beginner

COLLABORATORS			
	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 22, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 Format and Layout

```
Format and Layout
*****************
```

In this chapter we'll look at the rules which govern the format and layout of E code. In the previous Part we saw examples of E code that were quite nicely indented and the structure of the program was easily visible. This was just a convention and the E language does not constrain you to write code in this way. However, there are certain rules that must be followed. (This chapter refers to some concepts and parts of the E language which were not covered in Part One. Don't let this put you off--those things will be dealt with in later chapters, and it's maybe a good idea to read this chapter again when they have been.)

```
  Identifiers
  Statements
  Spacing and Separators
  Comments
```

1.2 Identifiers

```
Identifiers
===========
```

An identifier is a word which the compiler must interpret rather than treating literally. For instance, a variable is an identifier, as is a keyword (e.g., IF), but anything in a string is not (e.g., fred in 'fred and wilma' is not an identifier). Identifiers can be made up of upper- or lower-case letters, numbers and underscores (the _ character). There are only two constraints:

1. The first character cannot be a number (this would cause confusion with numeric constants).

2. The case of the first few characters of identifiers is significant.

For keywords (e.g., ENDPROC), constants (e.g., TRUE) and assembly mnemonics (e.g., MOVE.L) the first two characters must both be uppercase. For E built-in or Amiga system procedures/functions the first character must be uppercase and the second must be lowercase. For all other identifiers (i.e., local, global and procedure parameter variables, object names and element names, procedure names and code labels) the first character must be lowercase.

Apart from these constraints you are free to write identifiers how you like, although it's arguably more tasteful to use all lowercase for variables and all uppercase for keywords and constants.

1.3 Statements

Statements
==========

A statement is normally a single instruction to the computer, and each statement normally occupies a single line. If you think of a procedure as a paragraph then a statement is a sentence. Using the same analogy, variables, expressions and keywords are the words which make up the sentence.

So far in our examples we have met only two kinds of statement: the single line statement and the multi-line statement. The assignments we have seen were single line statements, and the vertical form of the IF block is a multi-line statement. The horizontal form of the IF block was actually the single line statement form of the IF block. Notice that statements can be built up from other statements, as is the case for IF blocks. The code parts between the IF, ELSEIF, ELSE and ENDIF lines are sequences of statements.

Single line statements can often be very short, and you may be able to fit several of them onto an single line without the line getting too long. To do this in E you use a semi-colon (the ; character) to separate each statement on the line. For example, the following code fragments are equivalent:

```
fred(y,z)
y:=x
x:=z+1

fred(y,z); y:=x; x:=z+1
```

On the other hand you may want to split a long statement over several lines. This is a bit more tricky because the compiler needs to see that you haven't finished the statement when it gets to the end of a line. Therefore you can only break a statement at certain places. The most common place is after a comma that is part of the statement (like in a procedure call with more than one parameter), but you can also split a line after binary operators and anywhere between opening and closing brackets. The following examples are rather silly but show some allowable

line breaking places.

```
        fred(a, b, c,
             d, e, f)   /* After a comma */

        x:=x+
           y+
           z            /* After a binary operator */

        x:=(1+2
            +3)         /* Between open...close brackets */

        list:= [ 1,2,
                 [3,4]
               ]        /* Between open...close brackets */
```

The simple rule is this: if a complete line can be interpreted as a statement then it will be, otherwise it will be interpreted as part of a statement which continues on the following lines.

Strings may also get a bit long. You can split them over several lines by breaking them into several separate strings and using + between them. If a line ends with a + and the previous thing on the line was a string then the E compiler takes the next string to be a continuation. The following calls to WriteF print the same thing:

```
       WriteF('This long string can be broken over several lines.\n')

       WriteF('This long string ' +
              'can be broken over several lines.\n')

       WriteF('This long' +
              ' string can be ' +
              'broken over several ' +
              'lines.\n')
```

1.4 Spacing and Separators

```
Spacing and Separators
======================
```

The examples we've seen so far used a rigid indentation convention which was intended to illuminate the structure of the program. This was just a convention, and the E language places no constraints on the amount of whitespace (spaces, tabs and linefeeds) you place between statements. However, within statements you must supply enough spacing to make the statement readable. This generally means that you must put whitespace between adjacent identifiers which start or end with a letter, number or underscore (so that the compiler does not think it's one big identifier!). So, in practice, you should put a space after a keyword if it might run into a variable or procedure name. Most other times (like in expressions) identifiers are separated by non-identifier characters (a comma, parenthesis or other symbol).

1.5 Comments

```
Comments
========

   A comment is something that the E compiler ignores and is only there
to help the reader.  Remember that one day in the future you may be the
reader, and it may be quite hard to decipher your own code without a few
decent comments!  Comments are therefore pretty important.

   You can write comments anywhere you can write whitespace that isn't
part of a string.  There are two kinds of comment: one uses /* to mark the
start of the comment text and */ to mark the end, and the other uses -> to
mark the start, with the comment text continuing to the end of the line.
You must be careful not to write /*, */ or -> as part of the comment text,
unless part of a nested comment.  In practice a comment is best put on a
line by itself or after the end of the code on a line.

        /* This line is a comment */
        x:=1  /* This line contains an assignment then a comment */
     /* y:=2  /* This whole line is a comment with a nested comment */*/

        x:=1  -> Assignment then a comment
     -> y:=2  /* A nested comment comment */
```