Reference

COLLABORATORS			
	TITLE : Reference		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 7, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Reference

1.1 E Language Reference Guide

```
                  +-----------------------------------------------+
    |           |
    |                Amiga E v2.1b                    |
    |          Compiler for The E Language            |
    |           By Wouter van Oortmerssen             |
    |           |
    |              Language Reference                 |
    |           |
    +-----------------------------------------------+
```

Contents:

1.2 E Language Reference: Chapter One

```
                +-----------------------------------------------------------------+
|                             FORMAT        |
+-----------------------------------------------------------------+

Section A:
                Tabs,lf etc.
                Section B:
                Comments
                Section C:
                Identifiers and types
```

1.3 E Language Reference: Chapter One, Section A

```
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 particulary many arguments, separated by commas ",", may
be spread over several lines by ending a line with a comma, thus
ignoring the following <lf>.
Any lexical element in a source code file may be separated from another
by any number of spaces, tabs etc.
```

1.4 E Language Reference: Chapter One, Section B

```
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.
```

1.5 E Language Reference: Chapter One, Section C

```
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()
```

1.6 E Language Reference: Chapter Two

```
                +---------------------------------------------------------------+
|                        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.

Section A:
                Decimal (1)
                Section B:
                Hexadecimal ($1)
                Section C:
                Binary (%1)
                Section D:
                Float (1.0)
                Section E:
                Character (''a'')
                Section F:
                Strings ('bla')
                Section G:
                Lists ([1,2,3]) and typed lists
```

1.7 E Language Reference: Chapter Two, Section A

```
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
```

1.8 E Language Reference: Chapter Two, Section B

```
hexadecimal ($1)
----------------
A hexadecimal value uses the additional characters "A" .. "F" (or
"a" .. "f") and is preceded by a "$" character.
Examples: $FC, $DFF180, -$ABCD
```

1.9 E Language Reference: Chapter Two, Section C

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

1.10 E Language Reference: Chapter Two, Section D

```
                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 (FFP)
representation.  See
                chapter 12
                 for more information on floats.
Examples: 3.14159, .1 (=0.1), 1. (=1.0)
```

1.11 E Language Reference: Chapter Two, Section E

```
character ("a")
---------------
The value of a character (enclosed in double "" quotes) is their
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).
```

1.12 E Language Reference: Chapter Two, Section F

```
string ('bla')
--------------
Strings are any ascii representation, enclosed in single '' quotes.
The value of such a string is a pointer to the first character of it.
More specifically, '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 0 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)
\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

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)

For example, to 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'
```

1.13 E Language Reference: Chapter Two, Section G

```
                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 initialized 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 the part on
list-functions
for a discussion on typed-immediate
lists and detailed information.

1.14 E Language Reference: Chapter Three

```
                +----------------------------------------------------------------+
|                           EXPRESSIONS       |
+-----------------------------------------------------------------+
```

Section A:
Format
Section B:
Precedence and grouping
Section C:
Types of expressions
Section D:
Function calls

1.15 E Language Reference: Chapter Three, Section A

format

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

- immediate values as discussed in
chapter 2
- operators as discussed in
chapter 4
- function calls as discussed in
chapter 3D
- brackets () as discussed in
chapter 3B
- variables or variable-expressions (see

3C
)

examples of expressions:
1
'hello'
$ABCD+(2*6)+Abs(a)
(a<1) OR (b>=100)

1.16 E Language Reference: Chapter Three, Section B

precedence and grouping

The E language has _no_ precedence whatsoever. This means that
expressions are evaluated left to right. You may change
precedence by bracketing some (sub-)expression:
1+2*3 /* =9 */ 1+(2*3) /* =7 */ 2*3+1 /* =7 */

1.17 E Language Reference: Chapter Three, Section C

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) or [] (array operator). For those,
  see chapters
  4D
  and
  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.

1.18 E Language Reference: Chapter Three, Section D

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 brackets () enclosing
zero to unlimited arguments, separated by commas ",".
Note that arguments to functions are again expressions.
See
chapter 6
on how to make your own functions, and chapters

9
and
10
on built-in functions. Examples:

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

1.19 E Language Reference: Chapter Four

```
                +-----------------------------------------------------------------+
|                           OPERATORS       |
+-----------------------------------------------------------------+
```

Section A:
Math (+ - * /)
Section B:
Comparison (= <> > < >= <=)
Section C:
Logical and bitwise (AND OR)
Section D:
Unary (SIZEOF ` ^ {} ++ -- -)
Section E:
Triple (IF THEN ELSE)
Section F:
Structure (.)
Section G:
Array ([])
Section H:
Float operator (|)
Section I:
Assignments expressions (:=)
Section J:
Sequencing (BUT)

1.20 E Language Reference: Chapter Four, Section A

math (+ - * /)

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

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

see
chapter 12
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

```
                Mul()
```

1.21 E Language Reference: Chapter Four, Section B

```
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.
```

1.22 E Language Reference: Chapter Four, Section C

```
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 */
```

1.23 E Language Reference: Chapter Four, Section D

```
                unary (SIZEOF ^ {} ++ -- `)
---------------------------
- SIZEOF <objectIdent>
  Simply returns the size of a certain object.
  Example: SIZEOF newscreen

- {<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
```

1.24 E Language Reference: Chapter Four, Section E

```
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')
```

1.25 E Language Reference: Chapter Four, Section F

```
                structure (.)
-------------
<ptr2object>.<memberofobject> builds a <varexp>
The pointer has to be declared as PTR TO <object> or ARRAY OF <object>
(see
                chapter 8
                 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
```

1.26 E Language Reference: Chapter Four, Section G

```
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, with
little limitations that it should not contain function calls and the like
when used on the left-hand side of an assignment.

Note1: "[]" is a shortcut for "[0]"

Note2: with an array of n elements, the index may be  0 .. n-1
Examples:

a[1]:=10           /* sets second element to 10 */
x:=table[y*4+1]    /* reads from array */
```

1.27 E Language Reference: Chapter Four, Section H

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

1.28 E Language Reference: Chapter Four, Section I

```
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()
```

1.29 E Language Reference: Chapter Four, Section J

```
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.
```

1.30 E Language Reference: Chapter Five

```
                +----------------------------------------------------------------+
|                           STATEMENTS        |
+----------------------------------------------------------------+

Section A:
                Format (;)
                Section B:
                Statement labels and gotos (JUMP)
                Section C:
                Assignment (:=)
                Section D:
                Assembly mnemonics
                Section E:
                Conditional statement (IF)
                Section F:
                For-statement (FOR)
                Section G:
                While-statement (WHILE)
                Section H:
                Repeat-statement (REPEAT)
                Section I:
                Loop-statement (LOOP)
                Section J:
                Select-case-statement (SELECT)
                Section K:
                Increase statement (INC/DEC)
                Section L:
                Void expressions (VOID)
```

1.31 E Language Reference: Chapter Five, Section A

```
                format (;)
----------
As suggested in
                chapter 1A
                , a statement generally stands in its
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
                chapters 5E - 5K
                - void expressions
- labels
- assembly instruction

The comma is the primary character to show that you do not wish to
end the statement with the next linefeed, but the following characters
also signal a continuation of a statement on the next line:

+ - * /
= > < <> >= <=
AND OR BUT THEN
```

1.32 Chapter Five, Sections E-K

```
                Section E:
                Conditional statement (IF)
                Section F:
                For-statement (FOR)
                Section G:
                While-statement (WHILE)
                Section H:
                Repeat-statement (REPEAT)
                Section I:
                Loop-statement (LOOP)
                Section J:
                Select-case-statement (SELECT)
                Section K:
                Increase statement (INC/DEC)
```

1.33 E Language Reference: Chapter Five, Section B

```
statement labels and gotos (JUMP)
---------------------------------
```

```
Labels are global-scoped identifiers with a ':' added to it, 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), 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 Mouse()=1 THEN JUMP stopnow

/* other parts of program */

stopnow:
```

1.34 E Language Reference: Chapter Five, Section C

```
assignment (:=)
---------------
The basic format of an assignment is:   <var> := <exp>
Examples: a:=1,  a:=myfunc(),  a:=b*3
```

1.35 E Language Reference: Chapter Five, Section D

```
                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
                chapter 15
                 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 */
```

1.36 E Language Reference: Chapter Five, Section E

```
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.
```

1.37 E Language Reference: Chapter Five, Section F

```
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.  Example:

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

1.38 E Language Reference: Chapter Five, Section G

```
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.
```

1.39 E Language Reference: Chapter Five, Section H

```
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!')
```

1.40 E Language Reference: Chapter Five, Section I

```
loop-statement (LOOP)
---------------------
LOOP, ENDLOOP

syntax:   LOOP
      <statements>
    ENDLOOP

builds an infinite loop.
```

1.41 E Language Reference: Chapter Five, Section J

```
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
```

1.42 E Language Reference: Chapter Five, Section K

```
increase statement (INC/DEC)
----------------------------
INC, DEC

syntax:   INC <var>
    DEC <var>

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

1.43 E Language Reference: Chapter Five, Section L

```
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.
```

1.44 E Language Reference: Chapter Six

```
                +----------------------------------------------------------------+
|              FUNCTION DEFINITIONS AND DECLARATIONS   |
+----------------------------------------------------------------+

Section A:
                Proc definition and arguments (PROC)
                Section B:
                Local and global definitions: scope (DEF)
                Section C:
                Endproc/return
                Section D:
                The 'main' function
                Section E:
                Built-in system variables
```

1.45 E Language Reference: Chapter Six, Section A

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 one return value.

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
chapter 8
) 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  */
```

1.46 E Language Reference: Chapter Six, Section B

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>,...
description:  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
chapter 8
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
chapter 8B
.
A declaration of a basic type can have an Initialization, 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.

1.47 E Language Reference: Chapter Six, Section C

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 also CleanUp() in chapter 9F.

```
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>
```

These are function definitions that only make small computations, like faculty functions and the like: (one-liners :-)

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

1.48 E Language Reference: Chapter Six, Section D

```
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()
```

1.49 E Language Reference: Chapter Six, Section E

```
                built-in system variables
-------------------------
Following global variables are always available in your program,
they're called system variables.

arg   "arg" 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.

conout    This is where that file handle is kept, and the console
    window will be automatically closed upon exit of your
    program.  See
                WriteF()
                 on how to use these two variables
    properly.

execbase, These five variables are always provided with their
dosbase,  correct values.
gfxbase,
intuitionbase,
mathbase

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 sources/examples dir how to
    make good use of wbmessage.
```

1.50 E Language Reference: Chapter Seven

```
                +-----------------------------------------------------------------+
|                   DECLARATION OF CONSTANTS      |
+-----------------------------------------------------------------+
```

Section A:
Const (CONST)
Section B:
Enumerations (ENUM)
Section C:
Sets (SET)
Section D:
Built-in constants

1.51 E Language Reference: Chapter Seven, Section A

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.

1.52 E Language Reference: Chapter Seven, Section B

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
```

1.53 E Language Reference: Chapter Seven, Section C

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 initialize 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 two properties hold:

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

1.54 E Language Reference: Chapter Seven, Section D

```
                built-in constants
------------------
Following are built-in constants that may be used:

TRUE,FALSE  Represent the boolean values (-1,0)
NIL   (=0), the uninitialized pointer.
ALL   Used with string functions like StrCopy() to copy all characters
GADGETSIZE  Minimum size in bytes to hold one gadget; see
                Gadget()
                 in 9D
OLDFILE,NEWFILE Mode-parameters for use with Open()
STRLEN    Always has the value of the length of the last immediate
    string used.  Example:

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

1.55 E Language Reference: Chapter Eight

```
                +-----------------------------------------------------------------+
|                            TYPES        |
+-----------------------------------------------------------------+

Section A:
                About the 'type' system
                Section B:
                The basic type (LONG/PTR)
                Section C:
                The simple type (CHAR/INT/LONG)
                Section D:
                The array type (ARRAY)
```

```
Section E:
The complex type (STRING/LIST)
Section F:
The compound type (OBJECT)
Section G:
Initialization
```

1.56 E Language Reference: Chapter Eight, Section A

```
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
- flexible way of programming: no problem that some types of return values
  don't match, no superfluous "casts" etc.
- no hard to find errors when mixing data of different sizes in expressions
- still benefit of self-documenting types, if you wish, like:

  PTR to newscreen
```

1.57 E Language Reference: Chapter Eight, Section B

```
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
```

1.58 E Language Reference: Chapter Eight, Section C

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.

1.59 E Language Reference: Chapter Eight, Section D

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
initialized.
```

1.60 E Language Reference: Chapter Eight, Section E

```
                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 is backwards compatible with PTR TO CHAR and
  of course ARRAY OF CHAR, but not the other way around.
  See the section on
                string functions
                 for more details.

- LISTs. This is a datatype not found in other procedural languages, it
  is something found in languages like ProLog and Lisp.  The E version
  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 as 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 chapters
                2G
                 and
                9C
                 for more on this.
```

1.61 E Language Reference: Chapter Eight, Section F

```
the compound type (OBJECT)
--------------------------
  OBJECTs are much like a struct in 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 a simple or again a compound type, or a simple
array type, i.e  [<numelements>]:ARRAY with the default CHAR size for
an element.  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++


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.

NOTE: OBJECTs in E are not like you might be used to in other
languages.  For example, not just any type can form a member of
an object, and because of that, recursive object accesses like x.y.z
don't make much sense (yet).
```

1.62 E Language Reference: Chapter Eight, Section G

```
Initialization
--------------

1. Always initialized 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. initialized to '' and [] respectively

   - global and local STRINGs
   - global and local LISTs

3. Not initialized
```

```
   - local variables (unless explicitly stated)
   - global and local ARRAYs
   - global and local OBJECTs
```

1.63 E Language Reference: Chapter Nine

```
                +-----------------------------------------------------------------+
|                       BUILT-IN FUNCTIONS      |
+-----------------------------------------------------------------+

Section A:
                I/O functions
                Section B:
                Strings and string functions
                Section C:
                Lists and list functions
                Section D:
                Intuition support functions
                Section E:
                Graphics support functions
                Section F:
                System support functions
                Section G:
                Math and other functions
                Section H:
                String and list linking functions
```

1.64 E Language Reference: Chapter Nine, Section A

```
                io functions
------------

  WriteF(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 the bit about
                strings
                 for more.

NOTE: If stdout=NIL, perhaps because 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 the 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.

  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.

  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:
                string support
                  oldout:=SetStdOut(newstdout)

Sets the standard output variable 'stdout'.  Equivalent to:

oldout:=stdout; stdout:=newstdout
```

1.65 E Language Reference: Chapter Nine, Section B

```
                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,NIL)                      /* 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)
```

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

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

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

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

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

```
  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.

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

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

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

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.

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

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 (<-!!!). 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 var {read} a 0:

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

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

searches string1 for the occurrence of string2, possibly starting from another position than 0. Returned is the distance from [0] at which the substring was found, else -1. 'startpos' does not reflect an address.

```
  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.

```
  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.

```
  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.

For string linking functions see
chapter 9H
.

1.66 E Language Reference: Chapter Nine, Section C

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 a ready initialized 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. Once you get to grips with lists, you'll never want to write a program without them.

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 initialized equivalent of ARRAY OF <type>, if <type> is an object-name, you'll be building initialized 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)
```

Copies num elements from list to listvar. Example:

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

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

Copies num items of list to the tail of listvar.

```
  ListCmp(list,list,num)
```

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
chapter 11C
.
For list linking functions see
chapter 9H
.

1.67 E Language Reference: Chapter Nine, Section D

intuition support functions

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

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)
```

opens a custom screen for you. Depth is number of bitplanes (1-6, 1-8 AGA),
sflags is something like 0, or $8000 for hires (add 4 for interlace).

```
  CloseS(sptr)
```

as CloseW(), now for screens.

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

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)
```

See examples like SuperVisor.e for a real-life example.

```
  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.

```
  x:=MouseX(win)      and     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
```

1.68 E Language Reference: Chapter Nine, Section E

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)
```

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)
```

draws a line.

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

draws a box.

```
  Colour(foreground,background)
```

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,...)
```

is exactly the same function as WriteF(), only outputs to some (x,y) on
your stdrast, instead of stdout. See:
WriteF()
and
strings
in the language
reference.

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

changes the output rastport of the e graphics functions.

```
  SetTopaz(size)
```

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 skrew up your window layout.
'size' is of course 8 or 9.

1.69 E Language Reference: Chapter Nine, Section F

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.

```
  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)
```

Exits the program from any point. It is the replacement for the DOS call Exit(): never use that one! Instead use CleanUp(), which allows for the deallocation of memory, closing libraries correctly etc.
The return value will be given to dos as returncode.

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

returns the amount of free stack space left. This should always be 1000 or more. See the chapter
implementation issues
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) RETURN IF n=1 THEN 1 ELSE fac(n-1)*n
```

would be less safe.

1.70 E Language Reference: Chapter Nine, Section G

```
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)

returns TRUE or FALSE if some expression is Odd or Even.

  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 integer from 0..999

To initialize 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.

  a:=Mod(b,c)

divides 32bit 'b' by 16bit 'c' and returns 16bit modulo 'a'.

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

shifts 'y' 'num' bits to left or right.

  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) and    PutInt(adr,a)        PutChar(adr,a)

Pokes value 'a' into memory.  See 'Long()' above.
```

1.71 E Language Reference: Chapter Nine, Section H

```
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.

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 'D.e'

1.72 E Language Reference: Chapter Ten

```
                +----------------------------------------------------------------+
|                  LIBRARY FUNCTIONS AND MODULES    |
+-----------------------------------------------------------------+
```

Section A:
Built-in library calls
Section B:
Interfacing to the amiga system with the 2.04 modules

1.73 E Language Reference: Chapter Ten, Section A

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 "Initialization code", always opens the four libraries Intuition, Dos, Graphics and Mathffp, and because of this, the compiler has all the calls to those five libraries (including Exec) integrated in the compiler (there are a few hundred of them). These are up to AmigaDos v2.04, v3.00 should be included by the next version of Amiga E. 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.

1.74 E Language Reference: Chapter Ten, Section B

interfacing to the amiga system with the 2.04 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 be simply be viewed; they need a utility like ShowModule (see utility.doc) 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 filerequestr
  IF aslbase:=OpenLibrary('asl.library',37)
    IF req:=AllocFileRequest()
      IF RequestFile(req) THEN WriteF('File: "\s" in "\s"\n',req.file,req.dir)
      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 initialized by the programmer. From 'libraries/Asl', it takes the definition of the filerequestr 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?

1.75 E Language Reference: Chapter Eleven

```
                +----------------------------------------------------------------+
|                      QUOTED EXPRESSIONS     |
+----------------------------------------------------------------+
```

Section A:
Quoting and scope
Section B:
Eval()
Section C:
Built-in functions

1.76 E Language Reference: Chapter Eleven, Section A

```
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 x3 when evaluated.
These pointers to functions are very different from normal PROCs, and
you should never mix the two up.  The biggest difference is that a
quoted expression is just a simple expression, and thus cannot have its
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.
```

1.77 E Language Reference: Chapter Eleven, Section B

```
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 initialize 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) lambda's.
```

1.78 E Language Reference: Chapter Eleven, Section C

```
                built-in functions
------------------
  MapList(var,list,listvar,func)

performs some function on all elements of 'list' and returns all
results in 'listvar'.  'func' must be a quoted expression (see
                previous
                ),
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]

  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.

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()
  LOCAL mem[4]:LIST,x
  MapList({x},[200,80,10,2500],mem,`AllocVec(x,0)) /* alloc some */
```

```
  IF ForAll({x},mem,`x)                              /* suxxes ? */
    WriteF('Yes!\n')
  ELSE
    WriteF('No!\n')
  ENDIF
  ForAll({x},mem,`IF x THEN FreeVec(x) ELSE NOP)    /* 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.

1.79 E Language Reference: Chapter Twelve

```
                +-----------------------------------------------------------------+
|                    FLOATING POINT SUPPORT     |
+-----------------------------------------------------------------+
```

Section A:
Using floats and float operator overloading
Section B:
Float expressions and conversion

1.80 E Language Reference: Chapter Twelve, Section A

using floats and float operator overloading

Overloading the standard operators + * etc with float equivalents is
possible starting from v2.0 of Amiga E, but I've removed the main
documentation on it because it is likely that the float-concept in E
will change as of v2.2 or later; that version may allow for 68881 inline
code generation next to normal FFP routines in a transparent fashion.

If you really want to use floats with v2.1b, you are advised to use the
SpXxx() built-in routines from the mathffp.library. Example:

```
x:=SpMul(y,0.013483)
```

Be aware that when v2.5 comes out, your sources may need to be
changed (for the better!).

1.81 E Language Reference: Chapter Twelve, Section B

float expressions and conversion
----
as
12A
.

1.82 E Language Reference: Chapter Thirteen

```
                +----------------------------------------------------------------+
|                        EXCEPTION HANDLING     |
+----------------------------------------------------------------+

Section A:
                Defining exception handlers (HANDLE/EXCEPT)
                Section B:
                Using the Raise() function
                Section C:
                Defining exceptions for built-in functions (RAISE/IF)
                Section A:
                Use of exception-ID's
```

1.83 E Language Reference: Chapter Thirteen, Section A

```
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 occur, 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 occur, 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.
```

1.84 E Language Reference: Chapter Thirteen, Section B

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

  Raise(exceptionID)

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)).
```

1.85 E Language Reference: Chapter Thirteen, Section C

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? 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 */
  FreeMem(mem,SIZE)   /* we'll never get here */
EXCEPT
  IF mem THEN FreeMem(mem,SIZE)
  Raise(exception)    /* 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.
```

1.86 E Language Reference: Chapter Thirteen, Section D

```
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 expandable 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 need 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
  impractical to have to coordinate the ID's 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 supply 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 ID's of other procedures, the task of the programmer
that has to deal with the exceptions will be greatly simplyfied.

Examples:

```
(system)

"MEM"   out of memory
"FLOW"  (nearly) stack overflow
"^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)

"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)

(dos)

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

The general tendancy is uppercase for general system
exceptions, and lowercase (and mixed) for specific modules.

```
- all others (including all negative ID's) remain reserved.
```

1.87 E Language Reference: Chapter Fourteen

```
+-----------------------------------------------------------------+
|                         OO PROGRAMMING       |
+-----------------------------------------------------------------+

As this hasn't been implemented yet, it's not documented either.
```

1.88 E Language Reference: Chapter Fifteen

```
                +-----------------------------------------------------------------+
|                        INLINE ASSEMBLY       |
+-----------------------------------------------------------------+

Section A:
                Identifier sharing
                Section B:
                The inline assembler compared to a macro assembler
                Section C:
                Ways using binary data (INCBIN/CHAR..)
                Section D:
                OPT ASM
```

1.89 E Language Reference: Chapter Fifteen, Section A

```
                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 */
```

1.90 E Language Reference: Chapter Fifteen, Section B

```
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.
- No support for LARGE model/reloc-hunks in assembly _YET_.
  This means practically that you have to use (PC)-relative addressing
  for now.
```

1.91 E Language Reference: Chapter Fifteen, Section C

```
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
```

1.92 E Language Reference: Chapter Fifteen, Section D

```
                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 from 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.
Example:

```
; OPT ASM

start:  MOVEQ #1,D0   ; /* do something silly */
  RTS     ; /* and exit */
```

This could be assembled by any assembler, including EC.

1.93 E Language Reference: Chapter Sixteen

```
                +----------------------------------------------------------------+
|                      IMPLEMENTATION ISSUES      |
+----------------------------------------------------------------+
```


1.94 E Language Reference: Chapter Sixteen, Section A

the OPT keyword

OPT, LARGE, STACK, ASM, NOWARN, DIR, OSVERSION

syntax: OPT <options>,...

allows you to change some compiler settings:

LARGE Sets code and data model to large. Default is small; the compiler generates 100% pc-relative code, with a

```
    max-size of 32k.  With LARGE, there are no such limits,
    and reloc-hunks are generated.  See
                -l
                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.

ASM   Set the compiler to assembly mode.  From there on, only
    assembly instructions are allowed, and no Initialization
    code is generated.  See: chapter
                inline assembly
                NOWARN    Shut down warnings.  The compiler will warn you if it
    *thinks* your program is incorrect, but still syntactically
    ok.  See
                -n
                .

DIR=moduledir Sets the directory where the compiler searches for modules.
    Default is '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 Initialization code when running on an older machine.
    However, checking the version yourself and giving an
    appropriate error-message is more helpful for the user.

example:

OPT STACK=20000,NOWARN,DIR='df1:modules',OSVERSION=39
```

1.95 E Language Reference: Chapter Sixteen, Section B

```
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 recognize 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

in your code.
```

1.96 E Language Reference: Chapter Sixteen, Section C

```
stack organization
------------------
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 PROC's are called in a linear fashion, there's
no way the stack will overflow.  Example:

global data:    10k (arrays e.d)
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.
```

1.97 E Language Reference: Chapter Sixteen, Section D

```
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

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

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

buffersize of generated code and identifiers  relative to source
buffersize of labels/branches and intermediate  independently (re)allocated
```

1.98 E Language Reference: Chapter Sixteen, Section E

```
                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.

However, the compilor may feel obliged to give you either
                warning
                 or
                error
                messages.  Note that sometimes the compilor will refer to a line ↩
                   that
represents the beginning of a linked set of lines, as in:

'blah blah blah' +    /* This line gets the blame for the next line's error. */
'blah blah blah blah
'blah blah blah blah'
```

1.99 Chapter Sixteen, Section E, Warning Messages

```
                warning messages
----------------

-
                A4/A5 used in inline assembly
                -
                keep an eye on your stacksize
                -
                stack is definitely too small
                -
                suspicious use of '=' in void expressions
```

1.100 Warning Messages: A4/A5 used in inline assembly

- "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.

1.101 Warning Messages: Stack-short related warnings

- Stack-short related warnings

These may be issued when you use OPT STACK=<size>. The compiler will simply match your <size> against its own estimate (see chapter 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.

1.102 Warning Messages: suspicious use of '=' in void expressions

- 'suspicious use of "=" in void expressions'

  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.

1.103 Chapter Sixteen, Section E, Error Messages

error messages

- syntax error
- unknown keyword/const
- ':=' expected
- unexpected characters in line
- label expected
- ',' expected
- variable expected
-

```
value does not fit in 32 bit
-
missing apostrophe/quote
-
incoherent program structure
-
illegal command-line option
-
division and multiplication 16bit only
-
superfluous items in expression/statement
-
procedure 'main' not available
-
double declaration of label
-
unsafe use of '*' or '/'
-
reading sourcefile didn't succeed
-
writing executable didn't succeed
-
no args
-
unknown/illegal addressing mode
-
unmatched parentheses
-
double declaration
-
unknown identifier
-
incorrect #of args or use of ()
-
unknown e/library function
-
illegal function call
-
unknown format code following '/'
-
/* not properly nested comment structure */
-
could not load binary
- '}'
expected
-
immediate value expected
-
incorrect size of value
-
no e code allowed in assembly modus
-
illegal/inappropriate type
-
']' expected
-
statement out of local/global scope
```

```
-
could not read module correctly
-
workspace full!
-
not enough memory while (re-)allocating
-
incorrect object definition
-
illegal use of/reference to object
-
incomplete if-then-else expression
-
unknown object identifier
-
double declaration of object identifier
-
reference(s) out of 32k range: switch to LARGE model
-
reference(s) out of 256 byte range
-
too sizy expression
-
incomplete exception handler definition
```

1.104 Error Messages: syntax error

```
- '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.
```

1.105 Error Messages: unknown keyword/const

```
- '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.
```

1.106 Error Messages: ':=' expected

```
- '":=" expected'
  ---------------
  You have written a FOR statement or an assignment, and put something
```

```
other than ":=" in its place.
```

1.107 Error Messages: unexpected characters in line

```
- 'unexpected characters in line'
  -------------------------------
  You used characters that have no syntactic meaning in E outside of
  a string.  Examples: "@!&\~"
```

1.108 Error Messages: label expected

```
- 'label expected'
  ----------------
  At some places, for example after the PROC or JUMP keyword,
  a label identifier is required.  You wrote something else.
```

1.109 Error Messages: ',' expected

```
- '","  expected'
  --------------
  In specifying a list of items (for example a parameter list)
  you wrote something else instead of a comma.
```

1.110 Error Messages: variable expected

```
- 'variable expected'
  -------------------
  This construction requires a variable, example:

  FOR <var>:= ... etc.
```

1.111 Error Messages: value does not fit in 32 bit

```
              - 'value does not fit in 32 bit'
  ------------------------------
  In specifying a constant value (see
              chapter 2A - 2E
              ) you wrote too
  large a number, examples:  $FFFFFFFFF, "abcdef".

  Also occurs when you define a SET of more than 32 elements.
```

1.112 Error Messages: missing apostrophe/quote

```
- 'missing apostrophe/quote'
  --------------------------
  You forgot the ' at the other end of a string.
```

1.113 Error Messages: incoherent program structure

```
- 'incoherent program structure'
  ------------------------------
  * You started a new PROC before ending the last one.
  * You didn't nest your loops properly, for example:

    FOR
      IF
      ENDFOR
    ENDIF
```

1.114 Error Messages: illegal command-line option

```
- 'illegal command-line option'
  -----------------------------
  In specifying 'EC -opt source' you wrote something for '-opt'
  that is not a legal option to EC.
```

1.115 Error Messages: division and multiplication 16bit only

```
              - '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
              Mul() and Div()
              .
```

1.116 Error Messages: superfluous items in expression/statement

```
- '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.
```

1.117 Error Messages: procedure 'main' not available

```
- 'procedure "main" not available'
  --------------------------------
  Your program does not include a 'main' procedure!
```

1.118 Error Messages: double declaration of label

```
- 'double declaration of label'
  -----------------------------
  You declared a label twice, for example:

  label:
  PROC label()
```

1.119 Error Messages: unsafe use of '*' or '/'

```
              - 'unsafe use of "*" or "/"'
  --------------------------
  This again has to do with 16bit instead of 32bit * and /.

  See
              division and multiplication 16bit only
              .
```

1.120 Error Messages: reading sourcefile didn't succeed

```
- "reading sourcefile didn't succeed"
  -----------------------------------
  Check your source spec. that you gave with 'ec mysource'
  make sure the file ends in '.e', and your command line doesn't.
```

1.121 Error Messages: writing executable didn't succeed

```
- "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.
```

1.122 Error Messages: no args

```
- 'no args'
  "USAGE: ec [-opts] <sourcecodefilename> (`.e' is added)"
  --------------------------------------------------------
  You get this by just typing 'ec' without any arguments.
```

1.123 Error Messages:unknown/illegal addressing mode

```
- '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.
```

1.124 Error Messages: unmatched parentheses

```
- 'unmatched parentheses'
  -----------------------
  Your statement has more "(" than ")" or the other way around.
```

1.125 Error Messages: double declaration

```
- 'double declaration'
  --------------------
  One identifier is used in two or more declarations.
```

1.126 Error Messages:unknown identifier

```
- 'unknown identifier'
  --------------------
  An identifier is not used in any declaration; it is unknown.
  You probably forgot to put it in a DEF statement.
```

1.127 Error Messages: incorrect #of args or use of ()

```
- 'incorrect #of args or use of ()'
  ---------------------------------
  * You forgot to put "(" or ")" at the right spot.
  * You supplied the incorrect #of arguments to some function.
```

1.128 Error Messages: unknown e/library 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.
```

1.129 Error Messages: illegal function 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!'))
```

1.130 Error Messages: unknown format code following '\'

```
              - 'unknown format code following "\"'
  -----------------------------------
  You specified a format code in a string which is illegal.
  See
              chapter 2F
               for a listing of format codes.
```

1.131 Error Messages: /* not properly nested comment structure */

```
- '/* not properly nested comment structure */'
  ---------------------------------------------
  The #of '/*' is unequal to the #of '*/', or is placed in a funny order.
```

1.132 Error Messages: could not load binary

```
- 'could not load binary'
  -----------------------
  <filespec> in INCBIN <filespec> could not be read.
```

1.133 Error Messages: '}' expected

```
- '"}" expected'
  --------------
  You started an expression with "{<var>" , but forgot the "}".
```

1.134 Error Messages: immediate value expected'

```
- '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 */
```

1.135 Error Messages: incorrect size of value'

```
- '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 */
```

1.136 Error Messages: no e code allowed in assembly modus

```
- '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.
```

1.137 Error Messages: illegal/inappropriate type

```
- '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
```

1.138 Error Messages: ']' expected

```
- '"]" expected'
  --------------
  You started with "[", but never ended with "]".
```

1.139 Error Messages: statement out of local/global scope

```
- '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.
```

1.140 Error Messages: could not read module correctly

```
- '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'
```

1.141 Error Messages: workspace full!

```
- 'workspace full!'
  -----------------
  Rarely occurs.  If it does, you'll need the '-m' 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.
```

1.142 Error Messages: not enough memory while (re-)allocating

```
- '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 some memory expansion (ahum).
```

1.143 Error Messages: incorrect object definition

```
              - 'incorrect object definition'
  -----------------------------
  You were being silly while writing the definitions between OBJECT and
  ENDOBJECT.  See
```

```
                chapter 8F
                 to find out how to do it right.
```

1.144 Error Messages: illegal use of/reference to object

```
- '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.
```

1.145 Error Messages: incomplete if-then-else expression

```
                - 'incomplete if-then-else expression'
  ------------------------------------
  If you use IF as an operator (see
                chapter 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.
```

1.146 Error Messages: unknown object identifier

```
- 'unknown object identifier'
  ---------------------------
  You used an identifier that was recognized 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.
```

1.147 Error Messages: double declaration of object identifier

```
- 'double declaration of object identifier'
  -----------------------------------------
  One identifier used in two object definitions.
```

1.148 Error Messages: reference(s) out of 32k range: switch to LARGE model

```
              - '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
              chapter 16B
```

1.149 Error Messages: reference(s) out of 256 byte range

```
- 'reference(s) out of 256 byte range'
  ------------------------------------
  You probably wrote BRA.S or Bcc.S over too great a distance.
```

1.150 Error Messages: too sizy expression

```
- 'too sizy expression'
  ---------------------
  You used a list [], possibly recursive [[]], that is too sizy.
```

1.151 Error Messages: incomplete exception handler definition

```
              - 'incomplete exception handler definition'
  -----------------------------------------
  You probably used EXCEPT without HANDLE, or the other way round
  see
              chapter 13
               on exception handling.
```

1.152 Chapter Two, Sections A through E

```
              Section A:
              Decimal (1)
              Section B:
              Hexadecimal ($1)
              Section C:
              Binary (%1)
              Section D:
              Float (1.0)
              Section E:
              Character (''a'')
```

1.153 E Language Reference: Chapter Sixteen, Section F

```
compiler buffer organization 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 some buffers, like the one for storing constants, 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' option.

Experiment with different types and sizes of example-sources in combination
with the '-b' option to see how this works in practice.
```

1.154 E Language Reference: Chapter Sixteen, Section G

```
a small history
---------------
E is not 'just another language'; it was carefully and gradually designed by
the author of the compiler because he was not too happy with existing
programming languages, and specifically not the sluggish-code-generating
and slow compilers that were written for them.  Amiga E had as primary
goal to be used as language for the author to program his amiga programs
in, and he has succeeded in doing so by far.  E was developed intensively
over the course of 1.5 years and was certainly not the first compiler
written by the author; some of you may remember the DEX compiler.

This one was slow and unpowerful and is hardly something that can be
compared to a compiler like Amiga E, but certainly gave the author some
useful experience to be able to make Amiga E what it is today.
DEX programmers will notice that it is very easy to convert their old
DEX sources to E, and continue developing with 10x the power at 20x the
speed.  A funny thing about DEX and E is that the development of the
two compilers did overlap; while DEX was done, E was halfway v1.6.
Because E was already much better back then, E libraries/examples and
code were transferred to DEX by popular demand, so the predecessor
inherited features from its successor.

The author also wrote numerous other compilers and interpreters, some
of which were never distributed in any way.

Amiga E is a product that will continue to be developed towards the ultimate
```

```
language / amiga development system:

- by implementing those missing parts in the language definition:

  * Object Orientedness
  * better float concept

- by making compiler specific enhancements:

  * possible 020/030/881 code generation
  * optimizing the compilation-process, thus possibly doubling the
    line/minute figures as in compiler.doc
  * enabling the user to compile own code to modules, and thus develop large
    applications in a more modular fashion

- by adding valuable elements to the distribution

  * an integrated editor ?
  * source-level debugger ?
  * CASE tools, for example

- by fixing bugs (what bugs!?!) 8*-)
```

1.155 E Language Reference: Chapter Seventeen

```
    +-----------------------------------------------+
    |             COMMAND-LINE OPTIONS              |
    +-----------------------------------------------+

These need to be written together, preceded by a "-":

-l  compiles with large code/data model.
-a  puts EC into assembler mode.
-n  suppresses warnings.
-w  puts wb to front (for scripts)
-b  shows buffer's memory usage information
-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.

example:  EC -l blabla

Compiles blabla.e with large model.

NOTE: in most standard cases you won't need to use any of these options.
```

1.156 E Reference Guide: Index

```
                Index
-----

This index did not exist in the original documentation (of course). It is
```

organized according to how one might want to find some information quickly. There may be a breach from the StyleGuide however, this may be a little more handy than searching through a rather cumbersome index. The "Sub Index" holds a more exhaustive list.

1.157 sub_index

Sub Index

You'll note that these are not alphabetically ordered, but ordered according to how they appeared in the original manual (by chapter). This again isn't particularly standard, but perhaps it's an improvement, as you'll have somewhat of an idea Where It Falls.

This particular index may be handy for Finding a particular value.

Format:

Strings ('bla')

Lists ([1,2,3]) and typed lists
Expressions:

Format

Precedence and grouping

Types of expressions

Function calls
Operators:

Math (+ - * /)

Comparison (= <> > < >= <=)

Logical and bitwise (AND OR)

Unary (SIZEOF ` ^ {} ++ -- -)

Triple (IF THEN ELSE)

Structure (.)

Array ([])

Float operator (|)

Assignments expressions (:=)

Sequencing (BUT)
Statements:

Format (;)

Statement labels and gotos (JUMP)

Assignment (:=)

Assembly mnemonics

Conditional statement (IF)

For-statement (FOR)

While-statement (WHILE)

Repeat-statement (REPEAT)

Loop-statement (LOOP)

Select-case-statement (SELECT)

Increase statement (INC/DEC)

```
syntax error

unknown keyword/const

':=' expected

unexpected characters in line

label expected

',' expected

variable expected

value does not fit in 32 bit

missing apostrophe/quote

incoherent program structure

illegal command-line option

division and multiplication 16bit only

superfluous items in expression/statement

procedure 'main' not available

double declaration of label

unsafe use of '*' or '/'

reading sourcefile didn't succeed

writing executable didn't succeed

no args

unknown/illegal addressing mode

unmatched parentheses

double declaration

unknown identifier

incorrect #of args or use of ()

unknown e/library function

illegal function call

unknown format code following '/'

/* not properly nested comment structure */
```

```
could not load binary
 '}'
expected

immediate value expected

incorrect size of value

no e code allowed in assembly modus

illegal/inappropriate type

']' expected

statement out of local/global scope

could not read module correctly

workspace full!

not enough memory while (re-)allocating

incorrect object definition

illegal use of/reference to object

incomplete if-then-else expression

unknown object identifier

double declaration of object identifier

reference(s) out of 32k range: switch to LARGE model

reference(s) out of 256 byte range

too sizy expression

incomplete exception handler definition
```

1.158 Using this documentation....

You'll note that the Browse gadgets don't work as one might expect. This document is set so you may browse the previous menu's sections at a time, scrolling back to the first section from the last, and last from the first. Hopefully, this scheme will not be distracting.

The Index is also probably non-standard: it has been arranged so you can find certain lists quickly and efficiently, without having to wade through a veritable TON of keywords and phrases. However, if you are looking for something relatively specific, and want to use AmigaGuide's Find command, try using the Index's "Sub Index", which lists all the various sections from each chapter in the original documentation (including ALL those error messages...).

1.159 Credits: Who gets blamed for what

```
Wouter van Oortmerssen - For the creation of a most exceptionally
       Amiga-like compilor, and for documenting his
       compilor.

Joseph E. Van Riper III- For changing Mr. van Oortmerssen's Reference
       documentation to AmigaGuide format, and mild editing
       of the same.

This document has been blessed by Mr. van Oortmerssen, and has his
permission to be released for everyone else's benefit.  If you should find
a bad reference in here, please contact jvanriper@uncavx.unca.edu.  If the
documentation itself is wrong, please contact Wouter@alf.let.uva.nl.
```