIZEGAD:

```
winflags:=DEPTH OR SIZEGAD
```

to set an additional SCROLLBAR flag:

```
winflags:=winflags OR SCROLLBAR
```

and to test if either of both of two properties hold:

```
IF winflags AND (SCROLLBAR OR DEPTH) THEN /* whatever */
```

7D. built-in constants

Following are built-in constants that may be used:

TRUE,FALSE	Represent the boolean values (-1,0)
NIL	(=0), the uninitialised pointer.
ALL	Used with string functions like StrCopy() to copy all characters
GADGETSIZE	Minimum size in bytes to hold one gadget; (see 9D, Gadget())

OLDFILE,NEWFILE	Mode-parameters for use with Open()
EMPTY	used with methods (might be keyword in the future)
STRLEN	Always has the value of the length of the last immediate string used. Example:

```
Write(handle,'hi folks!',STRLEN)          /* =9 */
```

8. TYPES

8A. about the 'type' system

E doesn't have a rigid type-system like Pascal or Modula2, it's even more flexible than C's type system: you might as well call it a datatype-system. This goes hand in hand with the philosophy that in E all datatypes are equal: all basic small values like characters, integers etc. All have the same 32bit size, and all other datatypes like arrays and strings are represented by 32bit pointers to them. This way, the e compiler can generate code in a very polymorphic way.
The (dis)advantages are obvious:

disadvantages of the E-type system
- less compiler checking on silly errors you make

advantages:
- low-level polymorphism, easier to make powerful generic functions.
- flexible way of programming: no problem that some types of return values don't match, no superfluous "casts" etc., no unnecessary errormessages.
- no hard to find errors when mixing data of different sizes in expressions

8B. the basic type (LONG/PTR)

There's only one basic, non-complex variable type in E, which is the 32bit type LONG. As this is the default type, it may be declared as:

```
DEF a:LONG
```

or just:

```
DEF a
```

This variable type may hold what's known as CHAR/INT/PTR/LONG types in other languages. A special variation of LONG is the PTR type. This type is compatible with LONG, with the only difference that it specifies to what type it is a pointer. By default, the type LONG is specified as PTR TO CHAR. Syntax:

```
DEF <var>:PTR TO <type>
```

where type is either a simple type or a compound type. Example:

```
DEF x:PTR TO INT, myscreen:PTR TO screen
```

Note that 'screen' is the name of an object as defined in intuition/screens.m For example, if you open your own screen with:

```
myscreen:=OpenS(...   etc.
```

you may use the pointer myscreen as in 'myscreen.rastport'. However, if you do not wish to do anything with the variable until you call CloseS(myscreen), you may simply declare it as

```
DEF myscreen
```

Variable declarations may have optional initialisations, but only integer constants, i.e. no full expression:

```
DEF a=1, b=NIL:PTR TO textfont
```

8C. the simple type (CHAR/INT/LONG)

The simple types CHAR (8bit) and INT (16bit) may not be used as types for a basic (single) variable; the reason for this must be clear by now. However they may be used as data type to build ARRAYs from, set PTRs to, use in the definition of OBJECTs etc. See those for examples.

8D. the array type (ARRAY)

ARRAYs are declared by specifying their length (in bytes):

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

this defines an array of 100 bytes. Internally, 'b' is a variable of type LONG and a PTR to this memory area. Default type of an array-element is CHAR, it may be anything by specifying:

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

where "newmenu" is an example of a structure, called OBJECTs in E.
Array access is very easy with:

```
<var>[<sexp>]

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

Note that the index of an array of size n ranges from 0 to n-1, and not from 1 to n.
Note that ARRAY OF <type> is compatible with PTR TO <type>, with the only difference that the variable that is an ARRAY is already initialised.

8E. the complex type (STRING/LIST)

- STRINGs. Similar to arrays, but different in the respect that they may only be changed by using E string functions, and that they contain length and maxlength information, so string functions may alter them in a safe fashion, i.e: the string can never grow bigger than the memory area it is in. Definition:

  ```
  DEF s[80]:STRING
  ```

  The STRING datatype (called an estring) is backwards compatible with PTR TO CHAR and of course ARRAY OF CHAR, but not the other way around. (see 9B on string functions for more details).

- LISTs. These may be interpreted as a mix between a STRING and an ARRAY OF LONG. I.e: this data structure holds a list of LONG variables which may be extended and shortened like STRINGs. Definition:

  ```
  DEF x[100]:LIST
  ```

  A powerful addition to this datatype is that it also has a 'constant' equivalent [], like STRINGs have ''. LIST is backward compatible with PTR TO LONG and of course ARRAY OF LONG, but not the other way around.
  (see 9C and 2G) for more on this.

8F. the compound type (OBJECT)

OBJECTs are like a struct/class in C/C++ or a RECORD in pascal. Example:

```
OBJECT myobj
  a:LONG
  b:CHAR
  c:INT
ENDOBJECT
```

This defines a data structure consisting of three elements. Syntax:

```
OBJECT <objname>
  <membername> [ : <type> ]           /* any number of these */
ENDOBJECT
```

where type is one of the following:

```
CHAR/INT/LONG/<object>
PTR TO CHAR/INT/LONG/<object>
ARRAY OF CHAR/INT/LONG/<object>
```

(ARRAY is short for ARRAY OF CHAR)

like DEF declarations, omitting the type means :LONG.

Note that <membername> need not be a unique identifier, it may be in other objects too. There are lots of ways to use objects:

```
DEF x:myobj                     /* x is a structure */
DEF y:PTR TO myobj              /* y is just a pointer to it */
DEF z[10]:ARRAY OF myobj

y:=[-1,"a",100]:myobj           /* typed lists */

IF y.b="a" THEN                 /* ... */

z[4].c:=z[d+1].b++
```

(see 4F and other parts of chapter 4 for these)

ARRAYs in objects are always rounded to even sizes, and put on even offsets:

```
OBJECT mystring
  len:CHAR, data[9]:ARRAY
ENDOBJECT
```

SIZEOF mystring is 12, and "data" starts at offset 2.

'PTR TO' is the only type in OBJECTs that may refer to yet undeclared other objects.

(see 14A for all other OBJECT features that are somehow OO related)

8G. initialisation

1. Always initialised to NIL (or else, if explicitly stated)
   - global variables
   - NOTE: for documentation purposes, it's always nicer if you write =NIL in the definitions of variables that you expect to be NIL.
2. Initialised to " and [] resp.
   - global and local STRINGs
   - global and local LISTs
3. Not initialised
   - local variables (unless explicitly stated)
   - global and local ARRAYs
   - global and local OBJECTs

8H. the essentials of the E typesystem

This section tries to explain how the E typesystem works from another perspective.

Most problems people have while programming in E stem from their incorrectview of how the E type-system works, Also, many people have an idea how types work from their previous programming language, and try to apply this to E, which is often fatal, because E is quite different when it come to types.

The Type System.
but E is in essence a TYPELESS language. Indeed, variables may have a type, but this is only used as a specification how to dereference a variable when it is used as a pointer. In almost ALL other language constructions, variables are treated as all being of the same type, namely the 32bit typeless value.

In practise this means that for example in expressions with the exception of the ".", "[]" and "++" operators etc., all operators and functions work on 32bit values, regardless of whether they represent booleans, integers, reals or pointers to something.

Pointer Types.

In the E type-system only 4 types exist, PTR TO CHAR, PTR TO INT, PTR TO LONG and PTR TO <object>, where <object> is a name of a previously defined OBJECT. When a variable (or an object member, as we'll see later) is declared as being of this type, It means that if the variable contains a value that is a legal pointer, this is how it should be dereferenced.

LONG, ARRAY etc.
All other types one may see in a DEF declaration are not really types, as they really are only other ways of writing one of the above four. As an example, ARRAY OF <type> is just another way of writing PTR TO <type>, with the only difference that the former is automatically assigned the address of an area of stackspace which is big enough to hold data for the #of elements specified in square brackets.

Here's a table that shows all E 'types' in terms of the basic four:

ARRAY OF CHAR, ARRAY, STRING, LONG	(are equal to)	PTR TO CHAR
ARRAY OF INT	(is equal to)	PTR TO INT
ARRAY OF LONG, LIST	(are equal to)	PTR TO LONG
ARRAY OF <object>, <object>	(are equal to)	PTR TO <object>

- LONG is for variables that are not intended to be used as a pointer, i.e integers. Its equivalence with PTR TO CHAR is quite logical, as conceptually both talk about things that are measured in units of 1. (for example, "++" has the same effect on both)
- LIST and STRING are the same as their ARRAY equivalents, in respect to the fact that they're initialised to a piece of stack-space, but their stack representation is a little more complex to facilitate runtime bounds-checking (when used with the correct functions).
- an <object> is equivalent to [1]:ARRAY OF <object>. both represent an initialised PTR TO <object>.

In an OBJECT one can have the same declarations, with the addition of CHAR and INT (similar to LONG), and the ommission of LIST and STRING, as these are complex objects in their own right, and cannot be part of an object.

Deferencing.
Given a pointer p of some type,

"[]" may index other elements that are sequentially ordered next to the element it is currently pointing to. note that this allows for both positive and negative indices, and also no assumptions are made about where and how many elements are actually allocated.

"++" sets the pointer to the next element in memory, "--" to the previous one. note that these operators always operate on the pointer and never on the the element the pointer is pointing to.

"." works similar to "[]", only now indexes the pointer by name, i.e. the pointer must be a PTR TO <object>.

"[]" and "." may be concatenated to a pointer p in any sequence, given the fact that the previous resulting value again is known to be of a "PTR TO" type.

One does not need to write out a de-reference in total, as in other languages, e.g. if p is an ARRAY OF obj, instead of having to write p[index].member you can write just p[index], which logically results in the address of that object. This also explains why p[].member is equivalent to p.member, since p[] is the same as p when it points to an object.

Reference Semantics.
Another type-related issue that makes E somewhat different from other languages and thus harder to grasp is it's accent on Reference Semantics rather than Value Semantics. I'll try to argue why that's good here.

Informally, Reference Semantics means that objects in a language (mostly other than the simple ones like LONGs) are represented by pointers, while Value Semantics treats these objects as just being themselves. An example of a language that has only Value Semantics is BASIC, examples of languages that have them both are the C/C++ and Pascal type-of languages, and examples of Reference only are newer Object Oriented languages, functional languages like LISP and of course E.

Using Reference Semantics doesn't mean being occupied with pointers all the time, rather you're worrying about them a lot less then in the mixed case or the Value-only case, especially since in real life programs most

non-trivial data-structures get allocated dynamically which implies pointers. The best example of this is LISP, where one programs heavily with pointers without noticing. In E, one could easily forget STRING is a pointer, given the easy by which one can pass it around to other functions; in C often lots of "&" are needed where in the equivalent E case none are, and the Oberon equivalent of bla('hallo') looks like bla(sys.ADR('hallo')) because the string doesn't represent a pointer, but a value as a whole...

9. BUILT-IN FUNCTIONS

9A. io functions

```
WriteF(formatstring,args,...)
PrintF(formatstring,args,...)
```

prints a string (which may contain formatting codes) to stdout. Zero to unlimited arguments may be added. Note that, as formatstrings may be created dynamically, no check on the correct number of arguments is (can be) made. Examples:

```
WriteF('Hello, World!\n')       /* just write a lf terminated string */

WriteF('a = \d \n',a)           /* writes: "a = 123", if a was 123 */
```

(see 2F about strings for more).
NOTE: if stdout=NIL, for example if your program was started from the Workbench, WriteF() will create an output window, and put the handle in conout and stdout. This window will automatically be closed on exit of the program, after the user typed a <return>. WriteF() is the only function that will open this window, so if you want to do IO on stdout, and want to be sure stdout<>NIL, perform a "WriteF(' ')" as first instruction of your program to ensure output. If you want to open a console window yourself, you may do so by placing the resulting file handle in the 'stdout' and 'conout' variables, as your window will then be closed automatically upon exit. If you wish to close this window manually, make sure to set 'conout' back to NIL, to signal E that there's no console window to be closed. PrintF() is the same as WriteF only uses the v37+ buffered IO. both return the length of the string that was printed.

```
Out(filehandle,char)
```

and

```
char:=Inp(filehandle)
```

Either write or read one single byte to some file or stdout if char=-1 then an EOF was reached, or an error occurred. Out returns the number of bytes actually written (<>1 is error).

```
len:=FileLength(namestring)
```

lets you determine the length of a file you *may* wish to load, and also, if it exists (returns -1 upon error/file not found).

```
ok:=ReadStr(filehandle,estring)
```

(see 9B)

```
oldout:=SetStdOut(newstdout)
oldin:=SetStdIn(newstdin)
```

Sets the standard output variable 'stdout'. Equivalent for: oldout:=stdout; stdout:=newstdout. Same goes for the stdin variable.

9B. strings and string functions

E has a datatype STRING. This is a string, from now on called 'Estring', that may be modified and changed in size, as opposed to normal 'strings', which will be used here for any zero-terminated sequence. Estrings are downward compatible with strings, but not the other way around, so if an argument requests a normal string, it can be either of them. If an Estring is requested, don't use normal strings. Example of usage:

```
DEF s[80]:STRING, n             -> s is an estring with a maxlen of 80
ReadStr(stdout,s)               -> read input from the console
n:=Val(s)                       -> get a number out of it
-> etc.
```

Note that all string functions will handle cases where string tends to get longer than the maximum length correctly:

```
DEF s[5]:STRING
StrAdd(s,'this string is longer than 5 characters',ALL)
```

s will contain just 'this '.
A string may also be allocated dynamically from system memory with the function String(), (note: the pointer returned from this function must always be checked against NIL)

```
s:=String(maxlen)
```

```
DEF s[80]:STRING
```

is equivalent to

```
DEF s
```

and

```
s:=String(10)
```

```
bool:=StrCmp(string,string,len=ALL)
```

compares two strings. len must be the number of bytes to compare, or 'ALL' if the full length is to be compared. Returns TRUE or FALSE (len is a default argument (see 6F))

```
StrCopy(estring,string,len=ALL)
```

copies the string into the estring. If len=ALL, all will be copied. returns the estring.

```
StrAdd(estring,string,len=ALL)
```

same as StrCopy(), only now the string is concatenated to the end. returns the estring.

```
len:=StrLen(string)
```

calculates the length of any zero-terminated string

```
len:=EstrLen(estring)
```

returns the length of an estring

```
max:=StrMax(estring)
```

returns the maximum length of a estring

```
StringF(estring,fmtstring,args,...)
```

similar to WriteF, only now output goes to estring instead of stdout.
example:

```
StringF(s,'result: \d\n',123)
```

's' will be 'result: 123\n'
returns the estring, and length as second returnvalue.

```
RightStr(estring,estring,n)
```

fills estring with the last n characters of the second estring
returns the estring.

```
MidStr(estring,string,pos,len=ALL)
```

copies any number of characters (including all if len=ALL) from position pos in string to estring. NOTEZ BIEN: in all string related functions where a position in a string is used, the first character in a string has position 0, not 1, as is common in languages like BASIC.
returns the estring.

```
value,read:=Val(string,read=NIL)
```

finds an integer encoded in ascii out of a string. Leading spaces/tabs etc. will be skipped, and also hexadecimal numbers (1234567890ABCDEFabcdef) and binary numbers (01) may be read this way if they are preceded by a "$" or a "%" sign respectively. A minus "-" may indicate a negative integer. Val() returns the number of characters read in the second argument, which must be given by reference (<-!!!), or can be received as second returnvalue. If "read" returns 0 (value will be 0 too) then the string did not contain an integer, or the value was too sizy to fit in 32bit. "read" may be NIL.

examples of strings that would be parsed correctly:

```
'-12345', '%10101010', '   -$ABcd12'
```

these would return both as "value" and in read a 0:

```
'', 'hello!'
```

```
foundpos:=InStr(string1,string2,startpos=0)
```

searches string1 for the occurrence of string2, possibly starting from another position than 0. Returned is the offset at which the substring was found, else -1.

```
newstringadr:=TrimStr(string)
```

returns the *address* of the first character in a string, i.e., after leading spaces, tabs etc.

```
UpperStr(string)
```

and

```
LowerStr(string)
```

changes the case of a string.
TAKE NOTE: these functions modify the contents of 'string', so they may only be used on estrings, and strings that are part of your programs data. Effectively this means that if you obtain the address of a string through some amiga-system function, you must first StrCopy() it to a string of your program, then use these functions. they return the string.

```
ok:=ReadStr(filehandle,estring)
```

will read a string (ending in ascii 10) from any file or stdout. ok contains -1 if an error occurred, or an EOF was reached. Note: the contents of the string read so far is still valid. Also note that, like Inp() Out(), etc. this function makes use of unbuffered 1.3 style IO, and thus may be slow. The dos.library Fgets() function forms a nice alternative.

```
SetStr(estring,newlen)
```

manually sets the length of a string. This is only handy when you read data into the estring by a function other then an E string function, and want to continue using it as an Estring. For example, after using a function that just puts a zero-terminated string at the address of estring, use SetStr(mystr,StrLen(mystr)) to make it manipulatable again. If the string is too long, SetStr does nothing (this should always be prevented).

```
AstrCopy(string1,string2,size)
```

'Array String Copy' copies string2 into the memory area denoted by string1. string1 is typically not an estring but an ARRAY. size is the total #of chars string1 can hold, i.e. if you write 5 and string2='helloworld', string1 will be 'hell' + 0termination.

```
order:=OstrCmp(string1,string2,max=ALL)
```

'Ordered String Compare' returns 1 if string2>string1, 0 for equal and -1 for less. only max chars are compared.

for string linking functions (see 9H)

9C. lists and list functions

Lists are like strings, only they consist of LONGs, not CHARs. They may also be allocated either global, local or dynamic:

```
DEF mylist[100]:LIST                  /* local or global */
DEF a
a:=List(10)                           /* dynamic */
```

(note that in the latter case, pointer 'a' may contain NIL)
Just as strings may be represented as constants in expressions, lists have their constant equivalent:

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

The value of such an expression is a pointer to an already initialised list. Special feature is that they may have dynamic parts, i.e, which will be filled in at runtime:

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

moreover, lists may have some other type than the default LONG, like:

```
[1,2,3]:INT
[65,66,67,0]:CHAR                    /* equivalent with   'ABC'   */
['topaz.font',8,0,0]:textattr
OpenScreenTagList(NIL,[SA_TITLE,'MyScreen',TAG_DONE])
```

As shown in the latter examples, lists are extremely useful with system functions: they are downward compatible with an ARRAY OF LONG, and object-typed ones can be used wherever a system function needs a pointer to some structure, or an array of those. Taglists and vararg functions may also be used this way. NOTEZ BIEN: all list functions only work with LONG lists, typed-lists are only convenient in building complex data structures and expressions.

As with strings, a certain hierarchy holds:
list variables -> constant lists -> array of long/ptr to long
When a function needs an array of long you might just as well give a list as argument, but when a function needs a listvar, or a constant list, then an array of long won't do.

It's important that one understands the power of lists and in particular typed-lists: these can save you lots of trouble when building just about any data-structure. Try to use these lists in your own programs, and see what function they have in the example-programs.
summary:

[<item>,<item>,...]	immediate list (of LONGs, use with listfuncs)
[<item>,<item>,...]:<type>	typed list (just to build data structures)

If <type> is a simple type like INT or CHAR, you'll just have the initialised equivalent of ARRAY OF <type>, if <type> is an object-name, you'll be building initialised objects, or ARRAY OF <object>, depending on the length of the list.

If you write [1,2,3]:INT you'll create a data structure of 6 bytes, of 3 16bit values to be precise. The value of this expression then is a pointer to that memory area. Same works if, for example, you have an object like:

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

writing [1,2,3]:myobject would then mean creating a data structure in memory of 8 bytes, with the first four bytes being a LONG with value 1, the following byte a CHAR with value 2, then a pad byte, and the last two bytes an INT (2 bytes) with value 3. you could also write:

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

you would be creating an ARRAY OF myobject with size 3. Note that such lists don't have to be complete (3,6,9 and so on elements), you may create partial objects with lists of any size

One last note on data size: on the amiga, you may rely on the fact that a structure like 'myobject' has size 8, and that it has a pad byte to have word (16bit) alignment. It is however very likely that an E-compiler for 80x86 architectures will not use the pad byte and make it a 7byte structure, and that an E-compiler for a sun-sparc architecture (if I'm not mistaken) will try to align on 32bit boundaries, thus make it a 10 or 12 byte structure. Some microprocessors (they are rare, but they exist) even use (36:18:9) as numbers of bits for their types (LONG:INT:CHAR), instead of (32:16:8) as we're used to. So don't make too great an assumption on the structure of OBJECTs and LISTs if you want to write code that stands a chance of being portable or doesn't rely on side effects.

```
ListCopy(listvar,list,num=ALL)
```

Copies num elements from list to listvar. example:

```
DEF mylist[10]:LIST
```

```
ListCopy(mylist,[1,2,3,4,5],ALL)
```

returns listvar.

```
ListAdd(listvar,list,num=ALL)
```

Copies num items of list to the tail of listvar. returns listvar.

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

Compares two lists, or some part of them.

```
len:=ListLen(list)
```

Returns length of list, like ListLen([a,b,c]) would return 3

```
max:=ListMax(listvar)
```

returns maximum possible length of a listvar.

```
value:=ListItem(list,index)
```

functions as value:=list[index] with the difference that list may also be a constant value instead of a pointer. This is very useful in situations like this where we directly want to use a list of values:

```
WriteF(ListItem(['ok!','no mem!','no file!'],error))
```

this prints an errormessage according to "error". it's similar to:

```
DEF dummy:PTR TO LONG
dummy:=['ok!','no mem!','no file!']
WriteF(dummy[error])
```

```
SetList(listvar,newlen)
```

manually sets the length of a list. This will only be useful when you read data into the list by a function other then a list-specific function, and want to continue using it as a true list.

for list functions that make use of quoted expressions (see 11C).
for list linking functions (see 9H).

9D. intuition support functions

```
wptr:=OpenW(x,y,width,height,IDCMP,wflags,title,screen,sflags,gadgets,taglist=NIL)
```

creates a window where wflags are flags for window layout (like BACKDROP, SIMPLEREFRESH e.d, usually $F) and sflags are for specifying the type of screen to open on (1=wb,15=custom). screen must only be valid if sflags=15, else NIL will do. gadgets may point to a glist structure, which you can easily create with the Gadget() function, else NIL.

```
CloseW(wptr)
```

closes that screen again. Only difference from CloseWindow() is that it accepts NIL-pointers and sets stdrast back to NIL.

```
sptr:=OpenS(width,height,depth,sflags,title,taglist=NIL)
```

opens a custom screen for you. depth is number of bitplanes (1-6, 1-8 AGA).

```
CloseS(sptr)
```

as CloseW(), now for screens.

```
nextbuffer:=Gadget(buffer,glist,id,flags,x,y,width,string)
```

[warning: this function is a bit out of date]
This function creates a list of gadgets, which can then be put in your window by giving them as an argument to OpenW(), or afterwards with intuition functions like AddGlist(). buffer is mostly an ARRAY of at least GADGETSIZE bytes to hold all the structures associated with one gadget, id is any number that may help you remember which gadget was pressed when an IntuiMessage arrives. Flags are: 0=normal gadget, 1=boolean gadget, 3=boolean gadget that is selected. Width is width in pixels, that should be large enough to hold the

string, which will be auto-centered. glist should be NIL for the first gadget, and glistvar for all others, so E may link all gadgets. The function returns a pointer to the next buffer (=buffer+GADGETSIZE). Example for three gadgets:

```
CONST MAXGADGETS=GADGETSIZE*3

DEF buf[MAXGADGETS]:ARRAY, next, wptr

next:=Gadget(buf,NIL,1,0,10,20,80,'bla')     /* the 1st gadget */
next:=Gadget(next,buf,... )
next:=Gadget(next,buf,... )                  /* any amount linked 2 1st */

wptr:=OpenW( ...,buf)

code:=Mouse()
```

gives you the current state of all 2 or 3 mouse buttons; left=1, right=2 and middle=4. If for example code=3 then left and right were pressed together.
NOTEZ BIEN: this is not a real intuition function, if you want to know about mouse-events the proper way, you'll have to check the intuimessages that your window receives. This is the only E function that directly checks the hardware, and thus only useful in demo-like programs and for testing. (DO NOT USE THIS FUNCTION IN PROGRAMS THAT ARE SUPPOSED TO WORK UNDER THE OS)

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

intuition Mouse() alternatives for programs that only want to test for a mouseclick.

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

enables you to read the mouse coordinates. win is the window they need to be relative to.

```
class:=WaitIMessage(window)
```

This function makes it easier to just wait for a window event. It simply waits until a intuimessage arrives, and returns the class of the event. It stores other variables like code and qualifiers as private global variables, for access with functions described below. WaitIMessage() represents the following code:

```
PROC waitimessage(win:PTR TO window)
  DEF port,mes:PTR TO intuimessage,class,code,qual,iaddr
  port:=win.userport
  IF (mes:=GetMsg(port))=NIL
    REPEAT
      WaitPort(port)
    UNTIL (mes:=GetMsg(port))<>NIL
  ENDIF
  class:=mes.class
  code:=mes.code             /* stored internally */
  qual:=mes.qualifier
  iaddr:=mes.iaddress
  ReplyMsg(mes)
ENDPROC class
```

as you see, it gets exactly one message, and does not forget about multiple messages arriving in one event, if called more than once. For example, say you opened a window that displays something and just waits for a closegadget (you specified IDCMP_CLOSEWINDOW only):

```
WaitIMessage(mywindow)
```

or, you have a program that waits for more types of events, handles them in a loop, and ends on a closewindow event:

```
WHILE (class:=WaitIMessage(win))<>IDCMP_CLOSEWINDOW
  /* handle other classes */
ENDWHILE

code:=MsgCode()
qual:=MsgQualifier()
iaddr:=MsgIaddr()
```

These all supply you with the private global variables as mentioned before. the values returned are all defined by the most recent call to WaitIMessage(). Example:

```
IF class:=IDCMP_GADGETUP
  mygadget:=MsgIaddr()
  IF mygadget.userdata=1 THEN  /* ... user pressed gadget #1 */
```

```
ENDIF
```

9E. graphics support functions

All graphics support functions that do not explicitly ask for a rastport, make use of the system-variable 'stdrast'. It is automatically defined by the last call to OpenW() or OpenS(), and is set to NIL by CloseW() and CloseS(). Calling these routines while stdrast is still NIL is legal. stdrast may be manually set by SetStdRast() or stdrast:=myrast

```
Plot(x,y,colour=1)
```

Draws a single dot on your screen/window in one of the colours available. colour ranges from 0-255, or 0-31 on pre-AGA machines.

```
Line(x1,y1,x2,y2,colour=1)
```

Draws a line

```
Box(x1,y1,x2,y2,colour=1)
```

Draws a box

```
Colour(foreground,background=0)
```

sets the colours for all graphics functions (from the library) that do not take a colour as argument. This is the colour *register* (i.e 0-31) and not colour *value*
NOTE: functions that have "colour" as an argument, change the Apen of stdrast.

```
TextF(x,y,formatstring,args,...)
```

exactly the same function as WriteF(), only outputs to some (x,y) on your stdrast, instead of stdout. (see 9A, WriteF() and strings in the language reference). returns the length of the resulting string.

```
oldrast:=SetStdRast(newrast)
```

changes the output rastport of the E graphics functions

```
SetTopaz(size=8)
```

lets you set the font of the rastport "stdrast" to topaz, just to be sure that some custom system font of the user won't screw up your window layout. size is of course 8 or 9. Only to be used as last resort if you can't support font-sensitivity.

```
SetColour(screen,colourreg,r,g,b)
```

set colour register (0..255) of screen to certain RGB values. each rgb value has a range of 0..255, i.e. 24bit colour. this function will automatically rescale to 12bit colour if no AGA is present, and also use the correct function.

9F. system support functions

```
bool:=KickVersion(vers)
```

Will give TRUE if the kickstart in the machine your program is running on is equal or higher than vers, else FALSE

```
mem:=New(n)
```

This dynamically creates an array (or memory area, if you wish) of n bytes. Difference with AllocMem() is that it is called automatically with flags $10000 (i.e cleared mem, any type) and that no calls to Dispose() are necessary, as it is linked to a memory list that is automatically de-allocated upon exit of your program.
(see 4K, also)

```
mem:=NewR(n)
```

same as New(), only now automatically raises the "MEM" exception instead of return if no memory could be allocated.

```
mem:=NewM(n,flags)
```

same as NewR(), but also allows you to specify flags (MEMF_CHIP etc.)

```
Dispose(mem)
```

Frees any mem allocated by New(). You only have to use this function if you explicitly wish to free memory _during_ your program, as all is freed at the end anyway.

```
CleanUp(returnvalue=0)
```

Exits the program from any point. It is the replacement for the DOS call Exit(): never use it! instead use CleanUp(), which allows for the deallocation of memory, closing libraries correctly etc. The return value will be given to dos as returncode. CleanUp() is ONLY necessary if you have to exit at a point different from ENDPROC in main.

```
amount:=FreeStack()
```

returns the amount of free stack space left. This should always be 1000 or more. (see 16C on how E organizes its stack. If you don't do heavy recursion, you need not worry about your free stack space.

```
bool:=CtrlC()
```

Returns TRUE if Ctrl-C was pressed since you last checked, else FALSE. This only works for programs running on a console, i.e. cli-programs.

Example of how these last three functions may be used:

```
/* calculate faculty from command-line argument */

OPT STACK=100000

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

PROC fac(n)
  DEF r
  IF FreeStack()<1000 OR CtrlC() THEN CleanUp(5)    /* xtra check */
  IF n=1 THEN r:=1 ELSE r:=fac(n-1)*n
ENDPROC r
```

Of course, this recursion will hardly run out of stack space, and when it does, it's halted by FreeStack() so fast you won't have time to press CtrlC, but it's the idea that counts here.
A definition of fac(n) like:

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

would be less safe.

```
mem:=FastNew(size)
FastDispose(mem,size)
FastDisposeList(list)
```

FastNew() and FastDispose() are replacements for NewR(size) and Dispose(ptr) (they are used by NEW and END). this is what they have in common:

- "NEW" exceptions may be Raised
- memory is always cleared
- auto-dealloc at end of program

but the following differences should be noted (positive):

- they are varying from 10 to 50 times faster (!)
- they use way less memory for small objects
- they do not fragment memory

[all this is for objects <=256 bytes, for bigger ones NewR() and Dispose() are used].
negative:

- they do not free mem, but recycle it.
- FastDispose() needs exact size of allocation. END also.

List allocated with NEW need the function FastDisposeList(). Because Lists have a length, the size parameter isn't needed.

9G. math and other functions

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

These work with the usual operations, boolean as well as arithmetic. Note that for And() and Or() an operator exists.

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

Performs the same operation as the '*' and '/' operators, but now in full 32bit. For speed reasons, normal operations are 16bit*16bit=32bit and 32bit/16bit=16bit. This is sufficient for nearly all calculations, and where it's not, you may use Mul() and Div(). NOTE: in the Div case, a is divided by b, not b by a.

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

Return TRUE or FALSE if some expression is Odd or Even

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

compute min and max of the two given ints.

```
randnum:=Rnd(max)
seed:=RndQ(seed)
```

Rnd() computes a random number from an internal seed in range 0 .. max-1. For example, Rnd(1000) returns an integer from 0..999 To initialise the internal seed, call Rnd() with a negative value; the Abs() of that value will be the initial seed.

RndQ() computes a random number "Q"uicker than Rnd() does, but returns only full range 32bit random numbers. Use the result as the seed for the next call, and for startseed, use any large value, like $A6F87EC1

```
absvalue:=Abs(value)
```

computes the absolute value.

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

computes the sign of v, i.e. returns -1,0,1

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

Divides 32bit c by 16bit d and returns 16bit modulo a and optionally a 16bit result of the division b. If these 16bit limits are exceeded, Mod() will not give correct results.

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

shifts y num bits to left or right (set new bits to 0).

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

peeks into memory at some address, and returns the value found. This works with 32, 16 and 8 bit values respectively. Note that the compiler does not check if 'adr' is valid. These functions are available in E for those cases where reading and writing in memory with PTR TO <type> would only make a program more complex or less efficient. You are _not_ encouraged to use these functions.

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

Pokes value 'a' into memory. See: Long()

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

makes sure x lies between bounds a and b, and adjust acordingly if necessary. It equals:

```
y:=IF x<a THEN a ELSE IF x>b THEN b ELSE x
```

9H. string and list linking functions

E provides for a set of functions that allows the creation of linked list with the STRING and LIST datatype, or strings and lists that were created with String() and List() respectively. As you may know by now, strings and lists, complex datatypes, are pointers to their respective data, and have extra fields to a negative offset of that pointer specifying their current length and maxlength. the offsets of these fields are PRIVATE. as an addition to those two, any complex datatype has a 'next' field, which is set to NIL by default, which may be used to build linked list of strings, for example. in the following, I will use 'complex' to denote a ptr to a STRING or LIST, and 'tail' to denote another such pointer, or one that already has other strings linked to it. 'tail' may also be a NIL pointer, denoting the end of a linked list. [note: these String-list functions have nothing to do with E-lists or Lisp-Cell lists]
The following functions may be used:

```
complex:=Link(complex,tail)
```

puts the value tail into the 'next' field of complex. returns again complex.
example:

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

creates a linked list like: s --> t --> NIL

```
tail:=Next(complex)
```

reads the 'next' field of var complex. this may of course be NIL, or a complete linked list. Calling Next(NIL) will result in NIL, so it's safe to call Next when you're not sure if you're at the end of a linked list.

```
tail:=Forward(complex,num)
```

same as Next(), only goes forward num links, instead of one, thus:

```
Next(c) = Forward(c,1)
```

You may safely call Forward() with a num that is way too large; Forward will stop if it encounters NIL while searching links, and will return NIL.

```
DisposeLink(complex)
```

same as Dispose(), with two differences: it's only for strings and lists allocated with String() or List(), and will automatically de-allocate the tail of complex too. Note that large linked lists containing strings allocated with String() as well as some allocated locally or globally with STRING may also be de-allocated this way.

For a good example of how linked lists of strings may be put to good use in real-life programs, see Src/Utils/D.e

9I. lisp-cells and cell functions

yep. that's right. you thought LISP was fun, then try E now. [or: the story about why E is a better LISP than LISP itself :-)]

Starting from v3, E has the cell datatype, almost identical to cells in the LISP language. more technically, E has:

'Conservative Mark and Sweep Garbage-Collected Lisp-Cells'

basically this amounts to being able to allocate LISP-cells, which are pairs of two values:

```
x:=<a|b>
```

which is much like NEW [a,b]:LONG, only now E will automatically deallocate the 8 bytes in question itself when it finds out it needs memory and no pointers are pointing to the cell. In practise this means that you can

freely have functions create cells as temporaries, without worrying about freeing them. And any LISP-programmer will be able to explain to you that with cells you can build any data-structure (most notably trees and lists). [note: this text does not thorougly explain how to make full use of cells, since dozens of LISP books have been written about this]

Selecting the values can easily be done using Car(x) and Cdr(x), two LISP-functions which select head and tail (first and second) element of the cell. if x is a PTR TO LONG, even x[0] and x[1] are allowed.

One can also write lists of cells:

```
<a,b,c>
```

(note the commas) as short for

```
<a|<b|<c|NIL>>>
```

An alternative for selection with Car/Cdr is E's unification:

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

instead of:

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

lisp-cell unification resembles E-list unification (see 4L). for example:

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

equals:

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

A lisp-nil value is available "<>", which equals NIL and 0.

some functions are available (note that Cons() is _only_ available through <...>)

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

fix the head and the tail value of a cell c

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

gives a bool for whether or not c points at a cell, so Cell(<1>)=TRUE, and Cell(3.14)=FALSE. This is not a fast function.

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

tell you about the amount of free cells available. very slow function. there should be no need to call this function other than curiosity.

```
SetChunkSize(k)
```

Sets the chunksize to allocate for cells to k kilobyte. default is 128k. This function can only be called once, and only before the first cons (<..>) allocation takes place. Thereafter it has no effect.

In general, get a good book about lisp to understand more about programming with cells.

One can write any LISP-functions in E, with exactly the same functionality:

```
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
```

using a destructive implementation for functions like these is also allowed.

techy stuff:

E's garbage collector implements a conservative mark and sweep algorithm that was tested to be 5 to 25 times faster than several logical and functional language implementations on the Amiga. Conservative means that in case of doubt, the GC will not deallocate a cell. this is necessary since in a typeless language such as E, the GC can easily bump into a value that is not a valid pointer etc.
The GC allocates big chunks (default 128k), in which it allocates cells. if out of cells, it will collect garbage by scanning the stack and registers for pointers into the cellspace, and recursively mark them. after that, all unmarked cells are reused, and if the gain after a collection was only small, a new chunk is allocated (if this fails, "NEW" is raised).

interaction with other E values:

- storing other values in cells is no problem whatsoever. objects, strings, floats, anything can be put into a cell without confusing the GC too much.
- storing cells in other values, for example a ptr to a cell in a dynamic object, is problematic, since the GC won't be able to find it there. a solution for this will be provided. However I think this case will seldom occur. ptrs to cells can safely be stored in global and local variables, even registers, and any stack datastructures. [and most importantly, in other cells!]

caveats:

- The GC currently can't collect cells that have a Car-list >1000 long or so, i.e. <<<NIL:a>:b>:c>, but then 1000 instead of 3 entries. this will hardly ever occur since lists like this are usually formed as Cdr-lists, which the GC can handle into infinity. (it will raise "STCK" if this fails)
- inline assembly code should never push stuff on the stack that is not LONG aligned. this was already necessary in v2.1b, but now is even more essential.

There's a trade off in chunk-size between time and space. Allocating small chunks obviously is nice since you won't waste any memory, however, when collecting garbage, the effort for each pointer to trace is almost proportional to the number of spaces. therefore:

- if speed is most important tune the chunkspacesize such that that only one space is needed. if the top cell-memory usage at a certain time is 50k, a chunkspace of 100k or 150k will give optimal performance.
- if memory usage is more important, in the example above a chunksize of 20k or 30k will be quite optimal for memory.

In general, time a heavy usage of your cell-algorithm with different sizes to see what trade-off suits you best.

10. LIBRARY FUNCTIONS AND MODULES

10A. built-in library calls

As you may have noticed from previous sections, the piece of code automatically linked to the start of your code, called the "initialisation code", always opens the three libraries intuition, dos, graphics and (and sometimes mathieeesingbas), and because of this, the compiler has all the calls to those four libraries (including exec) integrated in the compiler (there are a few hundred of them). These are up to AmigaDos v3.00 (v39). To call Open() from the dos library, simply say:

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

or AddDisplayInfo() from the graphics library:

```
AddDisplayInfo(mydispinfo)
```

it's as simple as that.

10B. interfacing to the amiga system with the v39 modules

To use any other library than the five in the previous section, you'll need to resort to modules. Also, if you wish to use some OBJECT or CONST definition from the Amiga includes as is usual in C or assembler, you'll need modules. Modules are binary files which may include constant, object, library and function (code) definitions. The fact that they're binary has the advantage over ascii (as in C and assembly), that they need not be compiled over and over again, each time your program is compiled. The disadvantage is that they cannot simply be viewed; they need a utility like ShowModule (see 17A) to make their contents visible. The modules that contain the library definitions (i.e the calls) are in the root of emodules: (the modules dir in the distribution), the constant/object definitions are in the subdirectories, structured just like the originals from Commodore.

MODULE

syntax:

```
     MODULE <modulenames>,...
```

Loads a module. A module is a binary file containing information on libraries, constants, and sometimes functions. Using modules enables you to use libraries and functions previously unknown to the compiler.

Now for an example, below is a short version of the source/examples/asldemo.e source that uses modules to put up a filerequester from the 2.0 Asl.library.

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

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

From the modules 'asl', the compiler takes asl-function definitions like RequestFile(), and the global variable 'aslbase', which only needs to be initialised by the programmer. From 'libraries/Asl', it takes the definition of the filerequester object, which we use to read the file the user picked. Well, that wasn't all that hard: did you think it was that easy to program a filerequester in E?

10C. compiling own modules

with v3, you can gather all PROCs, CONSTs, OBJECTs and to some extent also global variables that you feel somehow belong together in one source, write "OPT MODULE" to signal EC that this is supposed to be a module, and then compile all to a .m file to be used in the main program, just like you used to do with the old modules.

by default, all elements in a module are PRIVATE, i.e. not accessable to the code that imports the .m file. to show which elememts you wish to be visible in the module, simply write EXPORT before it:

```
EXPORT ENUM TESTING,ONE,TWO,THREE,FOUR

EXPORT DEF important_glob_var, bla:PTR TO x

EXPORT OBJECT x
  next, index, term
ENDOBJECT

EXPORT PROC burp()
  /* whatever */
ENDPROC
```

"EXPORT" is useful in making a distinction between private and public, especially when all functions of an OBJECT can be accessed via PROCs, you may wish to keep OBJECT private as an effective method of data hiding. more on this topic, (see 14C)
[EXPORT can be written on any line, doesn't affect everything though.]

If in a module _all_ elememts need to be exported (for example one with only constants), a 'OPT EXPORT' will export all, without the need for individual EXPORT keywords.

global variables require extra attention:

- try to avoid lots of global variables. having lot of globals in modules makes projects messy and error-prone
- globs in a module cannot have initialisations directly in the DEF statement (reason for this will become clear below). for example:
  `DEF a` not `DEF a=1`
  `DEF a:PTR TO x` not `DEF a[10]:ARRAY OF x`
- globals in a module which are not exported function as local for the module, i.e. they'll never clash with globals in other modules. those who _are_ exported though, are combined with the others, i.e. if in both the main program and in a module a variable with the same name are used, this will be one and the same variable. that's why one can write DEF a[10]:ARRAY OF x in the main program, and EXPORT DEF a:PTR TO x in the module, to share the array. Also, if both use for example 'gadtools.m', only one of the two needs to initialise 'gadtoolsbase' for both to be able to make calls to the library. If you do not want librarybases to be shared (i.e. you want to have a local, private library base), simply redeclare it in a DEF in the module that is not EXPORTed. If you export a variable in a general purpose module, make sure to give it a pretty unique name.
- using globals in modules which provide general purpose datatypes needs special attention, as the module may be in use from more than one other module, in which case it may be unclear who is responsable for resources. take good care of this.

Using modules in modules

This requires little extra attention. If the module (B) you include in your own module (A) is one that only declares CONSTs, LIBRARYs and OBJECTs (without code) nothing special happens, however if B includes PROCs, then it's obvious this code needs to be linked later to the main program when A is linked. Therefore if a main program uses A, B will need to be present for compilation. The fact that A needs B is stored in A, and can be viewed with ShowModule. This chain of uses may grow out to a tree of dependencies, which has the result that if you use just one module in your program, a lot of others are automatically linked to it. Thus, E's module system automatically keeps track of dependancies that other languages need makefiles for. EC also allows for circular inclusions, and loads/links modules at most once (i.e. doesn't link unused modules). One thing E's module system doesn't automatically do for you is recompile dependant modules. If you change B, it is often necessary to recompile A too, since it might be referring to offsets etc. in the old version of B, which might cause code to crash. If this gets too complex in your project, use a utility such as E-Build (see 17I).

Try out the new ShowModule (see 17A) to see what EC puts in modules.

Including modules from other directories.

By default, a module name is prefixed by 'emodules:' to obtain the actual file. Now you can prefix the name with a '*' to denote the directory the source is in, so:

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

if this statement would be in source 'WORK:E/burp.e', these two modules would be 'emodules:bla.m' and 'WORK:E/bla.m'.

This is naturally the way to include components of your app into other parts. If you write modules that you use in many of your programs it would be handy to store them in the emodules hierarchy, and the place for this is the 'emodules:other/' dir.

10D. the modulecache

(see 17D, ShowCache/FlushCache about this).

11. QUOTED EXPRESSIONS

11A. quoting and scope

Quoted expressions start with a backquote. The value of a quoted expression is not the result from the computation of the expression, but the address of the code. This result may then be passed on as a normal variable, or as an argument to certain functions. example:

```
myfunc:=`x*x*x
```

myfunc is now a pointer to a 'function' that computes x^3 when evaluated. These pointers to functions are very different from normal PROCs, and you should never mix the two up. The biggest differences are that a quoted expression is just a simple expression, and thus cannot have ist own local variables. In our example, "x" is just a local or global variable. That's where we have to be cautious: if we evaluate myfunc somewhat later in the same PROC, x may be local, but if myfunc is given as parameter to another PROC, and then evaluated, x needs of course to be global. There's no scope checking on this.

11B. Eval()

```
Eval(func)
```

simply evaluates a quoted expression (exp = Eval(`exp)).

NOTE: because E is a somewhat typeless language, accidentally writing "Eval(x*x)" instead of "Eval(`x*x)" will go unnoticed by the compiler, and will give you big runtime problems: the value of x*x will be used as a pointer to code.

To understand why 'quoted expressions' is a powerful feature think of the following cases: if you were to perform a set of actions on a set of different variables, you'd normally write a function, and call that function with different arguments. But what happens when the element that you want to give as argument is a piece of code? in traditional languages this would not be possible, so you would have to 'copy' the blocks of code representing your function, and put the expression in it. Not in E. say you wanted to write a program that times the execution time of different expressions. In E you would simply write:

```
PROC timing(func,title)
  /* do all sorts of things to initialise time */
  Eval(func)
  /* and the rest */
  WriteF('time measured for \s was \d\n',title,t)
ENDPROC
```

and then call it with:

```
timing(`x*x*x,'multiplication')
timing(`sizycalc(),'large calculation')
```

in any other imperative language, you would have to write out copies of timing() for every call to it, or you would have to put each expression in a separate function. This is just a simple example: think about what you could do with data structures (LISTs) filled with unevaluated code:

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

Note that this idea of functions as normal variables/values is not new in E, quoted expressions are literally from LISP, which also has the somewhat more powerful so-called Lambda function, which can also be given as argument to functions; E's quoted expressions can also be seen as parameterless (or global parameter only) lambdas.

11C. built-in functions

```
MapList(varadr,list,listvar,func)
```

performs some function on all elements of list and returns all results in listvar. func must be a quoted expression (see 11A), and var (which ranges over the list) must be given by reference. Example:

```
MapList({x},[1,2,3,4,5],r,`x*x)
```
results r in: `[1,4,9,16,25]`

returns listvar.

```
ForAll(varadr,list,func)
```

Returns TRUE if for all elements in the list the function (quoted expression) evaluates to TRUE, else FALSE. May also be used to perform a certain function for all elements of a list:

```
ForAll({x},['one','two','three'],`WriteF('example: \s\n',x))
```

```
Exists(varadr,list,func)
```

As ForAll(), only this one returns TRUE if for any element the function evaluates to TRUE (<>0). note that ForAll() always evaluates all elements, but Exists() possibly does not.

```
SelectList(v,list,listvar,quotedexp)
```

Much like MapList(), only now doesn't store the result from quotedexp, it uses it as a boolean value, and only those values for which it is true are stored in listvar (which should be capable of holding the same amount of elements as list. example:

```
SelectList({x},[1,2,0,3,NIl],r,`x<>0)
```

results in r being [1,2,3].
returns length of listvar.

Example of how to use these functions in a practical fashion: we allocate different sizes of memory in one statement, check them all together at once, and free them all, but still only those that succeeded. (example is v37+)

```
PROC main()
  DEF mem[4]:LIST,x
  MapList({x},[200,80,10,2500],mem,`AllocVec(x,0))            -> alloc some
  WriteF(IF ForAll({x},mem,`x) THEN 'Yes!\n' ELSE 'No!\n')    -> suxxes ?
  ForAll({x},mem,`IF x THEN FreeVec(x) ELSE NIL)              -> free only those <>NIL
ENDPROC
```

Note the absence of iteration in this code. Just try to rewrite this example in any other language to see why this is special.

12. FLOATING POINT SUPPORT

REALs (or FLOATs, whatever) are very different in E than in other languages. This mainly has to do with the fact that E doesn't really discriminate between types of values. One is advised to understand this chapter _well_ before attempting to use floats.

12A. float values

In E, a float is just another 32bit value. The E compiler treats them just like integers or pointers, with the difference that their bit representation means something different. The E float format is the IEEEsingle standard.

A float value looks like an integer value with the exception that somewhere a "." is present. for example, the following are valid floats:

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

these aren't:

```
.    1234
```

(i.e. atleast one "." and one "0-9" char must be present).

You can use these values at almost all places where LONG values are legal, i.e. if you have have a function or datastructure that handles arbitrary LONG values, it will also handle floats.

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

12B. computing with floats

Because to E a float will seem like just another LONG, it will happily apply integer math to it when used in an expression, which is mostly not what you want. Also, one would like to be able to convert to integer and vice-versa. The float operator "!" handles all this.

assume in the following examples that a,b,c contain integer values, and x,y,z float values.

By default, an expression in E is considered an integer expression. what the "!" does when it occurs in an expression is the following:

- changes the expression from int to float. any operators following (+ * - / = <> > < >= <=) will be float operations. "!" may occur any number of times in an expression, changing from and to float again and again.
- the expression that did occur before the "!", if any, is converted to the appropriate type.

examples:

```
x:=a!
```

converts "a" to float, and stores the result in x. "a" is an integer exp, which is then toggled to float, which implies a conversion.

```
a:=!x!
```

converts "x" to integer and stores the result in a.

```
x:=y
```

```
x:=Ftan(y)
```

no "!" is needed here since no operator-math or conversions are necessary.

```
x:=!y*z
```

the "*" acts on y and z as floats, since "!" denotes the whole as a float-exp. the float result is stored in x

```
a:=b!*x+y!
```

a more complex example: the int "b" is converted to float, then x and y are float-multiplied and float-added to it. The result is converted to int and stored in the int "a"

```
x:=!y*z-z*y+(a!)+z/z

z:=!x*Fsin(!x*y)
```

all (+ * - /) are computed as float, and the int "a" is converted to float somewhere in the middle. since "(" ")" denotes a new expression, it has it's own status of "!". Same idea for the function below.

```
IF !x<0.1 THEN WriteF('Float value too small!\n')
```

as you can see, "!" also works on the six comparison operators.

12C. builtin float functions

Some trans-math functions are present, more will probably follow.

```
x:=Fsin(y)     x:=Fcos(y)     x:=Ftan(y)
```

usual sin() etc. functions. they work with radians.

```
x:=Fabs(y)
```

compute absolute value of y

```
x:=Ffloor(y)     x:=Fceil(y)
```

compute lowest and highest whole-number float value near y

```
x:=Fexp(y)     x:=Flog(y)     x:=Flog10(y)     x:=Fpow(y,z)     x:=Fsqrt(y)
```

compute e^y, ln(y), log base 10, z^y and square root of y, respectively.

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

parses string "s" to produce float value x. will skip leading spaces and tabs. n is the number of characters parsed from the start of the string, or 0 if it couldn't be parsed as a float value. "x" will then be 0.0. accepts negative numbers (and number without a ".", even).
example:

```
RealVal(' 3.14 ')     results in     3.14, 5
RealVal('blabla')     results in     0.0, 0
```

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

Format a float value x to the estring s, with n positions after the ".". max for "n" is 8, even less if you have lots of digits leading the ".". an "n" of 0 will denote no fraction. The string is returned as result, so it can be reused in a WriteF() for example:

```
WriteF('float = \s\n',RealF(s,3.14159),4)     results in     'float = 3.1416\n'
```

RealF() tries hard to make sensible roundings for a certain "n", as the example shows. negative numbers are also handled properly.

```
RealF(s,-3.14159,0)          results in     '-3'
```

12D. float implementation issues

As said before, the E float format is IEEE (single), this means that older float code using the FFP format with the SpXxx functions will have to be rewritten (as stated in the v2.1b docs). The mathffp library is no longer directly supported by EC v3.0, and you'll have to open this library like all others if you want to use it.
single IEEE's were chosen because:

- double IEEE's don't fit in a LONG
- the FFP format routines do not make use of a 68881 if present, the IEEE ones do. Furthermore the FFP format is incompatible with the 68881, which also uses IEEE format.
- IEEE is the worldwide float-format standard, which encourages data-file compatability among software/platforms.

E's float routines use the mathieeesingbas.library and the mathieeesingtrans.library, which are not by default supplied with the ancient v1.3 of the OS. This means that if you want to write under / support 1.3 AND you want to use the _builtin_ floats, you have to make sure these libraries are present (they seem to be available, maybe through commodore?).

Both EC and the programs it generates do not open these libraries as long as no float-features are used.

If all else fails, one can always use other floatlibraries to use floats with 1.3. I might recommend the tools/longreal.m module which uses doubles.

In the future, EC will probably allow mathieee library calls to be replaced with inline 68881 code transparently.

[note: v3.1 (v40) of the amiga operating system is known to contain a bug in the IEEE code. Be sure to run a SetPatch that fixes this]

13. EXCEPTION HANDLING

13A. defining exception handlers (HANDLE/EXCEPT)

The exception mechanism in E is basically the same as in ADA; it provides for flexible reaction on errors in your program and complex resource management. NOTE: the term 'exception' in E has very little to do with exceptions caused directly by 680x0 processors.

An exception handler is a piece of program code that will be invoked when runtime errors occurs, such as windows that fail to open or memory that is not available. You, or the runtime system itself, may signal that something is wrong (this is called "raising an exception"), and then the runtime-system will try and find the appropriate exception handler. I say "appropriate" because a program can have more than one exception handler, on all levels of a program. A normal function definition may (as we all know) look like this:

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

a function with an exception handler looks like this:

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

The block between PROC and EXCEPT is executed as normal, and if no exception occurs, the block between EXCEPT and ENDPROC is skipped, and the procedure is left at ENDPROC. If an exception is raised, either in the PROC part, or in any function that is called in this block, an exception handler is invoked.

13B. using the Raise() function

There are many ways to actually "raise" an exception, the simplest is through the function Raise():

```
Raise(exceptionID=0)
```

the exception ID is simply a constant that defines the type of exception, and is used by handlers to determine what went wrong.
Example:

```
ENUM NOMEM,NOFILE  /* and others */

PROC bla() HANDLE
  DEF mem
  IF (mem:=New(10))=NIL THEN Raise(NOMEM)
  myfunc()
EXCEPT
  SELECT exception
    CASE NOMEM
      WriteF('No memory!\n')
    /* ... and others */
  ENDSELECT
ENDPROC

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

The "exception" variable in the handler always contains the value of the argument to the Raise() call that invoked it.
In both New() cases, the Raise() function invokes the handler of function bla(), and then exits it correctly to the caller of bla(). If myfunc() had its own exception-handler, that one would be invoked for the New() call in myfunc(). The scope of a handler is from the start of the PROC in which it is defined until the EXCEPT keyword, including all calls made from there.

This has three consequences:
A. handlers are organized in a recursive fashion, and which handler is actually invoked is dependant on which function calls which at runtime;
B. if an exception is raised within a handler, the handler of a lower level is invoked. This characteristic of handlers may be used to implement complex recursive resource allocation schemes with great ease, as we'll see shortly.
C. If an exception is raised on a level where no lower-level handler is available (or in a program that hasn't got any handlers at all), the program is terminated. (i.e: Raise(x) has the same effect as CleanUp(0))

other functions:

```
Throw(exceptionID,value)
```

same as Raise(), only now it takes an arbitrary value with it. in a handler one can then scrutinize this value with the variable 'exceptioninfo'

```
ReThrow()
```

has no args. simply does a Throw() on the current exception value, IFF it is <>0.

13C. defining exceptions for built-in functions (RAISE/IF)

With exceptions like before, we have made a major gain over the old way of defining our own "error()" function, but still it is a lot of typing to have to check for NIL with every call to New().

The E exception handling system allows for definition of exceptions for all E functions (like New(), OpenW() etc.), and for all Library functions (OpenLibrary(), AllocMem() etc.), even for those included by modules. Syntax:

```
RAISE <exceptionId> IF <func> <comp> <value> , ...
```

the part after RAISE may be repeated with a ",".
Example:

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

the first line says something like: "whenever a call to New() results in NIL, automatically raise the NOMEM exception". <comp> may be any of = <> > < >= <= After this definition, we may write all through our programs:

```
mem:=New(size)
```

without having to write:

```
IF mem=NIL THEN Raise(NOMEM)
```

Note that the only difference is that 'mem' never gets any value if the runtime system invokes the handler: code is generated for every call to New() to check directly after New() returns and call Raise() when necessary.

We'll now be implementing a small example that would be complex to solve without exception handling: we call a function recursively, and in each we allocate a resource (in this case memory), which we allocate before, and release after the recursive call. What happens when somewhere high in the recursion a severe error occurs, and we have to leave the program? right: we would (in a conventional language) be unable to free all the resources lower in the recursion while leaving the program, because all pointers to those memory areas are stored in unreachable local variables. In E, we can simply raise an exception, and from the end of the handler again raise an exception, thus recursively calling all handlers and releasing all resources. Example:

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

RAISE NOMEM IF AllocMem()=NIL

PROC main()
  alloc()
ENDPROC

PROC alloc() HANDLE
  DEF mem
```

```
  mem:=AllocMem(SIZE,0)              /* see how many blocks we can get */
  alloc()                            /* do recursion */
EXCEPT DO
  IF mem THEN FreeMem(mem,SIZE)
  ReThrow()                          /* recursively call all handlers */
ENDPROC
```

This is of course a simulation of a natural programming problem that is usually far more complex, and thus the need for exception handling becomes far more obvious. For a real-life example program whose error handling would have become very difficult without exception handlers, see the 'D.e' utility source.

The "DO" after an EXCEPT means that instead of jumping to ENDPROC, the main code will simply continue execution in the handler as soon as it gets there. it also sets exception to 0. This is handy if you free resources local to a PROC in the handler.

13D. use of exception-ID's

In real life an exception-ID is of course a normal 32-bit value, and you may pass just about anything to an exception handler: for example, some use it to pass error-description strings

```
Raise('Could not open "gadtools.library"!')
```

However, if you want to use exceptions in expandabele fashion and you want to be able to use future modules that raise exceptions not defined by your program, follow the following guidelines:

- Use and define ID 0 as "no error" (i.e. normal termination)

- For exceptions specific to your program, use the ID's 1-10000. Define these in the usual fashion with ENUM:

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

  (OK will be 0, and others will be 1+)
  ```

- ID's 12336 to 2054847098 (these are all identifiers consisting of upper/lowercase letters and digits of length 2,3 or 4 enclosed in "") are reserved as common exceptions. A common exception is an exception that need not be defined in your program, and that may be used by implementors of modules (with functions in them) to raise exceptions: for example, if you design a set of procedures that perform a certain task, you may want to raise exceptions. As you would want to use those functions in various programs, it would be unpractical to have to coordinate the IDs with the main program, furthermore, if you use more than one set of functions (in a module, in the future) and every module would have a different ID for 'no memory!', things could get out of hand. This is where common exceptions come in: the common out-of-memory ID is "MEM" (including the quotes): any implementor can now simply

  ```
   Raise("MEM")
  ```

  from all different procedures, and the programmer that uses the module only needs to suply an exception handler that understands "MEM"

  future modules that contain sets of functions will specify what exception a certain procedure may raise, and if these overlap with the IDs of other procedures, the task of the programmer that has to deal with the exceptions will be greatly simplified.

  examples:

  (system)

"MEM"	out of memory
"FLOW"	(nearly) stack overflow
"STCK"	garbage collector has stack problems
"^C"	Control-C break
"ARGS"	bad args

  (exec/libraries)

"SIG"	could not allocate signal
"PORT"	could not create messageport
"LIB"	library not available
"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 for others)

(intuition/gadtools/asl/gfx)

"WIN"	failed to open window
"SCR"	failed to open screen
"REQ"	could not open requester
"FREQ"	could not open filerequester
"GAD"	could not create gadget
"MENU"	could not create menu(s)
"FONT"	problem getting font

(dos)

"OPEN"	could not open a file / file does not exist
"OUT"	problems while writing
"IN"	problems while reading
"EOF"	unexpected end of file
"FORM"	input format error
"SEG"	loadseg problems

The general tendency is:
* all uppercase for general system exceptions,
* mixed case for exceptions used by >1 app, but not general enough.
* all lowercase for exceptions raised within your own multi-module application

- all others (including all negative IDs) remain reserved.

14. OO PROGRAMMING

14A. OO features in E

The features descibed here in this chapter are grouped as such since they constitute what is generally seen as the three essential main components that make a language 'Object Oriented' (i.e. inheritance - data hiding - polymorhism). However in E they are by no means a 'separate chapter' since each can be used in any way with other E features.

14B. object inheritance

it's always annoying not being able to express dependencies between OBJECTs, or reuse code that works on a particular OBJECT with a bigger OBJECT that encapsulates the first. Object Inheritance allows you to do just that in E. when you have an object a:

```
OBJECT a
  next, index, term
ENDOBJECT
```

you can make a new object b that has the same properties as a (and is compatible with code for a):

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

is equivalent to:

```
OBJECT b
  next, index, term         /* from a */
  bla, x, burp
ENDOBJECT
```

with DEF p:b, you can directly not only access p.bla as usual, but also p.next.

as an example, if one would have a module with an OBJECT to implement a certain datatype (for example a doubly-linked-list), and PROCs to support it, one could simply inherit from it, adding own data to the object, and use the _existing_ functions to manipulate the list. However, it's only in combination with methods (descibed below), inheritance can show its real power.

14C. data hiding (EXPORT/PRIVATE/PUBLIC)

E has a very handy data-hiding mechanism. Other languages, like C++, use data-hiding on classes, which raises the need for kludges (like 'friends'), and makes datahiding insecure (Eiffel). E's datahiding works on the module-level, which can model class-level datahiding, but enables more intelligent schemes also.

PRIVATE and PUBLIC let you declare a section of an object as visible to the outer world or not; the outer world here is all code outside the module. for the code within a module, everything is always visible. example:

```
OBJECT mydata PRIVATE                   -> whole object is private
  bla:PTR TO mydata, burp, grrr:INT
ENDOBJECT

OBJECT aaargh
  blerk:PTR TO aaargh                   -> public
PRIVATE
  x:INT, y:INT, z:INT                   -> private
PUBLIC
  hmpf[10]:ARRAY OF mydata              -> public again
ENDOBJECT
```

an object is by default public, an occurring PRIVATE or PUBLIC acts as a toggle-switch to the objects current visibility. In the first object, all is private. The second object has only (x,y,z) as private. PRIVATE and PUBLIC keywords may occur:

- in the object header line
- as a line on itself in the object-def
- preceding decls in an object-def (i.e virtually anywhere)

Why datahiding?
If you want to know why datahiding is a good technique, you'd probably want to read a good book on OO. But in short: it is generally assumed that lots of problems in maintaining and enhancing large pieces of software is the fact that it's hard to change things because lots of code start to depend on certain structures in your program. if you datahide an object, only the code within a module will rely on the format of objects, and you can easily change both representation of an object (for example changing a stack implementation from ARRAY to a linked list) and the code that works with it. If a lot of code of a large app depend on the fact that the stack is an ARRAY, you won't be able to simply change it, which will lead to kludges. In general, try to datahide as much as possible without becoming too restrictive on the use of your object. Using methods (below) will often enable you to keep the whole object private.

14D. methods and virtual methods

A method is much like a PROC, only now it's part of an OBJECT. It also allows you to exploit Polymorhism on objects, as we'll see below. definition of a method:

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

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

the 'OF blerk' part tells the compiler it belongs to the object 'blerk'. Note that apart from the 'OF' the syntax is completely like a 'PROC', however, it can't be invoked as one, and also functions quite differently.

'self' is a local variable that is available in every method, and is a pointer to the object that the method belongs to (in this case 'self:PTR TO blerk'). This function just returns the value of the x field of blerk, which actually makes sense, given that this allows you to later change whatever x represents.

we may call methods similar to object selections ".":

```
DEF a:PTR TO blerk
NEW a
...
a.getx()            -> invoke method getx() on object a
```

in this example, upon invocation 'a' becomes the value of 'self' during the execution of getx().

so far the use of methods has been nice, but hasn't show us ist real power, which only comes when used with inheritance.

If I inherit an object that has methods, I automatically get those in the new object:

```
OBJECT burp OF blerk PRIVATE        -> same as blerk, + extra field
  prut:INT
ENDOBJECT

DEF b:PTR TO burp
NEW b
...
b.getx()                            -> same method
```

The interesting thing is now, that instead of inheriting a method, you may also redefine it:

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

(it goes without saying that we may also add new methods) so where appropriate, we can choose to modify slightly the behaviour of methods we get from other objects, while the interface to it (i.e. 'getx()') stays the same. Not only does this allow us to reuse code selectively, we can also make use of polymorhism:

```
PROC dosomething(o:PTR TO blerk)
```

```
  ...
  o.getx()
  ...
ENDPROC

dosomething(a)
dosomething(b)
```

we may call that PROC with both a and b, since both are compatible with a blerk object. But which of the two method implementations of getx() is invoked at o.getx()? Answer: both are. Method calls in E are what virtual method calls are in other languages: they dynamically act on the real type of an object (o) and call the appropriate method.

A more clear example:

-> classical OO polymorphic example

```
OBJECT loc
  PRIVATE xpos:INT, ypos:INT
ENDOBJECT

OBJECT point OF loc
  PRIVATE colour:INT
ENDOBJECT

OBJECT circle OF point
  PRIVATE radius:INT
ENDOBJECT

PROC show() OF loc IS WriteF('I''m a Location!\n')
PROC show() OF point IS WriteF('I''m a Point!\n')
PROC show() OF circle IS WriteF('I''m a Circle!\n')

PROC main()
  DEF x:PTR TO loc,l:PTR TO loc,p:PTR TO point,c:PTR TO circle
  ForAll({x},[NEW l,NEW p,NEW c],`x.show())
ENDPROC
```

In the above, x is a PTR TO loc, so many would expect x.show() to write 'I'm a Location' three times, but instead it writes the right string for each object.

If one would write this example in a non-OO language, one would need a SELECT for every operation like show(), testing some value present in the object to see what it is. If I would add a new shape to this, say:

```
OBJECT ellipse OF circle
```

I would need to change all SELECTs throughout my app to account for it. With object-polymorhism, I can just write a show() method, and ALL code throughout my app that calls x.show() will act correctly when x is a PTR TO ellipse, even without recompilation!

It's difficult to show why this is powerful in a few examples, and best to discover this is using it in real life apps. and: like I said, read a book on it.

How does polymorphism work?
In the above examples, it's clear the compiler can't know what method it's going to call. that's why E uses a 'class object', and every object created gets a ptr to this object. In the class object, all information is stored that is common for all objects of that type, such as pointers to methods. when the E compiler sees a call like x.show(), instead of looking directly at the show() that belongs to the type of x (i.e. loc), it will generate code to retrieve the pointer to the show() method from loc's class object. since the class object for point looks the same as loc (only maybe slightly larger), that code will automatically call point's show(), when x is really a point object. This is sometimes called runtime binding.

Objects that have methods therefore always are 4 bytes larger than you expect them to be, since they contain a class object pointer. This pointer is automatically installed by NEW, which is the reason _currently_ NEW is the only way to create such an object.

If a method is declared with the sole purpose of enabling subclasses to redefine it (this type of class is known as a virtual baseclass in some languages), one may use EMPTY:

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

it may then be redefined in subclasses. Since a programmer might not implement all methods at once, it is not an error when the above method is executed. it will just return 0 or NIL.

One may effectively add methods to system OBJECTs:

```
OBJECT mygadget OF gadget  -> from intuition!
  -> extra fields here
ENDOBJECT

PROC creategadget() OF mygadget IS ...
```

A pointer to an object such as mygadget above is then compatible with a normal gadget pointer, i.e. it may be added directly to a window etc.

14E. Constructors, Destructors and Super-Methods

The constructor name may be anything, but it is usually given the same name as the class. One may even have multiple constructors for one class. A constructor is called directly on the object created with NEW:

```
NEW obj.stack()
```

Destructors however have to be named "end". An object is destroyed like this:

```
END obj
```

If 'obj' has a .end() method, it is automatically called. end() shouldn't have any arguments, and it is of no use returning a value. (see 5M for a desciption of END's precise functioning).

The super-method of a method is the method by the same name of its super class. Sometimes it's handy to call this method because you might want to add its behaviour to the implementation of your class, however, since you've redefined it, calling the super-method by that name will just call yourself (!). The SUPER keyword allows you to call any method of your superclass (or someone else's superclass):

```
SUPER obj.method()
```

This piece of code above can be used as expression and statement. Care has to be taken though, since if your supermethod calls another method of that object, it will call the redefined version, not the one at its own 'level', so to speak (generally this is what one wants). Also, the compiler looks at the static type of 'obj' to find the superclass, not the dynamic type (though it may still have that behaviour).

15. INLINE ASSEMBLY

15A. identifier sharing

As you've probably guessed from the example in chapter 5D, assembly instructions may be freely mixed with E code. The big secret is, that a complete assembler has been built in to the compiler. Apart from normal assembly addressing modes, you may use the following identifiers from E:

```
mylabel:
LEA mylabel(PC),A1          /* labels */

DEF a                       /* variables */
MOVE.L (A0)+,a              /* note that <var> is <offset>(A4) (or A5) */

MOVE.L dosbase,A6           /* library call identifiers */
JSR    Output(A6)

MOVEQ  #TRUE,D0             /* constants */
```

EC's assembler supports following constructs,

where
n = registernum
x = index
lab = label, from: "label:" or "PROC label()"
abs = absolute addressing
s = size. L, W or B where appropriate.

- addressing modes supported by EC:
  Dn, An, (An), (An)+, -(An), x(An), x(An,Dn.s), lab(PC), lab(PC,Dn.s), abs, abs.W
  (note: write abs.W in hexadecimal, to not confuse it with a float value, i.e. write MOVE.L $4.W,A6)

- supported partially:
  #<constexp>

- not supported:
  lab (same as abs), #lab
  use LEA lab(PC),An instead.

- extra modes:
  var.s (transfers contents of var. optionally size is ".W" or ".B", default is ".L")

LibraryFunction(A6)

example:

```
...
MOVE.W myvar.W,D0               -> move lowword of 'myvar'
...
MOVE.L dosbase,A6
JSR    Write(A6)
```

As an extra, E allows to directly return registers from a function:

```
ENDPROC D0
```

You may even interpret this as multiple return values, i.e. D0/D1/D2.

15B. the inline assembler compared to a macro assembler

The inline assembler differs somewhat from your average macro-assembler, and this is caused mainly by the fact that it is an extension to E, and thus it obeys E-syntax. Main differences:

- comments are with /* */ and not with ";", they have a different meaning.
- keywords and registers are in uppercase, everything is case sensitive
- no macros and other luxury assembler stuff (well, there's the complete E language to make up for that ...)
- You should be aware that registers A4/A5 may not be trashed by inline assembly code, as these are used by E code. Also, if your code can be called by code that is register-allocated, you should preserve D3-D7. an instruction like

  ```
  MOVEM.L D3-D7,-(A7); /* inline asm */; MOVEM.L (A7)+,D3-D7
  ```

  should help if problems occur.
- no support for LARGE model/reloc-hunks in assembly _YET_. This means practically that you have to use (PC)-relative addressing for now (which is faster anyway).

15C. ways using binary data (INCBIN/CHAR..)

INCBIN

syntax:

```
INCBIN <filename>
```

includes a binary file at the exact spot of the statement, should therefore be separate from the code. Example:

```
mytab: INCBIN 'df1:data/blabla.bin'
```

LONG, INT, CHAR

syntax:

```
LONG <values>,...
INT <values>,...
CHAR <values>,...
```

Allows you to place binary data directly in your program. Functions much like DC.x in assembly. Note that the CHAR statement also takes strings, and will always be aligned to an even word-boundary. Example:

```
mydata: LONG 1,2; CHAR 3,4,'hi folks!',0,1
```

15D. OPT ASM

OPT ASM is discussed also in chapter 16A. It allows you to operate 'EC' as an assembler. There's no good reason to use EC over some macro-assembler, except that it is significantly faster than for example A68k, equals DevPac and loses of AsmOne (sob 8-{). You will also have a hard time trying to squeeze your disks of old seka-sources through EC, because of the differences as described in chapter 15B. If you want to write assembly programs with EC, and want to keep your sources compatible with other assemblers, simply precede all E-specific elements with a ";", EC will use them, and any other assembler will see them as a comment. Start all regular comments with a smiley (";->").
Example:

```
; OPT ASM

start:   MOVEQ #1,D0            ;-> do something silly
         RTS                    ;-> and exit
```

this will be assembled by any assembler, including EC

15E. Inline asm and register variables

register variables are a great companion to inline assembly, as they function just as registers, but at the same time have clear identifiers instead of Dx, and also are automatically saved and restored by E code. example:

```
PROC bla()
  DEF count:REG
  MOVEQ #10,count
loop: WriteF('count=\d\n',count)
  DBRA count,loop
ENDPROC
```

all instruction that can work with a Dx EA, work with register variables. examples:

```
MOVEQ #1,a
MOVEM.L D0/D1/a/b/A0,-(A7)
LSL.L a,b
```

etc.

as may be known, EC uses D3-D7 for these register variables. If you wish to write code that freely mixes assembly with E, it's advisable to keep longer-term values in register variables, and temporaries in D0-D2/A0-A3/A6

16. TECHNICAL AND IMPLEMENTATION ISSUES

16A. the OPT keyword

OPT, LARGE, STACK, ASM, NOWARN, DIR, OSVERSION, MODULE, EXPORT, RTD, REG

syntax:

```
OPT <options>,...
```

allows you to change some compiler settings:

LARGE	Sets code and data model to large. Default is small; the compiler generates mostly pc-relative code, with a max-size of 32k. With LARGE, there are no such limits, and reloc-hunks are generated (see 0D, LARGE).
STACK=x	Set stacksize to x bytes yourself. Only if you know what you are doing. Normally the compiler makes a very good guess itself at the required stack space (see 16C).
ASM	Set the compiler to assembly mode. From there on, only assembly instructions are allowed, and no initialisation code is generated. (see 15D, inline assembly)
NOWARN	Shut down warnings. The compiler will warn you if it *thinks* your program is incorrect, but still syntactically ok. (see 0D, -n)
DIR=moduledir	Sets the directory where the compiler searches for modules. default='emodules:'
OSVERSION=vers	Default=33 (v1.2). Sets the minimum version of the kickstart (like 37 for v2.04) your program runs on. That way, your program simply fails while the dos.library is being opened in the initialisation code when running on an older machine. However, checking the version yourself and giving an appropriate error-message is more helpful for the user.
MODULE	denotes this source to be a module. (see 10C)
EXPORT	automatically export all declarations in a module
RTD	generates RTD's instead of RTS in the main source. 020+ only. [experimental optimisation]
020,881,040	generate code for these CPUs. not really usable yet.
REG=n	use n register for register-allocation.

example:

OPT STACK=20000,NOWARN,DIR='df1:modules',OSVERSION=39,REG=3

16B. small/large model

Amiga E lets you choose between SMALL and LARGE code/data model. Note that most of the programs you'll write (especially if you just started with E) will fit into 32k when compiled: you won't have to bother setting some code-generation model. You'll recognise the need for LARGE model as soon as EC starts complaining that it can't squeeze your code into 32k anymore. To compile a source with LARGE model:

```
1> ec -l sizy.e
```

or better yet, put the statement

```
OPT LARGE
```

at the top of your code.

16C. stack organisation

To store all local and global variables, the run-time system of an executable generated by Amiga E allocates a chunk of memory, from which it takes some fixed part to store all global variables. The rest will be dynamically used as functions get called. as a function is called in E, space on the stack is reserved to store all local data, which is released upon exit of the function. That is why having large arrays of local data can be dangerous

when used recursively: all data of previous calls to the same function still resides on the stack and eats up large parts of the free stack space. However, if PROCs are called in a linear fashion, there's no way the stack will overflow.
Example:

```
global data:          10k (arrays etc.)
local data PROC #1:    1k
local data PROC #1:    3k
```

the runtime system always reserves an extra 10k over this for normal recursion (for example with small local-arrays) and additional buffers/ system spaces, thus will allocate a total of 24k stack space

16D. hardcoded limits

Note these signs: (+-) just about, depends on situation,
(n.l.) no clear limit, but this seems reasonable.

OBJECT/ITEM	SIZE/AMOUNT/MAX
value datatype CHAR	0 .. 255
value datatype INT	-32 k .. +32 k
value datatype LONG/PTR	-2 gig .. +2 gig
identifierlength	100 bytes (n.l.)
length of one source line	2000 lexical tokens (+-)
source length	2 gig (theoretically)
constant lists	few hundred elements (+-)
constant strings	1000 chars (n.l.)
max. nesting depth of loops (IF, FOR etc.)	500 deep
max. nesting depth of comments	infinite
#of local variables per procedure	8000
#of global variables	7500
#of arguments to own functions	8000 (together with locals)
#of arguments to E-varargs functions (WriteF())	64 (v2.1) / 1024 (v2.5)
one object (allocated local/global or dyn.)	8 k
one array, list or string (local or global)	32 k
one string (dynamically)	32 k
one list (dynamically)	128 k
one array (dynamically)	2 gig
objects with NEW	64k elem.
CHAR/INT/LONG with NEW	2 gig.
local data per procedure	250 meg
global data	250 meg
code size of one procedure	32 k
code size of executable	32 k SMALL, 2 gig LARGE model
current practical limit (may extend in future)	2-5 meg (v2.1) / 10 meg (v2.5)
buffersize of generated code and identifiers	relative to source
buffersize of labels/branches and intermediate	independently (re)allocated

16E. error messages, warnings and the unreferenced check

Sometimes, when compiling your source with EC, you get a message of the sort UNREFERENCED: <ident>, <ident>, ...

This is the case when you have declared variables, functions or labels, but did not use them. This is an extra service rendered to you by the compiler to help you find out about those hard to find errors.

There are several warnings that the compiler issues to notify you that something might be wrong, but is not really an error.

- 'A4/A5 used in inline assembly'
  This is the warning you'll get if you use registers A4 or A5 in your assembly code. The reason for this is that those registers are used internally by E to address the global and local variables respectively. Of course there might be a good reason to use these, like doing a MOVEM.L A4/A5,-(A7) before a large part of inline assembly code

- 'keep an eye on your stacksize'
- 'stack is definitely too small'
  Both these may be issued when you use OPT STACK=<size>. The compiler will simply match your <size> against its own estimate (see 16C), and issue the former warning if it thinks it's ok but a bit on the small side, and the latter if it's probably too small.

- 'suspicious use of "=" in void expressions (s). (line %d)'
  This warning is issued if you write expressions like 'a=1' as a statement. One reason for this is the fact that a comparison doesn't make much sense as a statement, but the main reason is that it could be an often occurring typo for 'a:=1'. Forgetting those ":" may be hard to find, and it may have disastrous consequences.

- 'module changed OPT settings'
  If you use a module that has OPT OSVERSION=37, this changes the OPT for the main program too. this warning serves to make you aware of this. put such an OPT in the main program too to get rid of it.

- 'variable used as function'
  in v3, arbitrary variables may be used as function. this function is there to warn you so you don't accidentally do this.

- 'code outside PROCs'
  You wrote E code in between PROCs, which is only rarely useful.

Errors.
The compiler will print the source-code-line that caused the error below it, with a cursor at the exact spot. The cursor denotes the spot the compiler was when it _discovered_ the error, it is thus likely that the symbol that caused the error is the one just _before_ the cursor.

- 'syntax error'
  Most common error. This error is issued either when no other error is appropriate or your way of ordering code in your sources is too abnormal.

- 'unknown keyword/const'
  You have used an identifier in uppercase (like "IF" or "TRUE"), and the compiler could not find a definition for it. Causes:
  * mispelled keyword
  * you used a constant, but forgot to define it in a CONST statement
  * you forgot to specify the module where your constant is defined

- '":=" expected'
  You have written a FOR statement or an assignment, and put something other than ":=" in its place.

- 'unexpected characters in line'
  You used characters that have no syntactic meaning in E outside of a string. examples: "@!&\~"

- 'label expected'
  At some places, for example after the PROC or JUMP keyword, a label identifier is required. You wrote something else.

- '"," expected'
  In specifying a list of items (for example a parameter list) you wrote something else instead of a comma.

- 'variable expected'
  This construction requires a variable, example:

```
FOR <var>:= ... etc.
```

- 'value does not fit in 32 bit'
  In specifying a constant value (see 2A-2E) you wrote too large a number, examples: $FFFFFFFFF, "abcdef". Also occurs when you define a SET of more than 32 elements.

- 'missing apostrophe/quote'
  You forgot the ' at the other end of a string.

- 'incoherent program structure'
  * you started a new PROC before ending the last one
  * you don't nest your loops properly, for example:

```
FOR
  IF
  ENDFOR
ENDIF
```

- 'illegal command-line option'
  In specifying 'EC -opt source' you wrote something for '-opt' that is not a legal option to EC.

- 'division and multiplication 16bit only'
  The compiler detected that you were about to use 32bits for * or /. This would not have the desired result at runtime.
  (see 9G, Mul() and Div()).

- 'superfluous items in expression/statement'
  After the compiler already compiled your statement, it still found lexical tokens instead of an end of line. You probably forgot the <lf> or ";" to separate two statements.

- 'procedure "main" not available'
  Your program does not include a main procedure !

- 'double declaration of label'
  You declared a label twice, for example:

```
label:
PROC label()
```

- 'unsafe use of "*" or "/"'
  This again has to do with 16bit instead of 32bit * and /. See 'division and multiplication 16bit only'.

- "reading sourcefile didn't succeed"
  Check your source spec. that you gave with 'ec mysource' make sure the file ends in '.e'

- "writing executable didn't succeed"
  Trying to write the generated code as an executable caused a dos error. For example, the executable that did already exist could not be overwritten.

- 'no args'
  "USAGE: ec [-opts] <sourcecodefilename> ('.e' is added)" You get this by just typing 'ec' without any arguments.

- 'unknown/illegal addressing mode'
  This error is reported only by the inline assembler. Possible causes are:
  * you used some addressing mode that does not exist on the 68000
  * the addressing mode exists, but not for this instruction. not all assembly instructions support all combinations of effective addresses for source and destination.

- 'unmatched parentheses'
  Your statement has more "(" than ")" or the other way around

- 'double declaration'
  One identifier is used in two or more declarations.

- 'unknown identifier'
  An identifier is not used in any declaration; it is unknown. You probably forgot to put it in a DEF statement.

- 'incorrect #of args or use of ()'
  * You forgot to put "(" or ")" at the right spot
  * you supplied the incorrect #of arguments to some function

- 'unknown e/library function'
  You wrote an identifier with the first character in uppercase, and the second in lowercase, but the compiler could not find a definition.
  Possible causes:
  * Misspelled name of function
  * You forgot to include the module that defines this library call.

- 'illegal function call'
  Rarely occurs. You get this one if you try to construct weird function calls like nested WriteF()'s. Example:

```
WriteF(WriteF('hi!'))
```

- 'unknown format code following "\"'
  You specified a format code in a string which is illegal.
  (see 2F for a listing of format codes)

- '/* not properly nested comment structure */'
  The #of '/*' is unequal to the #of '*/', or is placed in a funny order.

- 'could not load binary'
  <filespec> in INCBIN <filespec> could not be read.

- '"}" expected'
  You started an expression with "{<var>" , but forgot the "}"

- 'immediate value expected'
  Some constructions require an immediate value instead of an expression.
  Example:

```
DEF s[x*y]:STRING            /* wrong: only something like s[100]:STRING is legal */
```

- 'incorrect size of value'
  You specified an unacceptably large (or small) value for some construction.
  Examples:

```
DEF s[-1]:STRING, t[1000000]:STRING      /* needs to be 0..32000  */
MOVEQ #1000,D2                           /* needs to be -128..127 */
```

- 'no e code allowed in assembly modus'
  You wish to operate the compiler as an assembler by writing 'OPT ASM', but, by accident, wrote some E code.

- 'illegal/inappropriate type'
  At someplace where a <type> spec. was needed, you wrote something inappropriate. Examples:

```
DEF a:PTR TO ARRAY                       /* no such type */
[1,2,3]:STRING
```

- '"]" expected'

You started with "[", but never ended with "]"

- 'statement out of local/global scope'
  A breakpoint of scope is the first PROC statement. before that, only global definitions (DEF, CONST, MODULE etc.) are allowed, and no code. In the second part, only code and function definitions are legal, no global definitions.

- 'could not read module correctly'
  A dos error occurred while trying to read a module from a MODULE statement. Causes:
  * emodules: was not assigned properly
  * module name was misspelled, or did not exist in the first place
  * you wrote MODULE 'bla.m' instead of MODULE 'bla'

- 'workspace full!'
  Rarely occurs. If it does, you'll need the '-m' (ADDBUF) option to manually force EC to make a bigger estimate on the needed amount of memory. Try compiling with -m2, then -m3 until the error disappears. You'll probably be writing huge applications with giant amounts of data just to even possibly get this error.

- 'not enough memory while (re-)allocating'
  Just like that. Possible solutions:
  1. You were running other programs in multitasking. Leave them and try again.
  2. You were low on memory anyway and your memory was fragmented. Try rebooting.
  3. None of 1-2. Buy a memory expansion (um).

- 'incorrect object definition'
  You were being silly while writing the definitions between OBJECT and ENDOBJECT. (see 8F to find out how to do it right).

- 'illegal use of/reference to object'
  If you use expressions like ptr.member, member needs to be a legal member of the object ptr is pointing to.

- 'incomplete if-then-else expression'
  If you use IF as an operator (see 4E), then an ELSE part needs to be present: an expression with an IF in it always needs to return a value, while a statement with an IF in it can just 'do nothing' if no ELSE part is present.

- 'unknown object identifier'
  You used an identifier that was recognised by the compiler as being part of some object, but you forgot to declare it. Causes:
  * misspelled name
  * missing module
  * the identifier in the module is spelled not like you expected from the RKRM's. Check with ShowModule. Note that amiga-system-objects inherit from assembly identifiers, not from C. Second: identifiers obey E-syntax.

- 'double declaration of object identifier'
  One identifier used in two object definitions

- 'reference(s) out of 32k range: switch to LARGE model'
  Your program is growing larger than 32k. Simply put 'OPT LARGE' in your source and code on. (see 16B).

- 'reference(s) out of 256 byte range'
  You probably wrote BRA.S or Bcc.S over too great a distance.

- 'too sizy expression'
  You used a string '' or list [], possibly recursive [[]], that is too sizy.

- 'incomplete exception handler definition'
  You probably used EXCEPT without HANDLE, or the other way round
  (see 13A on exception handling).

- 'not allowed in a module'
  You're doing one of the few things you can't do in a module, such as global variables with initialisations.

- 'allowed in modules only'
  you probably use EXPORT in your main source

- 'this doesn't make sense'
  general error.

- 'you need a newer version of EC for this :-)'
  You probably use a module that was compiled with a newer version of EC than you currently have.

- 'no matching "["'
  within a statement, a "]" was found, without a matching "]".

- 'this instruction needs a better CPU/FPU (see OPT)'
  You use a construction (probably an asm instruction) that requires an OPT 020 or the like.

- 'object doesn't understand this method'
  you invoke a method on a object that wasn't defined for its type.

- 'method doesn't have same #of args as method of baseclass'
  If you redefine a method, you have to make sure the new one has the same #of args as the original.

- 'too many register variables in this function'
  If you use :REG to assign register variables yourself, you can't use more than 5, currently.

- 'Linker can't find all symbols'
  If you use a module A that uses again a module B, B also needs to linked. A relies on certain PROCs to be present in B, and if B was recompiled with those PROCs removed, the linker has trouble putting your exe together.

- 'could not open "mathieeesingbas.library"'
  If you use float code, the compiler itself may need float functions to be able to generate code.

- 'illegal destructor definition'
  You defined an end() method with arguments (or with a returnvalue)

- 'implicit initialisation of private members'
  You write a [...] or NEW [...] expression that has private parts.

- 'double method declaration'
  You defined a method on this object twice.

- 'object referenced by other object not found'
  You probably inherited from some other object in another module, and then changed it (its name for example) without changing/recompiling other dependant modules too.

- 'unknown preprocessor keyword'
  only #define, #ifdef, #ifndef and #endif are known by EC's PP.

- 'illegal macro definition'
  You made a syntax error in typing your #define (see 17L)

- 'incoherent #ifdef/#ifndef nesting'
  You forgot to close with #endif or similar

- 'macro redefinition'
  You can't use the same macro-name-identifier twice.

- 'syntax error in #ifdef/#ifndef/#else/#endif'

(see 17L for the correct way to write conditionally compiled code)

- 'macro(s) nested too deep'
  you will get this if macros expand to other macros which expand to yet others... such that the amount of memory needed for this is getting out of hand. More likely you just defined a recursive macro (which will want to expand forever).

- 'method definition out of object/module scope'
  You can only define methods for an object in the same module/source where that object is defined.

16F. compiler buffer organisation and allocation

When you get the error 'workspace full' (very unlikely), or want to know what really happens when your program is compiled, it's useful to know how EC organizes its buffers.

A compiler, and in this case EC needs buffers to keep track of all sorts of things, like identifiers etc., and it needs a buffer to keep the generated code in. EC doesn't know how big these buffers need to be. for most buffers, like the one for various strcutures, this is no problem: if the buffer is full while compiling, EC just allocates a new piece of memory and continues. Other buffers, like the one for the generated code, need to be a continuous block of memory that doesn't move while compiling: EC needs to make a pretty good estimate of this buffersize to be able to compile small and large sources alike. To do this, EC computes the needed memory relative to the size of your source code, and adds a nice amount to it. This way, in 99% of the cases, EC will have allocated enough memory to compile just about any source, in other cases, you'll get the error and have to specify more memory with the '-m' (ADDBUF) option.
Experiment with different types and sizes of example-sources in combination with the '-b' (SHOWBUF) option (see 0D) to see how this works in practice.

16G. register allocation

E v3 supports a register allocation, which is a technique to keep variables in registers instead of on the stack. For normal code that uses OS-routines you won't notice the difference very much, but for tight computation-loops, this optimisation can make a big difference. There are two ways to use register allocation:

- with the option REG.
  If you write for example EC REG=3 bla.e, (max=5, currently), EC will compute for each PROC the 3 most-used variables in registers. Register allocation is a technique that tries to be intelligent: it will compute for each var a weight, and will use heuristics to increase that weight, for example a var used in a FOR loop gets relatively a higher weight than one outside it, and one in an IF gets an even lower weight. These weights are combined, so a WHILE in a FOR gets quite a high weight.

- DIY: you can put the keyword REG in front of any type in a declaration, for example:

  ```
  DEF x:REG, s[4]:REG LIST
  ```

  you can do this if you don't trust the register-allocator, or if you want to fine-tune just one PROC. You can even use both together: if in a PROC you have one var with :REG, compiling with REG=5 will allow EC to pick the remaining 4 by itself.

The default is REG=0, so EC works much like the older versions.

The variables that CAN be allocated are only local variables that are not parameters. also, if you take the address of a variable with {} it can't be put in a register either (guess why...). registers can't be allocated in PROCs that have an exception handler, for now.

There are a few things to note when using registers:
- this part of EC is currently (E v3.0a) was tested to be pretty reliable, but you still check that behaviour is the same as in non-allocated code. it _should_ work ok, but it's too early to guarantee it :-) In short: be careful for now when applying these techniques.

- EC uses registers D7..D3 for variables, so if you use inline assembly, you need to check that PROCs that use register-allocation or :REG don't trash these (see 15E). The code generated for a PROC automatically saves the registers it uses (callee save) to protect the code that called it.
- hint: compiling with REG=5 is not inherently fastest, since variable saving on function/library calls also incurs an overhead. REG=3 may be better for some cases. Also if _all_ code in question deals with library calls instead of pure computation, expect no gain from registers.

-> register allocation will easily make this program twice as fast

```
PROC main()
  DEF a,b=10,c=20,d
  FOR a:=1 TO 1000000 DO d:=b+c
ENDPROC
```

-> at most 5% faster when using register allocation

```
PROC main()
  DEF a,s[100]:STRING,t
  t:='putting "a" in a reg won''t give that much of a speedup, I think.'
  FOR a:=1 TO 100000 DO StrCopy(s,t)
ENDPROC
```

17. Essential E Utilities

17A. bin/showmodule

As you might have noticed, E's equivalent for "includes", modules, are binary files, much like those usually suplied with Modula2 compilers, for example. To display the contents of such a file in readable ascii form, you may use showmodule:

```
showmodule <modulespec>
```

examples:

```
1> showmodule emodules:intuition/intuition
1> showmodule >gadtools.txt emodules:gadtools
```

note that showmodule by default outputs to stdout, and may be interrupted at any point by <ctrlc>.

17B. sources/utilities/showhunk.e, bin/showhunk

Displays all types of executable files, and also object ".o" files as generated by (other) compilers/assemblers. will show you the (very simple) structure of executables generated by EC, but also support complex overlay-files. also dumps labels (like XREF's and XDEF's).

most important of all, ShowHunk features a disassembler for code-hunks. use the option 'DISASM/S'

```
showhunk <exefile>
```

like:

```
1> showhunk helloworld
1> showhunk disasm dpaint
```

17C. bin/iconvert, bin/pragma2module

These two utilities are for advanced E-programmers only. if you don't feel like one (yet), skip this part.

[NOTE: like the showmodule utility, the sources to these utilities have been removed from the distribution, because people misused their knowledge of the .m module format. It is PRIVATE. Contact me first if you want to do something with it.]

Iconvert will convert structure and constant definitions in assembly ".i" files to E modules, and pragma2module will do the same for SAS/C pragma library definition files. Of course, all commodores includes have already been converted this way, but say you find a nice PD library that you may want to use with E, you will need these utilities.

most libraries come with various includes defining, most obvious, the library calls of the library, as well as the constants and structures (OBJECTs in E) that it uses. say that it is called "tools.library", then it will probably feature:

pragmas/tools_pragmas.h
includes/tools.i

then do:

```
1> pragma2module tools_pragmas.h
```

rename the resulting "tools_pragmas.m" to "tools.m" and put it in emodules:, check with ShowModule if all went well.

Now, in your program you may use tools.library:

```
MODULE 'tools'

PROC main()
  IF (toolsbase:=Openlibrary('tools.library',37))=NIL THEN error()

  ToolsFunc()
```

...etc.

convert tools.i with Iconvert to another tools.m, which you place in emodules:libraries, for example. Iconvert needs an assembler like the PD A68k to do the hard work of understanding the actual assembly.

```
1> iconvert tools.i
```

see with showmodule what became of the ".i" file. use in your program with:

```
MODULE 'libraries/tools'

DEF x:toolsobj, y=TOOLS_CONST
```

converting with Iconvert may require some assembly expertise, as Iconvert relies on the correct format of the ".i" file, just like commodores assembly includes. About 10% of the ".i" files need to be patched by hand to be "convertable". definitions that Iconvert judges correctly are amongst others

```
<label> EQU <any_expression>

STRUCTURE <sname>,0      ; if <>0, then   <struct>_SIZEOF
ULONG <sname>_<label>
BPTR <sname>_<label>
   ; etc.
LABEL <sname>_SIZEOF     ; or "_SIZE"
```

to get an idea what kind of assembly-expression Iconvert can handle, take a look at commodores assembly includes and compare them to the equivalent modules (for example intuition.i).

17D. bin/ShowCache, bin/FlushCache

The E Module Cache is a piece of memory that is able to hold modules (.m) between compiles. The first time you use a certain module, EC will load it from disk and put it in the cache. the second time and on, EC will find it in cache and doesn't have to load anything from disk. If EC compiles a module of which an old version is present in cache, it will flush it. One can imagine that this a tremendous speedup, even for people with HD's when they use a lot of modules and recompile often.
To see what's currently stored in the cache (and how much memory it's wasting :-), type:

```
1> ShowCache
```

A second utility, FlushCache, allows you to selectively remove modules from the cache. Reasons for this can be:

- you can't afford the memory it uses
- you created a new .m, with a tool other than EC, so you need to flush by hand. The argument to Flushcache is the substring that needs to occur in a module name for it to be flushed. no args means flush all.

```
1> FlushCache                       ; empty whole cache
1> FlushCache intuition/            ; flush all intuition-related modules
```

You can use the EC option 'IGNORECACHE/S' to compile a source without using the cache. Whether the cache is full or empty, EC will load all from disk, and store nothing new in cache.

If two EC's try to access the cache simultaneously in multitasking, the second EC acts as if its IGNORECACHE flag were set.

17E. rexx/ecompile.rexx

```
[what's keeping the other rexx-scripts?]
```

This is a rexx-script for CygnusEd (tm), and enables you to compile E programs from the editor. Just assign this script a function key in the editor with "Install Dos/Arexx command ..." (check your CED-manual if you're not sure how to do this). Now write your programs, and press Fx if you wish to compile. Your source will be

saved if necessary, the compiler will be invoked on a separate console window, and the program is run on the same console. When your program is done, you may press <return> to return to the editor (CED-screen to back and front is automatically done by the script). If an error occurred during compilation, the script will let CED jump to the line of error after you pressed <return>

Note: in the script there is a path name as to where the compiler can be found. You probably need to change this. Also, the script copies EC to ram: for systems with a slower SYS: device, you may want to disable this if you have a fast HD.

17F. bin/o2m

If you have large pieces of assembly source that you'd like to use it would be tedious at best to convert them all by hand to E's inline assembly. o2m allows you to simply have your favourite macro-assembler assemble it all to a .o file, and o2, then will turn this .o file into a .m file for use with E. If you have a file bla.o:

```
1> o2m bla
```

will produce bla.m. However, the .o file will have to obey certain rules. It should consist of just one code-hunk with external definitions (XDEFs) for each symbol you wish to reference from E, and no XREFs. typically, your source would look like:

```
XDEF add__ii

add__ii:
   move.l  4(a7),d0
   add.l   8(a7),d0
   rts
```

this example shows a bit of assembly code that gets two arguments (hence the two "i" for integer). arguments can be found on the stack, where 4(a7) is the last arg, 8(a7) the one before that etc.

Showhunk shows you this:

```
            hunk_unit:
HUNK -1     hunk_name:
            hunk_code: 12 bytes
            hunk_ext
              add__ii = $0
```

this type of .o file is easily transformed to .m by o2m:

```
/* this module contains 12 bytes of code! */

PROC add(a,b)
```

- if your asm code uses D3-D7/A4/A5 you should probably save it.
- if a label doesn't have the "__" with an "i" for each function, it becomes a parameterless function. Don't worry if the label actually references data, you can simply get the address of this 'proc' with {}, and use it as a ptr to your data.

theoretically, o2m could be used to link C code to E programs, however in practise this is often not feasable. If your C compiler allows you to 'tune' the resulting .o files a bit, this might work.

some problems are:
- reference of C functions, for example _printf()
- reference of globals vars created by C startup code. C code may reference "DOSBase" as an XREF, whereas E's startup code makes this value available somewhere on the stack
- call/register conventions.

I did manage to link a small C function to E that only does some computation, and call it succesfully (this was done using MaxonC++, whose linker uses the __ii convention for parameters also).

17G. bin/EYacc

This is a port of the famous Yacc utility for unix, which now produces E code instead of C code. It is only a first version, so don't expect too much from it. If you have no clue what Yacc does, read a text on it, since I'm not going to explain that in full here.

Basically you can write .y sources as normal, only where actions used to be written in C, now you can write E. See the Src/Yacc/bcalc.y
example.

```
1> eyacc bcalc.y
```

produces a file 'yyparse.e'

```
1> ec yyparse
```

will get you a module, which contains only the function yyparse(). The rest of 'how to interface with Yacc' should be analoguous to C.

[note: I'm halfway through a translation of Lex to E-Lex, but not done yet.]

further info.
E-Yacc is a modification of Berkeley Yacc 1.8, originally by corbett@berkeley.edu. The inclusion of this modified version in the E distribution is totally legal, as the author states in the original BYacc1.8 README:

> " 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. "

17H. bin/SrcGen

[note: this utility hasn't been updated to work better with the E module system, it still outputs just the raw source (which is then easily incorporated in any module). If there is much demand I will upgrade this utility. If you want to add a nice GUI to your program, also take a look at modules/tools/EasyGUI.m, or newer toolkits suchs as the upcoming BGUI from Jan van den Baard, the author of GadToolsBox.]

SrcGen GadToolBox source generator for E: beta version

You'll be needing GadToolBox v2.0 or higher, and have the gadtoolsbox.library that comes with it in your LIBS:. Now, with GTB, make some simple example (just a window with a few gadgets/menus etc.), save it as "bla" (filename will be "bla.gui"), and type:

```
1> SrcGen bla
1> EC bla
1> bla
```

"bla.e" contains the routines for opening your interface, as well as some routines to handle idcmpmessages, errors etc., and a dummy "main" that just waits for one selection. here you can put in your own code. see the commandline template how to stop SrcGen from generating these routines.

That's all there's to it. If you have problems, just check the source that has been generated.

17I. bin/EBuild

EBuild is a "Make" clone, and it functions likewise. Build is a tool that helps you in recompiling necessary parts of a large application after modification. You write a file ".build" in the directory that contains the sources of your project. The file contains info about which sources depend on which, and what actions need to be performed if a module or exe needs to be rebuilt. Build checks the dates of the files to see if a source has been modified after the last compilation, and if the source uses modules that also have been modified, it will compile these first.

the syntax equals that of unix-make. in general, "#" precedes lines with comments, and:

```
target: dep1 dep2 ...
  action1
  action2
  ...
```

target is the resulting file we're talking about, in most cases an exe or module, but may be anything. Following the ":" you write all files that it depends upon, most notably its source, and other modules. The actions on the following lines are normal AmigaDos commands, and need to be preceded by at least one space or tab to distinquish them from targets.

```
bla: bla.e defs.m
     ec bla quiet
```

this simple example will only recompile 'bla.e' if it was modified, or if the defs.m which it uses was modified.

If you type 'build' with no args, build will ensure the first target to be up to date.

```
TARGET,FROM/K,FORCE/S:
```

if you supply a TARGET, this way build will start with another target. FROM allows you to use another file than ".build", and FORCE will rebuild everything, regardless of whether it was really necessary.

Example:

```
# test build file

all:           bla burp

defs.m:        defs.e
               ec defs quiet

bla:           bla.e defs.m
               ec bla quiet

burp:          burp.e
               ec burp quiet

clean:
               delete  defs.m bla burp
```

this build file is about two programs, bla and burp, of which bla also depends on a module defs.m. An extra fake target 'clean' has been added so you can type 'build clean' to delete all results.

Other dependencies and actions are easily added. for example, if your project uses a parser generated by E-Yacc:

```
yyparse.m:     parser.y
               eyacc parser.y
               ec yyparse quiet
```

Or incorporates macro-assembly code as often used tool module:

```
blerk.m:       blerk.s
               a68k blerk.s
               o2m blerk
               copy blerk.m emodules:tools
               flushcache tools/blerk
```

Once you get to know build, you'll discover you can use it for more purposes than just this. see it as an intelligent script tool.

If you want to find out the details of what build can do, read the documentation of some unix-make, as build should be somewhat compatible with this. what it doesn't do for now, is:

- rule out cyclic dependancies
- allow "\" at the end of a line for longer rules
- constant definitions

For v3.1 it was updated by Jason Hulance, to fix the bug that executed actions in reverse order. Also he changed execution of action into a script (transparent). in this script the variable 'target' is set to the actual target. example:

```
test:    test.e
         ec "$target"
         if warn
               echo "Error: compile failed"
         else
               echo "Compiled OK... running"
               "$target"
         endif
```

largely equivalent to the old code below, but allows more.

```
all:     test
         echo "ok, running:"
         test

test:    test.e
         ec -q test
```

17J. EE / Aprof

These are described in their own documentation in the tools directory.

17K. EDBG

EDBG is the E sourcelevel debugger. To use it, compile your source with the DEBUG flag (this works on both main program an modules). This will add debug infos to your executable/module

NOTE: Do NOT distribute a program of which any part is compiled with DEBUG/S. (You can check this with ShowHunk, it should not contain any hunk_debug). Programs with debug-infos are compiled with extra NOPs to facilitate debugging, which isn't wanted in the final code.

Always make sure sources and compiled are in the current directory, then fire up EDBG with:

```
1> EDBG exename
```

template:

EXECUTABLE/A,PUBSCREEN/K:

with for example PUBSCREEN=Workbench you can run EDBG anywhere. By default it opens it own screen, which is a clone of the workbench in size and display mode. EDBG works fine on the Picasso etc.

EDBG opens a window for every source in your project, and it will start with the one that contains 'main'. From you can step through your code, and windows will automatically open when necessary. When code doesn't have a source attached, it can be executed only (which is sometimes handy, if it doesn't need to be debugged).

From here all is pretty intuitive. Major buttons are the first two pictures which are step over/in. Step in follows the code every step as it is executed, step over does the same but does not enter subroutines. [Just try it, the debugger is quite intuitive]

Other functions show memory or register windows, and various other functions (some not implemented). An important one is double-clicking on variable-names: this will show their contents in the temporary window (I know, this should be something more comfortable in the future).

CAVEATS:

- The debugged program runs on the same task as EDBG. this means that all code that does something special to the task will have to be careful. An example is Forbid() etc.
- A special case is ReadArgs(). because EDBG already read the args, a call from the debugged program will cause a read from the console. So you can conveniently type the args to your program on the commandline and press return.

NOTE: EDBG is still a bit beta, but already VERY useful. It misses a lot of features, and you'll just have to wait a bit before these get implemented. [so don't come and warn me "X doesn't work" or "EDBG needs X", since I _know_.]

The LINEDEBUG and SYM options.
The LINEDEBUG option adds linedebug info to your executable (for each line of code inside a PROC). This option is necessary for EDBG, but the DEBUG switch automatically turns it on. LINEDEBUG is partially compatible with the "LINE" HUNK_DEBUG produced by other compilers/assemblers, so it can be useful with other debug-tools as well. The SYM option is not necessary for EDBG, but can be useful for others, such as AProf or disassemblers.

worst known bugs:
- You can't scroll the current-line out of sight in a sourceview because EDBG tries to keep it in sight.
- lots others... probably

17L. EC PreProcessor

EC has an internal preprocessor which features macro substitution and conditional compilation. These are not features of the E language, rather it has been integrated with EC for speed and flexibility.

Activating the preprocessor.
Until you type:

```
OPT PREPROCESS
```

EC will behave as normal. This OPT is necessary for any preprocessor related feature.

Macros.
The macropreprocessor is compatible with Mac2E and the C language PP. You can define macros as follows:

```
#define MACRONAME
#define MACRONAME BODY
#define MACRONAME(ARG,...) BODY
#define MACRONAME(ARG,...) BODY \
  REST OF BODY
```

MACRONAME and ARG may be ANY case, and may contain "_" and 0-9 as usual. Whitespace may be added everywhere, except between MACRONAME and "(", because otherwise EC can't see the difference between arguments and a body. The BODY may contain occurances of the ARGs. A macro may continue on the next line by preceding the end_of_line with a "\". [a macroname with no body is useful in combination with conditional compilation].

Macro identifiers have precedence over other identifiers.

Macro definitions defined in a module will be saved in the module only if OPT EXPORT is on (#define can't be preceded with EXPORT). If that's a problem, keep macros together in their own modules. Macros in modules can be used in other code simply by importing them with MODULE and OPT PREPROCESS.

Using a macro.
Using MACRONAME anywhere in the program will insert the body of the macro at that spot. Note that this substitution is text substitution, and has little to do with actual E syntax. If the macros have arguments they will be inserted at their respective places in the macrobody. If the body (or the arguments) contain futher macros, these will be expanded after that.

example:

```
#define MAX(x,y) (IF x>y THEN x ELSE y)

WriteF('biggest = \d\n',MAX(10,a))
```

will equal writing:

```
WriteF('biggest = \d\n',(IF 10>a THEN 10 ELSE a))
```

This immediately shows a danger of macros: since it simply copies textually, writing MAX(large_computation(),1) will generate code that executes large_computation() twice. Be aware of this.

Conditional compilation.
This can be useful if you wish to decide at compile time which part of your code to use. syntax:

```
#ifdef MACRONAME
```

or

```
#ifndef MACRONAME
```

the piece of source following this will or won't be compiled depending on whether MACRONAME was defined. you can simply do this with:

```
#define MYFLAG
```

or somesuch. End a conditionally compiled block with:

```
#endif
```

Blocks like these may be nested. example:

```
#define DEBUG
->#define HEAVYDEBUG

#ifdef DEBUG
  WriteF('now entering bla() with x = \d\n',x)
#ifdef HEAVYDEBUG
  WriteF('now dumping memory...\n')
  /* ... */
#endif
#endif
```

18. APPENDICES

18A. The E grammar

This is a grammar of E for those who are interested. Don't expect it to be up to date or otherwise complete/correct though (it should be quite ok for E upto v2.1b atleast).

lex syntax: regular expressions parse syntax: own ASF/SDF adaption;

```
name     = grammar ident
"name"   = constant
()       = grouping
|        = or
e*       = 0 or more of e
e+       = 1 or more of e
{e s}*   = 0 or more of e separated by s
{e s}+   = 1 or more of e separated by s
[e]      = e is optional
; e      = e is comment :-)
```

LEX

```
whitespace        = [ \t]     ; also \n if last token is [,+-*/] or similar anything between
                                "/*" and "*/" from "->" to \n
eol               = [;\n]

constant          = [A-Z] ( [A-Z] [A-Za-z0-9_]* )?
builtin           = [A-Z] [a-z] [A-Za-z0-9_]*
$ident,objident   = [a-z] [a-zA-Z0-9_]*

num               = [0-9]+    ; "-" is separate token  [0-9A-Fa-f]+  %[01]+
fnum              = [0-9]*.[0-9]*

stringconst       = anything in ''
charconst         = anything in ""
```

PARSE

```
program           = opts globalpart localpart

globalpart        = ( modulestat | defstat | objdecl | constdecl | raisedecl )*
localpart         = ( procdecl | constdecl )+

modulestat        = "MODULE" { conststring "," }+ eol
defstat           = "DEF" vardecllist eol
objdecl           = "OBJECT" ident [ "OF" ident ] eol
                    ( vardecllist eol )+
                    "ENDOBJECT" eol
constdecl         = "CONST" { ( constant "=" constexp ) "," }+ |
                    "ENUM" { ( constant | constant "=" constexp ) "," }+ |
                    "SET" { constant "," }+
procdecl          = [ "EXPORT" ] "PROC" ident "(" argdecllist ")"
                    "OF" ident ] [ "HANDLE" ]
                    ( ( "RETURN" | "IS" ) { exp "," }* |
                    eol defstat* stats
                    [ "EXCEPT" eol stats ]
                     "ENDPROC" { exp "," }* eol )
raisedecl         = "RAISE" { ( constant "IF" builtin "()" compop num ) "," }+
opts              = ( "OPT" { setting "," }+ )*            ; machine dependant

vardecllist       = { vardecl "," }+
vardecl           = ident [ "=" num ]
                    [ ":" ( "LONG" | "REAL" | "PTR" "TO" ptrtype ) ] |
                    ident ":" objtype |
                    ident "[" num "]" ":"
                    ( "ARRAY" |
                    "ARRAY" "OF" ptrtype |
```

```
                    "STRING" |
                    "LIST" )
argdecllist       = { argdecl "," }+
argdecl           = ident [ "=" defaultarg ]
                    [ ":" ( "LONG" | "REAL" | "PTR" "TO" ptrtype ) ]
ptrtype           = objtype | simpletype
simpletype        = CHAR | INT | LONG
objtype           = ident

stats             = ( ( onelinestat | multlinestat ) eol )*
onelinestat       = exp |
                    lval ":=" exp |
                    { var "," }+ ":=" exp |
                    "IF" exp "THEN" onelinestat "ELSE" onelinestat |
                    "FOR" var ":=" exp "TO" exp [ "STEP" num ]
                    "DO" onelinestat |
                    "WHILE" exp "DO" onelinestat |
                    "RETURN" { exp "," }* |
                    "JUMP" ident |
                    ( "INC" | "DEC" ) var |                ; nearly obsolete
                    asm_mnemonic { operand "," }*     |    ; machine dependant
                    "INCBIN" stringconst |                 ; inline asm support
                    simpletype { num "," }+ |
                    "VOID" exp                             ; obsolete
multlinestat      = "IF" exp eol stats
                    [ ( "ELSEIF" exp eol stats )* ]
                    [ "ELSE" eol stats ]
                    "ENDIF" |
                    "FOR" var ":=" exp "TO" exp [ "STEP" num ] eol
                    stats "ENDPROC" |
                    "WHILE" exp eol stats "ENDWHILE" |
                    "REPEAT" eol stats "UNTIL" exp |
                    "SELECT" var eol
                    ( "CASE" exp eol stats )+
                    [ "DEFAULT" eol stats ]
                    "ENDSELECT" |
                    "LOOP" eol stats "ENDLOOP"

explist           = { exp "," }+
exp               = [ "-" ] { item binop }+ |
                    exp "BUT" exp
item              = num | fnum | lval | stringconst | charconst |
                    "SIZEOF" objident |
                    "IF" exp "THEN" exp "ELSE" exp |
                    "[" explist "]" [ ":" ptrtype ] |
                    ( builtin | ident ) "(" explist ")" |
                    var ":=" exp |
                    "{" ident "}" |
                    "`" exp |
binop             = mathop | compop | logop
mathop            = "+" | "-" | "*" | "/"
compop            = "=" | "<>" | ">" | "<" | ">=" | "<="
logop             = "AND" | "OR"
constexp          = [ "-" ] { num ( "+" | "-" | "*" | "/" ) }+
lval              = var ( "[" [ exp ] "]" | "." ident )* [ "++" | "--" ] |
                    "^" var [ "++" | "--" ] |
var               = ident
defaultarg        = num
```

18B. Tutorial

NOTE: the original E v2.1b tutorial has been removed, since it was not a very useful tutorial, IMHO. Instead, Jason Hulance made an extensive E tutorial that you'd definitly want to take a look at.

18C. Mapping E to C/C++/Pascal/Ada/Lisp etc.

[some new stuff has been added here]

In the first/second column I will match E against AnsiC/C++, the third column is reserved for a third language. I will mainly use Pascal here, but where a feature asks for it, I will use others (for example, LISP with quoted expression, Ada with exceptions etc.

note well: take these tables with a grain of salt. I'll try to denote syntactic equivalences, and semantic properties as well as possible, but different languages still need their own evaluation.

usage of signs:

- = feature not available in language in question.
? = author has no clue what this feature translates to. (or atleast he's not sure).
... = feature may be available, but no appropriate 1:1 translation possible to make it interesting.
x,y,z = arbitrary identifiers
e,f,g = arbitrary expressions
s,t,u = arbitrary statements
i,j,k = arbitrary integers
etc.

STRUCTURE/STATEMENTS

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) see WHILE; C has no FOR, "for" in C is another way of writing "while"
(2) only STEP -1 as DOWNTO

VALUES

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) in translating from E to C, you can often simulate them with:

```
myfunc([1,2,3])
```

becomes:

```
int dummy [] = {1,2,3};
myfunc(dummy);
```

OPERATORS

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) see QUOTED EXPRESSIONS
(2) also for others, equivalences between *(x+e) and x[e] hold.
(3) if ARRAY OF <object>
(4) ONLY for giving by reference. otherwise: "[]"

CONSTANTS/TYPES

E	C/C++	Pascal
CONST X=1	#define X 1 const int X=1;	CONST X=1;
ENUM X,Y,Z	#define X 0 (etc.) enum x{X,Y,Z};	TYPE 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) when translating from E to C, simulate with an array of char/int resp., and do your own range-checking etc.
(2) no Wirth Pascal, but available in all popular dialects.
(3) or public class.

QUOTED EXPRESSIONS

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

example:

```
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) really QUOTE, but sometimes used where in LISP LAMBDA would be used, like in MapList()
(2) not even in ProLog, see other logical languages.
(3) lazyness would be used instead here

UNIFICATION AND LISP CELLS

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]`	-	-

EXCEPTIONS

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 handles only one specific exception, it's quite different from general exception handlers as used in E.
(2) the runtime system does raise some exceptions, but I'm not sure whether automatically raised exceptions can be _defined_ in Ada.

OBJECT ORIENTED PROGRAMMING

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)`

[see also next part under NEW]

BUILTIN FUNCTIONS AND MEMORY ALLOCATION
(only a few are presented here, as an example)

E	C/C++	Pascal
`WriteF(fs,...)`	`printf(fs,...);` `enum x{X,Y,Z};`	`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) when translating from C, make sure you turn the arrays of char into proper STRINGs.
(2) dunno what function is needed in the pointer case.

18D. Amiga E FAQ

[94_10_1..94_12_1]

This FAQ-list (Frequently Asked Questions) was compiled by looking through old email and gathering those questions which keep popping up.

Compiler/Linking/Executables

- How can I link E code with other languages? / use standard object format?

  Converting between .m and .o really isn't a huge problem, as one can see from the o2m utility, which allows cooperation between E and Macro-Assembly quite marvelously. The problem with C is in the compiled code, not the file format. Anything but trivial C will reference things like _DOSBase, link-library functions, stack-bases or other things from startup-code that E provides in a quite different way, for example E's startup code is very different from that of C compilers, and dosbase in E is placed on the stack, not in the exe as with C. Even two C compilers may have these problems. If you can get your C compiler to deliver a .o without external references etc., you can link it with E (I did this once with MaxonC++).

- Can I link amiga.lib?

Currently no. To a lesser extend this has the same problems as .o files, however with the help of an assembler and o2m a .lib file can be translated to .m, infact it already has been done for some. It's only a matter of time before someone does this for (parts of) amiga.lib.

- Can E code be made resident?

  Nearly ALL E code is already resident-able without the help of the programmer. As fas as I know, the only E construct that can violate this is a static list [] with an expression in it (so [a], for example. [1] or NEW [a] is ok.).

- I have this application with pieces of inline asm, and after OPTI/S it behaves weird. what is going on?

  Inline asm may use the same registers as the register optimizer does. check E.doc for this.

- I wrote X in both C and E, and the E version is Y times slower/faster. how come?

  It's not easy to compare both in a fair way. Often it may be the case that one of the two has more optimized routines for something (string handling, I/O etc.). Use of a profiler can reveal this. If the code in question only does calculations, there's no reason why any of the two should be significantly slower than the other, if used properly.

- I have written this nice 'myutils.m', only when I use one function from it, EC links all. I don't like that.

  The style of writing modules in E is to keep them as relatively small units, with only related code and data. (This makes more sense if you know that the module is the unit of datahiding in E). Split it up!

- Can I make EC resident?

  no, currently not. EC itself is still written in old-fashioned assembly style, and cannot be made resident ;-)

Resources

- There's no explanation in the docs of all those handy functions from intuition.library etc. How come?

  The functions are not part of E, rather they are part of the amiga OS. As such, they are described by Commodore's documents, not by the E documents. "The AMIGA ROM Kernal Reference Manuals" are a series of books published by Addison Wesley. A good place to start for example "Libraries", ISBN 0-201-56774-1. Other books are worth considering too, an example is "The Amiga Guru Book" by Ralph Babel.

- I'd like to read more source code than what comes with the distribution where should I look?

  There are lots of places too look, but the best is without doubt Aminet (A collection of FTP-sites on the internet), for example ftp.luth.se. In the directory 'pub/aminet/dev/e' you'll find all sorts of E related stuff. Some BBS'es outside of the internet also carry Aminet, and there are even CDROMS available. Another good place to pick up sources and to talk with fellow E programmers is the E mailing list. Send an email with 'HELP' in the body to amigae-request@bkhouse.cts.com to hear all about it. There are E discussion on other nets as well (FidoNet, AmigaNet), public domain disk series (EPD in europe/germany, same address as the german registration site), furthermore there are heaps of user-clubs, BBS's etc. supporting E these days. just look around in your area.

- I have heard about this mailing list. is it any good?

  Find out for yourself. If your serious about E it's definitly a must, also because it's the place where news on E is generally released first.

- I have a hard time getting on the mailing list. Could you take care of that for me?

  I personally don't have anything to do with the administration of the list, so I can't help out any better than anyone else can. Remember that the server is an automated process, so you need to be very precise in what

you send there. Don't send administrative post to the list at large, instead try and get hold of the administrator.

- I'm new to internet. Could you help me get on the mailing list / FTP files from Aminet etc...?

  Stuff like that is way beyond my service. Please try and contact someone locally.

- How do I know what is the latest version / How do I receive it?

  If you're on the mailing list you'll be the first to know. Aminet is the place where releases and updates are uploaded first. Registered users get updates/notices automatically in their mbox.

- How do I register?

  please read E.doc on this.

- Does E exist for other platforms?

  Not Yet. Personally I have undertaken small project to make portable compilers/translators, and several others have E compiler projects on other platforms, but nothing has come out of this so far.

- Can I become an E registration site for country X?

  Generally I pick sites myself, when I feel the necessity, and know someone in that country pretty well.

- Can I start an E support BBS / an E programming club...?

  You can always do that, without asking me. I of course enjoy hearing about efforts like this...

Programming

- Can I do X in E? (where X = {games,dtp-packages,...})

  E is a general purpose programming language, so there shouldn't be any type of program that can't be done in E (with few rare exceptions, which are covered by E's inline asm). This doesn't mean E is especially equipped for certain types of programs, i.e. E has no special functions for games (though its extensibility makes it easy to add them).

- How do I create X in E (where X = {window, interupt handler,...})

  as with the last question, E just opens the possibilities to write anything, and there isn't always a specific function available. In the worst case this means having to use difficult functions from spooky libraries, but it's worth it getting to grips with that. If you're lucky some E programmer already has done something similar which you can learn from.

- Please write me an example how to do X in E.

  Please try to figure it out yourself first, or ask other E programmers to help out.

- The compiler refuses to compile things like myscreen.rastport.bitmap, eventhough x.y.z is generally possible. Why is that?

  to be able to dereference that last .bitmap, the compiler needs to know the type of the expression myscreen.rastport. And if you look at the module 'intuition/screens.m' you'll see that the rastport field has no type.

- ... if that is so, why don't we have new correctly typed modules?

Because the assembly includes (.i) from which the E modules are converted don't contain that information. It's not possible to use C .h files either, since to some extend they use other identifiers, and thus would break backward compatability.

- ... so what do I do then?

  The classical method was to load x.y into a typed pointer, and then reference .z with that. With pointer typing in v3, you can also dereference it directly (go and see E.doc).

- I'm writing code like this:

```
DEF s[100]:STRING
s:='my beautiful string'
```

  which is allowed by the compiler, but later gives me problems. What can possibly be wrong with this?

  If you're used to for example BASIC, you're used to handle strings in thesame way as integers, as they are both _values_. In E however there are no real string variables in the BASIC sense, the DEF above creates a piece of memory to hold a string, then sets the variable as a pointer to this memory. The pointer and the memory are implementation-wise two unrelated entities. From that DEF on, all operation that directly access 's' access the pointer, which is just an integer which tells us were to find the actual string. The assignment thus puts the address of 'my beautiful string' into 's', overwriting the old value. The string-memory created by the DEF sits there unaltered, and now unaccessable because no pointer points to it. Functions like StrCopy() can use this pointer to find the real memory, and can fill it:

```
StrCopy(s,'my beautiful string')
```

  is correct. Of course assigning strings as pointers also has a use, for example if you just want to read the string data and do nothing else with it. then:

```
DEF s:PTR TO CHAR
```

  allocates no memory for a string, only the pointer. the assignment above would now make sense. This all boils down to the differences between pointers and values (or value/reference semantics), and is important to understand to succesfully program in E.

- How do I return a STRING / OBJECT etc. from a function?

  depends. If you just wish to use it as temporary return value, give a string as argument and let the routine fill it. If you need to create a NEW string/ object, allocate it dynamically within the function, and return the pointer.

<more of these specific questions are to be added>

Bugs

- I remember programming something when suddenly the compiler crashed/misbehaved/produced an internal error/gave wrong error messages. shouldn't you be doing something about this?

  If I have no clue where to look for a bug I can't fix it. If you stumble across something that you're sure of is a compiler bug, make a copy of the source that reproduces the bug (possibly cut the source down while making sure it still shows the bug) and send it to me with as much info as possible. EC-enforcer hits for example are very useful.

- I wrote program X, but it crashes. I'm positive I made no mistakes, so certainly this must be a compiler bug.

  due to E's untyped-ness, you can never be quite sure you're code is correct. Most code send to me with comments like the above later proved to be errors in the code, mostly due to lock of E knowledge. Read the docs, and use EDBG!

Future Features

Will E have...

- Registerised arguments?

  It surely is possible but like many things it's not a priority for me.

- Library/Device linker?

  I would have done this already if it weren't for E's way of storing global variables (on the stack), which complicates things a lot. This'll just need a little time.

- Multiple Inheritance?

  Not possible in E. MI breaks with structural compatability between objects, and needs relatively strong typing to resolve this.

- PROCs inside PROCs?

  Not likely.

- multidimensional arrays?

  Not likely either. In E there isn't really something like an array, just pointers that point to a large amounts of equally sized objects. To have twodimensional arrays, one needs to have the concept of 'size' of an array, which E doesn't have.

- more 020/881 support?

  Yes, eventually. It just isn't at the top of my TODO-list.

- C compatible syntax?

  Often people are used to some type of language (mostly C), and don't understand why a programming language design isn't as configurable as a directory utility or text editor:
  'I like "=="/"=" better than "="/":="'
  'Can't you switch """ and "'" ?'
  'Why doesn't "--" work as in C?'
  'I hate typing CAPS for keywords!'
  'No precedence sucks!'
  The features 'referenced' above will never change, and you'd better get used to it. All design decisions have very good reasons, and unlike some rumours they were not done for compiler speed (infact implementing them more traditionally wouldn't make the compiler any slower).

- Why is feature X in E not like language Y, which I find better.

  Like above, the design of a language isn't changed just because of personal preference. If you have strong ideas of what a language should look like according to you, and those ideas stem from language Y, then use Y. If you can't find your ideas in any existing language, then it's time to design and implement your own! (I'm not kidding, it's worth the trouble! and you can always start with E... :-)

- X maybe?

  Having seen literally hundreds of programming languages, your chances of suggesting a new feature for E which I haven't thought of before are not too bright. Many things don't 'fit' into E, and in general time is limited to add more fancy stuff. (If you look closely, the amount of features in E is already quite high for an average programming language). I have quite a huge list of possible features for E, and eventually some of these will be implemented. just wait and see...

----------EOF----------