[To view this manual with your Web browser, double-click on html\refman.htm]

Euphoria Programming Language

version 2.4

Reference Manual

TABLE OF CONTENTS

Part I - Core Language

1. Introduction

Euphoria is a new programming language with the following advantages over conventional languages:

* a remarkably simple, flexible, powerful language definition that is easy to learn and use.

* dynamic storage allocation. Variables grow or shrink without the programmer having to worry about allocating and freeing chunks of memory. Objects of any size can be assigned to an element of a Euphoria sequence (array).

* a high-performance, state-of-the-art interpreter that's 30 times faster than conventional interpreters such as Perl and Python.

* an optimizing Euphoria To C Translator, that can boost your speed even further, often by a factor of 2x to 5x versus the already-fast interpreter.

* lightning-fast pre-compilation. Your program source is checked for syntax and converted into an efficient internal form at over 35,000 lines per second on a lowly Pentium-150. There's no need to mess around with extra "byte-code" files.

* extensive run-time checking for: out-of-bounds subscripts, uninitialized variables, bad parameter values for library routines, illegal value assigned to a variable and many more. There are no mysterious machine exceptions -- you will always get a full English description of any problem that occurs with your program at run-time, along with a call-stack trace-back and a dump of all of your variable values. Programs can be debugged quickly, easily and more thoroughly.

* features of the underlying hardware are completely hidden. Programs are not aware of word-lengths, underlying bit-level representation of values, byte-order etc.

* a full-screen source debugger and an execution profiler are included, along with a full-screen, multi-file editor. On a color monitor, the editor displays Euphoria programs in multiple colors, to highlight comments, reserved words, built-in functions, strings, and level of nesting of brackets. It optionally performs auto-completion of statements, saving you typing effort and reducing syntax errors. This

editor is written in Euphoria, and the source code is provided to you without restrictions. You are free to modify it, add features, and redistribute it as you wish.

* Euphoria programs run under Linux, FreeBSD, 32-bit Windows, and any DOS environment, and are not subject to any 640K memory limitations. You can create programs that use the full multi-megabyte memory of your computer, and a swap file is automatically used when a program needs more memory than exists on your machine.

* You can make a single, stand-alone .exe file from your program.

* Euphoria routines are naturally generic. The example program below shows a single routine that will sort any type of data -- integers, floating-point numbers, strings etc. Euphoria is not an "object-oriented" language, yet it achieves many of the benefits of these languages in a much simpler way.

1.1 Example Program

The following is an example of a complete Euphoria program.

~~~~~~

```
sequence list, sorted_list

function merge_sort(sequence x)
-- put x into ascending order using a recursive merge sort
    integer n, mid
    sequence merged, a, b

    n = length(x)
    if n = 0 or n = 1 then
        return x  -- trivial case
    end if

    mid = floor(n/2)
    a = merge_sort(x[1..mid])       -- sort first half of x
    b = merge_sort(x[mid+1..n])     -- sort second half of x

    -- merge the two sorted halves into one
    merged = {}
    while length(a) > 0 and length(b) > 0 do
        if compare(a[1], b[1]) < 0 then
            merged = append(merged, a[1])
            a = a[2..length(a)]
        else
            merged = append(merged, b[1])
            b = b[2..length(b)]
        end if
    end while
    return merged & a & b  -- merged data plus leftovers
end function
```
~~~~~~

```
procedure print_sorted_list()
-- generate sorted_list from list
    list = {9, 10, 3, 1, 4, 5, 8, 7, 6, 2}
    sorted_list = merge_sort(list)
    ? sorted_list
end procedure

print_sorted_list()     -- this command starts the program
```

~~~~~~

The above example contains 4 separate commands that are processed in order. The first declares two variables: list and sorted_list to be sequences (flexible arrays). The second defines a function merge_sort(). The third defines a procedure print_sorted_list(). The final command calls procedure print_sorted_list().

The output from the program will be:

```
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.
```

merge_sort() will just as easily sort {1.5, -9, 1e6, 100} or {"oranges", "apples", "bananas"} .

This example is stored as euphoria\tutorial\example.ex. This is not the fastest way to sort in Euphoria. Go to the euphoria\demo directory and type "ex allsorts" to see timings on several different sorting algorithms for increasing numbers of objects. For a quick tutorial example of Euphoria programming see euphoria\demo\bench\filesort.ex.

1.2 Installation

To install Euphoria on your machine, first read the file install.doc. Installation simply involves copying the euphoria files to your hard disk under a directory named "euphoria", and then modifying your autoexec.bat file so that euphoria\bin is on your search path, and the environment variable EUDIR is set to the euphoria directory.

When installed, the euphoria directory will look something like this:

```
\euphoria
       readme.doc
       readme.htm
       \bin
              ex.exe and exw.exe, or exu (Linux/FreeBSD), ed.bat, guru.bat,
              other utilities
       \include
              standard include files, e.g. graphics.e
       \doc
              refman.doc, library.doc, and several other plain-text
              documentation files
       \html
              HTML files corresponding to each of the .doc files in the doc
```
~~~~~~

```
              directory
       \tutorial
              small tutorial programs to help you learn Euphoria
       \demo
              generic demo programs that run on all platforms
              \dos32
                     DOS32-specific demo programs (optional)
              \win32
                     WIN32-specific demo programs (optional)
              \linux
                     Linux/FreeBSD-specific demo programs (optional)
              \langwar
                     language war game (pixel-graphics version for DOS, or
                     text version for Linux/FreeBSD)
              \bench
                     benchmark programs
       \register
              information on ordering the Complete Edition
```

The Linux subdirectory is not included in the DOS/Windows distribution, and the dos32 and win32 subdirectories are not included in the Linux/FreeBSD distribution. In this manual, directory names are shown using backslash (\). Linux/FreeBSD users should substitute forward slash (/).

1.3 Running a Program

Euphoria programs are executed by typing "ex", "exw" or "exu" followed by the name of the main Euphoria file. You can type additional words (known as arguments) on this line, known as the command-line. Your program can call the built-in function command_line() to read the command-line. The DOS32 version of the Euphoria interpreter is called ex.exe. The WIN32 version is called exw.exe. The Linux/FreeBSD version is called exu. By convention, main Euphoria files have an extension of ".ex", ".exw" or ".exu". Other Euphoria files, that are meant to be included in a larger program, end in ".e" or sometimes ".ew" or ".eu". To save typing, you can leave off the ".ex", and the ex command will supply it for you automatically. exw.exe will supply ".exw", and exu will supply ".exu". If the file can't be found in the current directory, your PATH will be searched. You can redirect standard input and standard output when you run a Euphoria program, for example:

```
       ex filesort.ex < raw.txt > sorted.txt
```

or simply,

```
       ex filesort < raw.txt > sorted.txt
```

Unlike many other compilers and interpreters, there are no special command-line options for ex, exw or exu. Only the name of your Euphoria file is expected, and if you don't supply it, you will be prompted for it.

For frequently-used programs under DOS/Windows you might want to make a small .bat (batch) file, perhaps called myprog.bat, containing two statements like:

```
       @echo off
```

```
        ex myprog.ex %1 %2 %3
```

The first statement turns off echoing of commands to the screen. The second runs ex myprog.ex with up to 3 command-line arguments. See command_line() for an example of how to read these arguments. If your program takes more arguments, you should add %4 %5 etc. Having a .bat file will save you the minor inconvenience of typing "ex" (or "exw") all the time, i.e. you can just type:

```
        myprog
```

instead of:

```
        ex myprog
```

Unfortunately DOS will not allow redirection of standard input and output when you use a .bat file

Under Linux/FreeBSD, you can type the path to the Euphoria interpreter on the first line of your main file, e.g. if your program is called foo.exu:

```
        #!/home/rob/euphoria/bin/exu

        procedure foo()
            ? 2+2
        end procedure

        foo()
```

Then if you make your file executable:

```
        chmod +x foo.exu
```

You can just type:

```
        foo.exu
```

to run your program. You could even shorten the name to simply "foo". Euphoria ignores the first line when it starts with #!. Be careful though that your first line ends with the Linux/FreeBSD-style \n, and not the DOS/Windows-style \r\n, or the Linux/FreeBSD shell might get confused.

You can also run bind.bat (DOS32), or bindw.bat (WIN32) or bindu (Linux/FreeBSD) to combine your Euphoria program with ex.exe, exw.exe or exu, to make a stand-alone executable file (.exe file on DOS/Windows). With a stand-alone .exe file you *can* redirect standard input and output. Binding is discussed further in 1.5 Distributing a Program.

Either exu or ex.exe and exw.exe are in the euphoria\bin directory which must be on your search path. The environment variable EUDIR should be set to the main Euphoria directory, e.g. c:\euphoria.

1.3.1 Running under Windows

You can run Euphoria programs directly from the Windows environment, or from a

DOS shell that you have opened from Windows. By "associating" .ex files with ex.exe, and .exw files with exw.exe you can simply double-click on a .ex or .exw file to run it. Under Windows you would define a new file type for .ex, by clicking on My Computer / view / options / file types. It is possible to have several Euphoria programs active in different windows. If you turn your program into a .exe file, you can simply double-click on it to run it.

1.3.2 Use of a Swap File

If you run a Euphoria program under Linux/FreeBSD or Windows (or in a DOS shell under Windows), and the program runs out of physical memory, it will start using "virtual memory". The operating system provides this virtual memory automatically by swapping out the least-recently-used code and data to a system swap file. To change the size of the Windows swap file, click on Control Panel / 386 Enhanced / "virtual memory...". Under OS/2 you can adjust the "DPMI_MEMORY_LIMIT" by clicking the Virtual DOS machine icon / "DOS Settings" to allocate more extended memory for your program.

Under pure DOS, outside of Windows, there is no system swap file so the DOS-extender built in to ex.exe (DOS32) will create one for possible use by your program. See platform.doc.

1.4 Editing a Program

You can use any text editor to edit a Euphoria program. However, Euphoria comes with its own special editor that is written entirely in Euphoria. Type: ed followed by the complete name of the file you wish to edit (the .ex/.exw/.exu extension is not assumed). You can use this editor to edit any kind of text file. When you edit a Euphoria file, some extra features such as color syntax highlighting and auto-completion of certain statements, are available to make your job easier.

Whenever you run a Euphoria program and get an error message, during compilation or execution, you can simply type ed with no file name and you will be automatically positioned in the file containing the error, at the correct line and column, and with the error message displayed at the top of the screen.

Under Windows you can associate ed.bat with various kinds of text files that you want to edit. Color syntax highlighting is provided for .ex, .exw, .exu, .e, .ew, .eu, and .pro (profile) files.

Most keys that you type are inserted into the file at the cursor position. Hit the Esc key once to get a menu bar of special commands. The arrow keys, and the Insert/Delete/Home/End/PageUp/PageDown keys are also active. Under Linux/FreeBSD some keys may not be available, and alternate keys are provided. See the file euphoria\doc\ed.doc (euphoria\html\ed.htm) for a complete description of the editing commands. Esc h (help) will let you view ed.doc from your editing session.

If you need to understand or modify any detail of the editor's operation, you can edit the file ed.ex in euphoria\bin (be sure to make a backup copy so you don't lose your ability to edit). If the name ed conflicts with some other command on your system, simply rename the file euphoria\bin\ed.bat to something else. Because this editor is written in Euphoria, it is remarkably concise and easy to understand. The same functionality implemented in a language like C, would take far more lines of code.

ed is a simple text-mode editor that runs on DOS, Linux, FreeBSD and the Windows console. See also David Cuny's excellent ee.ex editor for DOS and Linux/FreeBSD. You can download ee.ex from the Euphoria Web site. There are also some Windows editors oriented to Euphoria. These are also on the Web site.

1.5 Distributing a Program

Euphoria provides you with 4 distinct ways of distributing a program.

In the first method you simply ship your users the Public Domain ex.exe or exw.exe or exu file, along with your main Euphoria .ex, .exw, or .exu file and any .e include files that are needed (including any of the standard ones from euphoria\include). If the Euphoria source files are placed together in one directory and ex.exe, exw.exe or exu is placed in the same directory or somewhere on the search path, then your user can run your program by typing "ex" ("exw") or ("exu") followed by the path of your main .ex, .exw, or .exu file. You might also provide a small .bat file so people won't actually have to type "ex" ("exw"). This method assumes that you are willing to share your Euphoria source code with your users.

The Complete Edition gives you two more methods of distribution. You can "shroud" your program, or you can "bind" your program. Shrouding combines all of the .e files that your program needs, along with your main file to create a single .ex, .exw, or .exu file. You can either encrypt/compact your program to conceal it and make it tamper-proof, or you can leave it readable to allow better diagnostics from your users. Binding combines your shrouded program with ex.exe, exw.exe, or exu to create a single, stand-alone executable (.exe) file. For example, if your program is called "myprog.ex" you can create "myprog.exe" which will run identically. For more information about shrouding and binding, see bind.doc.

Finally, with the Euphoria To C Translator, you can "translate" your Euphoria program into C and then compile it with a C compiler to get an executable (.exe) file. The Translator is a separate download available on the RDS Web site.

1.5.1 Licensing

You have complete royalty-free rights to distribute any Euphoria programs that you develop. You are free to distribute the Public Domain Edition ex.exe, exw.exe and exu files so anyone can run your program. With the Complete Edition, you can shroud or bind your programs and distribute the resulting files royalty-free.

You may incorporate any Euphoria source files from this package into your program, either "as is" or with your modifications. (You will probably need at least a few of the standard include files in any large program).

We would appreciate it if you told people that your program was developed using Euphoria, and gave them the address: http://www.RapidEuphoria.com of our Web page, but we do not require any such acknowledgment.

The only Interpreter-related files that you may *not* distribute are the ex.exe, exw.exe, bind.ex, bind.bat, bindw.bat, and shroud.bat files that come with the Complete Edition Interpreter for WIN32 + DOS32, and exu, bind.ex, bindu and shroud that come with the Complete Edition Interpreter for Linux/FreeBSD. The sample icon file, euphoria.ico, that's included with the WIN32 + DOS32 Complete Edition, may be distributed with or without your changes.

The Euphoria Interpreter is written in ANSI C, and can be compiled with many different C compilers. The source is available for purchase. Before buying it, please read the Interpreter Source License.

Some additional legal restrictions might apply when you use the Euphoria To C Translator.

2. Language Definition

2.1 Objects

2.1.1 Atoms and Sequences

All data objects in Euphoria are either atoms or sequences. An atom is a single numeric value. A sequence is a collection of numeric values.

The objects contained in a sequence can be an arbitrary mix of atoms or sequences. A sequence is represented by a list of objects in brace brackets, separated by commas. Atoms can have any integer or double-precision floating point value. They can range from approximately -1e300 (minus one times 10 to the power 300) to +1e300 with 15 decimal digits of accuracy. Here are some Euphoria objects:

```
       -- examples of atoms:
       0
       1000
       98.6
       -1e6

       -- examples of sequences:
       {2, 3, 5, 7, 11, 13, 17, 19}
       {1, 2, {3, 3, 3}, 4, {5, {6}}}
       {{"jon", "smith"}, 52389, 97.25}
       {}                        -- the 0-element sequence
```

Numbers can also be entered in hexadecimal. For example:

```
       #FE             -- 254
       #A000           -- 40960
       #FFFF00008      -- 68718428168
       -#10            -- -16
```

Only the capital letters A, B, C, D, E, F are allowed in hex numbers.

Sequences can be nested to any depth, i.e. you can have sequences within sequences within sequences and so on to any depth (until you run out of memory). Brace brackets are used to construct sequences out of a list of expressions. These expressions can be constant or evaluated at run-time. e.g.

```
       {x+6, 9, y*w+2, sin(0.5)}
```

The "Hierarchical Objects" part of the Euphoria acronym comes from the hierarchical nature of nested sequences. This should not be confused with the class hierarchies of certain object-oriented languages.

Why do we call them "atoms"? Why not just "numbers"? Well, an atom *is* just a number, but we wanted to have a distinctive term that emphasizes that they are indivisible. Of course in the world of physics, atoms were split into smaller parts many years ago, but in Euphoria you can't split them. They are the basic building blocks of all the data that a Euphoria program can manipulate. With this analogy, sequences might be thought of as "molecules", made from atoms and other molecules. A better analogy would be that sequences are like directories, and atoms are like files. Just as a directory on your computer can contain both files and other directories, a sequence can contain both atoms and other sequences (and *those* sequences can contain atoms and sequences and so on).

As you will soon discover, sequences make Euphoria very simple *and* very powerful. Understanding atoms and sequences is the key to understanding Euphoria.

Performance Note:

Does this mean that all atoms are stored in memory as 8-byte floating-point numbers? No. The Euphoria interpreter usually stores integer-valued atoms as machine integers (4 bytes) to save space and improve execution speed. When fractional results occur or numbers get too big, conversion to floating-point happens automatically.

2.1.2 Character Strings and Individual Characters

A character string is just a sequence of characters. It may be entered using quotes e.g.

```
       "ABCDEFG"
```

Character strings may be manipulated and operated upon just like any other sequences. For example the above string is entirely equivalent to the sequence:

```
       {65, 66, 67, 68, 69, 70, 71}
```

which contains the corresponding ASCII codes. The Euphoria compiler will immediately convert "ABCDEFG" to the above sequence of numbers. In a sense, there are no "strings" in Euphoria, only sequences of numbers. A quoted string is really just a convenient notation that saves you from having to type in all the ASCII codes.

It follows that "" is equivalent to {}. Both represent the sequence of length-0, also known as the empty sequence. As a matter of programming style, it is natural to use "" to suggest a length-0 sequence of characters, and {} to suggest some other kind of sequence.

An individual character is an atom. It must be entered using single quotes. There is a difference between an individual character (which is an atom), and a character string of length-1 (which is a sequence). e.g.

```
       'B'   -- equivalent to the atom 66 - the ASCII code for B
       "B"   -- equivalent to the sequence {66}
```

Again, 'B' is just a notation that is equivalent to typing 66. There aren't really any "characters" in Euphoria, just numbers (atoms).

Keep in mind that an atom is *not* equivalent to a one-element sequence containing the same value, although there are a few built-in routines that choose to treat them similarly.

Special characters may be entered using a back-slash:

```
       \n        newline
       \r        carriage return
       \t        tab
       \\        backslash
       \"        double quote
       \'        single quote
```

For example, "Hello, World!\n", or '\\'. The Euphoria editor displays character strings in green.

2.1.3 Comments

Comments are started by two dashes and extend to the end of the current line. e.g.

```
       -- this is a comment
```

Comments are ignored by the compiler and have no effect on execution speed. The editor displays comments in red.

On the first line (only) of your program, you can use a special comment beginning with #!, e.g.

```
       #!/home/rob/euphoria/bin/exu
```

This informs the Linux shell that your file should be executed by the Euphoria interpreter, and gives the full path to the interpreter. If you make your file executable, you can run it, just by typing its name, and without the need to type "exu". On DOS and Windows this line is treated as a comment.

2.2 Expressions

Like other programming languages, Euphoria lets you calculate results by forming expressions. However, in Euphoria you can perform calculations on entire sequences of data with one expression, where in most other languages you would have to construct a loop. In Euphoria you can handle a sequence much as you would a single number. It can be copied, passed to a subroutine, or calculated upon as a unit. For example,

```
       {1,2,3} + 5
```

is an expression that adds the sequence {1,2,3} and the atom 5 to get the resulting sequence {6,7,8}.

We will see more examples later.

2.2.1 Relational Operators

The relational operators < > <= >= = != each produce a 1 (true) or a 0 (false) result.

```
       8.8 < 8.7   -- 8.8 less than 8.7 (false)
       -4.4 > -4.3 -- -4.4 greater than -4.3 (false)
       8 <= 7      -- 8 less than or equal to 7 (false)
       4 >= 4      -- 4 greater than or equal to 4 (true)
       1 = 10      -- 1 equal to 10 (false)
       8.7 != 8.8  -- 8.7 not equal to 8.8 (true)
```

As we will soon see you can also apply these operators to sequences.

2.2.2 Logical Operators

The logical operators 'and', 'or', 'xor', and 'not' are used to determine the "truth" of an expression. e.g.

```
       1 and 1     -- 1 (true)
       1 and 0     -- 0 (false)
       0 and 1     -- 0 (false)
       0 and 0     -- 0 (false)

       1 or  1     -- 1 (true)
       1 or  0     -- 1 (true)
       0 or  1     -- 1 (true)
       0 or  0     -- 0 (false)

       1 xor 1     -- 0 (false)
       1 xor 0     -- 1 (true)
       0 xor 1     -- 1 (true)
       0 xor 0     -- 0 (false)

       not 1       -- 0 (false)
       not 0       -- 1 (true)
```

You can also apply these operators to numbers other than 1 or 0. The rule is: zero means false and non-zero means true. So for instance:

```
       5 and -4    -- 1 (true)
       not 6       -- 0 (false)
```

These operators can also be applied to sequences. See below.

In some cases "short-circuit" evaluation will be used for expressions containing "and" or "or".

2.2.3 Arithmetic Operators

The usual arithmetic operators are available: add, subtract, multiply, divide, unary minus, unary plus.

```
       3.5 + 3  -- 6.5
       3 - 5    -- -2
       6 * 2    -- 12
       7 / 2    -- 3.5
       -8.1     -- -8.1
       +8       -- +8
```

Computing a result that is too big (i.e. outside of -1e300 to +1e300) will result in one of the special atoms +infinity or -infinity. These appear as "inf" or "-inf" when you print them out. It is also possible to generate "nan" or "-nan". "nan" means "not a number", i.e. an undefined value (such as inf divided by inf). These values are defined in the IEEE floating-point standard. If you see one of these special values in your output, it usually indicates an error in your program logic, although generating inf as an intermediate result may be acceptable in some cases. For instance, 1/inf is 0, which may be the "right" answer for your algorithm.

Division by zero, as well as bad arguments to math library routines, e.g. square root of a negative number, log of a non-positive number etc. cause an immediate error message and your program is aborted.

The only reason that you might use unary plus is to emphasize to the reader of your program that a number is positive. The interpreter does not actually calculate anything for this.

2.2.4 Operations on Sequences

All of the relational, logical and arithmetic operators described above, as well as the math routines described in Part II - Library Routines, can be applied to sequences as well as to single numbers (atoms).

When applied to a sequence, a unary (one operand) operator is actually applied to each element in the sequence to yield a sequence of results of the same length. If one of these elements is itself a sequence then the same rule is applied again recursively. e.g.

```
       x = -{1, 2, 3, {4, 5}}   -- x is {-1, -2, -3, {-4, -5}}
```

If a binary (two-operand) operator has operands which are both sequences then the two sequences must be of the same length. The binary operation is then applied to corresponding elements taken from the two sequences to get a sequence of results. e.g.

```
       x = {5, 6, 7, 8} + {10, 10, 20, 100}
       -- x is {15, 16, 27, 108}
```

If a binary operator has one operand which is a sequence while the other is a single number (atom) then the single number is effectively repeated to form a sequence of equal length to the sequence operand. The rules for operating on two sequences then apply. Some examples:

```
       y = {4, 5, 6}

       w = 5 * y              -- w is {20, 25, 30}

       x = {1, 2, 3}

       z = x + y              -- z is {5, 7, 9}

       z = x < y              -- z is {1, 1, 1}

       w = {{1, 2}, {3, 4}, {5}}

       w = w * y              -- w is {{4, 8}, {15, 20}, {30}}

       w = {1, 0, 0, 1} and {1, 1, 1, 0}    -- {1, 0, 0, 0}

       w = not {1, 5, -2, 0, 0}     -- w is {0, 0, 0, 1, 1}

       w = {1, 2, 3} = {1, 2, 4}    -- w is {1, 1, 0}
       -- note that the first '=' is assignment, and the
       -- second '=' is a relational operator that tests
       -- equality
```

Note: When you wish to compare two strings (or other sequences), you should
-----
not (as in some other languages) use the '=' operator:

```
      if "APPLE" = "ORANGE" then  -- ERROR!
```

'=' is treated as an operator, just like '+', '*' etc., so it is applied to corresponding sequence elements, and the sequences must be the same length. When they are equal length, the result is a sequence of 1's an 0's. When they are not equal length, the result is an error. Either way you'll get an error, since an if-condition must be an atom, not a sequence. Instead you should use the equal() built-in routine:

```
      if equal("APPLE", "ORANGE"} then  -- CORRECT
```

In general, you can do relational comparisons using the compare() built-in routine:

```
      if compare("APPLE", "ORANGE"} = 0 then  -- CORRECT
```

You can use compare() for other comparisons as well:

```
      if compare("APPLE", "ORANGE"} < 0 then  -- CORRECT
          -- enter here if "APPLE" is less than "ORANGE" (TRUE)
```

2.2.5 Subscripting of Sequences

A single element of a sequence may be selected by giving the element number in square brackets. Element numbers start at 1. Non-integer subscripts are rounded down to an integer.

For example, if x contains {5, 7.2, 9, 0.5, 13} then x[2] is 7.2. Suppose we assign something different to x[2]:

```
       x[2] = {11,22,33}
```

Then x becomes: {5, {11,22,33}, 9, 0.5, 13}. Now if we ask for x[2] we get {11,22,33} and if we ask for x[2][3] we get the atom 33. If you try to subscript with a number that is outside of the range 1 to the number of elements, you will get a subscript error. For example x[0], x[-99] or x[6] will cause errors. So will x[1][3] since x[1] is not a sequence. There is no limit to the number of subscripts that may follow a variable, but the variable must contain sequences that are nested deeply enough. The two dimensional array, common in other languages, can be easily represented with a sequence of sequences:

```
       x = {
            {5, 6, 7, 8, 9},      -- x[1]
            {1, 2, 3, 4, 5},      -- x[2]
            {0, 1, 0, 1, 0}       -- x[3]
           }
```

where we have written the numbers in a way that makes the structure clearer. An expression of the form x[i][j] can be used to access any element.

The two dimensions are not symmetric however, since an entire "row" can be selected with x[i], but there is no simple expression to select an entire column. Other logical structures, such as n-dimensional arrays, arrays of strings, structures, arrays of structures etc. can also be handled easily and flexibly:

3-D array:

```
       y = {
            {{1,1}, {3,3}, {5,5}},
            {{0,0}, {0,1}, {9,1}},
            {{-1,9},{1,1}, {2,2}}
           }


    y[2][3][1] is 9
```

Array of strings:

```
       s = {"Hello", "World", "Euphoria", "", "Last One"}


    s[3] is "Euphoria"
    s[3][1] is 'E'
```

A Structure:

```
       employee = {
                   {"John","Smith"},
                   45000,
                   27,
                   185.5
                  }
```

To access "fields" or elements within a structure it is good programming style to make up a set of constants that name the various fields. This will make your program easier to read. For the example above you might have:

```
       constant NAME = 1
       constant FIRST_NAME = 1, LAST_NAME = 2

       constant SALARY = 2
       constant AGE = 3
       constant WEIGHT = 4
```

You could then access the person's name with employee[NAME], or if you wanted the last name you could say employee[NAME][LAST_NAME].

Array of structures:

```
       employees = {
                    {{"John","Smith"}, 45000, 27, 185.5},   -- a[1]
                    {{"Bill","Jones"}, 57000, 48, 177.2},   -- a[2]

                    -- .... etc.
                   }
```

    employees[2][SALARY] would be 57000.

Euphoria data structures are almost infinitely flexible. Arrays in other languages are constrained to have a fixed number of elements, and those elements must all be of the same type. Euphoria eliminates both of those restrictions. You can easily add a new structure to the employee sequence above, or store an unusually long name in the NAME field and Euphoria will take care of it for you. If you wish, you can store a variety of different employee "structures", with different sizes, all in one sequence.

Not only can a Euphoria program easily represent all conventional data structures but you can create very useful, flexible structures that would be extremely hard to declare in a conventional language. See 2.3 Euphoria versus Conventional Languages.

Note that expressions in general may not be subscripted, just variables. For example: {5+2,6-1,7*8,8+1}[3] is *not* supported, nor is something like: date()[MONTH]. You have to assign the sequence returned by date() to a variable, then subscript the variable to get the month.

2.2.6 Slicing of Sequences

A sequence of consecutive elements may be selected by giving the starting and ending element numbers. For example if x is {1, 1, 2, 2, 2, 1, 1, 1} then x[3..5] is the sequence {2, 2, 2}. x[3..3] is the sequence {2}. x[3..2] is also allowed. It evaluates to the length-0 sequence {}. If y has the value: {"fred", "george", "mary"} then y[1..2] is {"fred", "george"}.

We can also use slices for overwriting portions of variables. After x[3..5] = {9, 9, 9} x would be {1, 1, 9, 9, 9, 1, 1, 1}. We could also have said x[3..5] = 9 with the same effect. Suppose y is {0, "Euphoria", 1, 1}. Then y[2][1..4] is "Euph". If we say y[2][1..4]="ABCD" then y will become {0, "ABCDoria", 1, 1}.

In general, a variable name can be followed by 0 or more subscripts, followed in turn by 0 or 1 slices. Only variables may be subscripted or sliced, not expressions.

We need to be a bit more precise in defining the rules for empty slices. Consider a slice s[i..j] where s is of length n. A slice from i to j, where j = i-1 and i >= 1 produces the empty sequence, even if i = n+1. Thus 1..0 and n+1..n and everything in between are legal (empty) slices. Empty slices are quite useful in many algorithms. A slice from i to j where j < i - 1 is illegal , i.e. "reverse" slices such as s[5..3] are not allowed.

2.2.7 Concatenation of Sequences and Atoms - The '&' Operator

Any two objects may be concatenated using the & operator. The result is a sequence with a length equal to the sum of the lengths of the concatenated objects (where atoms are considered here to have length 1). e.g.

```
        {1, 2, 3} & 4              -- {1, 2, 3, 4}

        4 & 5                      -- {4, 5}

        {{1, 1}, 2, 3} & {4, 5}    -- {{1, 1}, 2, 3, 4, 5}

        x = {}
        y = {1, 2}
        y = y & x                  -- y is still {1, 2}
```

You can delete element i of any sequence s by concatenating the parts of the sequence before and after i:

```
     s = s[1..i-1] & s[i+1..length(s)]
```

This works even when i is 1 or length(s), since s[1..0] is a legal empty slice, and so is s[length(s)+1..length(s)].

2.2.8 Sequence-Formation

Finally, sequence-formation, using braces and commas:

```
        {a, b, c, ... }
```

is also an operator. It takes n operands, where n is 0 or more, and makes an n-element sequence from their values. e.g.

```
        x = {apple, orange*2, {1,2,3}, 99/4+foobar}
```

The sequence-formation operator is listed at the bottom of the precedence chart.

2.2.9 Other Operations on Sequences

Some other important operations that you can perform on sequences have English names, rather than special characters. These operations are built-in to ex.exe/exw.exe/exu , so they'll always be there, and so they'll be fast. They are described in detail in Part II - Library Routines, but are important enough to Euphoria programming that we should mention them here before proceeding. You call these operations as if they were subroutines, although they are actually implemented much more efficiently than that.

length(s)

length() tells you the length of a sequence s. This is the number of elements in s. Some of these elements may be sequences that contain elements of their own, but length just gives you the "top-level" count. You'll get an error if you ask for the length of an atom. e.g.

```
        length({5,6,7})             -- 3
        length({1, {5,5,5}, 2, 3})  -- 4 (not 6!)
        length({})                  -- 0
        length(5)                   -- error!
```

repeat(item, count)
----

repeat() makes a sequence that consists of an item repeated count times. e.g.

```
        repeat(0, 100)         -- {0,0,0,...,0}   i.e. 100 zeros
        repeat("Hello", 3)     -- {"Hello", "Hello", "Hello"}
        repeat(99,0)           -- {}
```

The item to be repeated can be any atom or sequence.

append(s, item) / prepend(s, item)
----

append() creates a new sequence by adding an item to the end of a sequence s.
prepend() creates a new sequence by adding an element to the beginning of a sequence s. e.g.

```
        append({1,2,3}, 4)         -- {1,2,3,4}
        prepend({1,2,3}, 4)        -- {4,1,2,3}

        append({1,2,3}, {5,5,5})   -- {1,2,3,{5,5,5}}
        prepend({}, 9)             -- {9}
        append({}, 9)              -- {9}
```

The length of the new sequence is always 1 greater than the length of the original sequence. The item to be added to the sequence can be any atom or sequence.

These two built-in functions, append() and prepend(), have some similarities to the concatenate operator, &, but there are clear differences. e.g.

```
        -- appending a sequence is different
        append({1,2,3}, {5,5,5})   -- {1,2,3,{5,5,5}}
        {1,2,3} & {5,5,5}          -- {1,2,3,5,5,5}

        -- appending an atom is the same
        append({1,2,3}, 5)         -- {1,2,3,5}
        {1,2,3} & 5                -- {1,2,3,5}
```

2.2.10 Precedence Chart

The precedence of operators in expressions is as follows:

```
        highest precedence:     function/type calls
```

```
                        unary-  unary+  not

                        *  /

                        +  -

                        &

                        <  >  <=  >=  =  !=

                        and  or  xor

        lowest precedence:      { , , , }
```

Thus 2+6*3 means 2+(6*3) rather than (2+6)*3. Operators on the same line above have equal precedence and are evaluated left to right.

The equals symbol '=' used in an assignment statement is not an operator, it's just part of the syntax of the language.

2.3 Euphoria versus Conventional Languages

By basing Euphoria on this one, simple, general, recursive data structure, a tremendous amount of the complexity normally found in programming languages has been avoided. The arrays, structures, unions, arrays of records, multidimensional arrays, etc. of other languages can all be easily represented in Euphoria with sequences. So can higher-level structures such as lists, stacks, queues, trees etc.

Furthermore, in Euphoria you can have sequences of mixed type; you can assign any object to an element of a sequence; and sequences easily grow or shrink in length without your having to worry about storage allocation issues. The exact layout of a data structure does not have to be declared in advance, and can change dynamically as required. It is easy to write generic code, where, for instance, you push or pop a mix of various kinds of data objects using a single stack. Making a flexible list that contains a variety of different kinds of data objects is trivial in Euphoria, but requires dozens of lines of ugly code in other languages.

Data structure manipulations are very efficient since the Euphoria interpreter will point to large data objects rather than copy them.

Programming in Euphoria is based entirely on creating and manipulating flexible, dynamic sequences of data. Sequences are *it* - there are no other data structures to learn. You operate in a simple, safe, elastic world of *values*, that is far removed from the rigid, tedious, dangerous world of bits, bytes, pointers and machine crashes.

Unlike other languages such as LISP and Smalltalk, Euphoria's "garbage collection" of unused storage is a continuous process that never causes random delays in execution of a program, and does not pre-allocate huge regions of memory.

The language definitions of conventional languages such as C, C++, Ada, etc.

are very complex. Most programmers become fluent in only a subset of the language. The ANSI standards for these languages read like complex legal documents.

You are forced to write different code for different data types simply to copy the data, ask for its current length, concatenate it, compare it etc. The manuals for those languages are packed with routines such as "strcpy", "strncpy", "memcpy", "strcat", "strlen", "strcmp", "memcmp", etc. that each only work on one of the many types of data.

Much of the complexity surrounds issues of data type. How do you define new types? Which types of data can be mixed? How do you convert one type into another in a way that will keep the compiler happy? When you need to do something requiring flexibility at run-time, you frequently find yourself trying to fake out the compiler.

In these languages the numeric value 4 (for example) can have a different meaning depending on whether it is an int, a char, a short, a double, an int * etc. In Euphoria, 4 is the atom 4, period. Euphoria has something called types as we shall see later, but it is a much simpler concept.

Issues of dynamic storage allocation and deallocation consume a great deal of programmer coding time and debugging time in these other languages, and make the resulting programs much harder to understand. Programs that must run continuously often exhibit storage "leaks", since it takes a great deal of discipline to safely and properly free all blocks of storage once they are no longer needed.

Pointer variables are extensively used. The pointer has been called the "go to" of data structures. It forces programmers to think of data as being bound to a fixed memory location where it can be manipulated in all sorts of low-level, non-portable, tricky ways. A picture of the actual hardware that your program will run on is never far from your mind. Euphoria does not have pointers and does not need them.

2.4 Declarations

2.4.1 Identifiers

Identifiers, which consist of variable names and other user-defined symbols, may be of any length. Upper and lower case are distinct. Identifiers must start with a letter and then be followed by letters, digits or underscores. The following reserved words have special meaning in Euphoria and may not be used as identifiers:

```
   and            end             include          to
   by             exit            not              type
   constant       for             or               while
   do             function        procedure        with
   else           global          return           without
   elsif          if              then             xor
```

The Euphoria editor displays these words in blue.

Identifiers can be used in naming the following:

* procedures
* functions
* types
* variables
* constants

procedures

These perform some computation and may have a list of parameters, e.g.

```
        procedure empty()
        end procedure

        procedure plot(integer x, integer y)
            position(x, y)
            puts(1, '*')
        end procedure
```

There are a fixed number of named parameters, but this is not restrictive since any parameter could be a variable-length sequence of arbitrary objects. In many languages variable-length parameter lists are impossible. In C, you must set up strange mechanisms that are complex enough that the average programmer cannot do it without consulting a manual or a local guru.

A copy of the value of each argument is passed in. The formal parameter variables may be modified inside the procedure but this does not affect the value of the arguments.

Performance Note:

The interpreter does not actually copy sequences or floating-point numbers unless it becomes necessary. For example,

```
            y = {1,2,3,4,5,6,7,8.5,"ABC"}
            x = y
```

The statement x = y does not actually cause a new copy of y to be created. Both x and y will simply "point" to the same sequence. If we later perform x[3] = 9, then a separate sequence will be created for x in memory (although there will still be just one shared copy of 8.5 and "ABC"). The same thing applies to "copies" of arguments passed in to subroutines.

functions

These are just like procedures, but they return a value, and can be used in an expression, e.g.

```
        function max(atom a, atom b)
            if a >= b then
                return a
            else
                return b
            end if
        end function
```

Any Euphoria object can be returned. You can, in effect, have multiple return values, by returning a sequence of objects. e.g.

```
        return {x_pos, y_pos}
```

We will use the general term "subroutine", or simply "routine" when a remark is applicable to both procedures and functions.

```
types
-----
```

These are special functions that may be used in declaring the allowed values for a variable. A type must have exactly one parameter and should return an atom that is either true (non-zero) or false (zero). Types can also be called just like other functions. See 2.4.3 Specifying the Type of a Variable.

```
variables
---------
```

These may be assigned values during execution e.g.

```
        -- x may only be assigned integer values
        integer x
        x = 25

        -- a, b and c may be assigned *any* value
        object a, b, c
        a = {}
        b = a
        c = 0
```

When you declare a variable you name the variable (which protects you against making spelling mistakes later on) and you specify the values that may legally be assigned to the variable during execution of your program.

```
constants
---------
```

These are variables that are assigned an initial value that can never change e.g.

```
        constant MAX = 100
        constant Upper = MAX - 10, Lower = 5
```

```
       constant name_list = {"Fred", "George", "Larry"}
```

The result of any expression can be assigned to a constant, even one involving calls to previously defined functions, but once the assignment is made, the value of the constant variable is "locked in".

Constants may not be declared inside a subroutine.

2.4.2 Scope

A symbol's scope is the portion of the program where that symbol's declaration is in effect, i.e. where that symbol is visible.

In Euphoria, every symbol must be declared before it is used. You can read a Euphoria program from beginning to end without encountering any variables or routines that haven't been defined yet. It is possible to call a routine that comes later in the source, but you must use the special functions, routine_id(), and either call_func() or call_proc() to do it. See Part II - Library Routines - Dynamic Calls.

Procedures, functions and types can call themselves recursively. Mutual recursion, where routine A calls routine B which directly or indirectly calls routine A, requires the routine_id() mechanism.

A symbol is defined from the point where it is declared to the end of its "scope". The scope of a variable declared inside a procedure or function (a "private" variable) ends at the end of the procedure or function. The scope of all other variables, constants, procedures, functions and types ends at the end of the source file in which they are declared and they are referred to as "local", unless the keyword "global" precedes their declaration, in which case their scope extends indefinitely.

When you include a Euphoria file in a main file (see 2.6 Special Top-Level Statements), only the variables and routines declared using the "global" keyword are accessible or even visible to the main file. The other, non-global, declarations in the included file are forgotten at the end of the included file, and you will get an error message, "not declared", if you try to use them in the main file.

Symbols marked as "global" can be used externally. All other symbols can only be used internally within their own file. This information is helpful when maintaining or enhancing the file, or when learning how to use the file. You can make changes to the internal routines and variables, without having to examine other files, or notify other users of the include file.

Sometimes, when using include files developed by others, you will encounter a naming conflict. One of the include file authors has used the same name for a global symbol as one of the other authors. If you have the source, you can simply edit one of the include files to correct the problem, but then you'd have repeat this process whenever a new version of the include file was released. Euphoria has a simpler way to solve this. Using an extension to the include statement, you can say for example:

```
    include johns_file.e as john
    include bills_file.e as bill

    john:x += 1
    bill:x += 2
```

In this case, the variable x was declared in two different files, and you want to refer to both variables in your file. Using the namespace identifier of either john or bill, you can attach a prefix to x to indicate which x you are referring to. We sometimes say that john refers to one namespace, while bill refers to another distinct namespace. You can attach a namespace identifier to any user-defined variable, constant, procedure or function. You can do it to solve a conflict, or simply to make things clearer. A namespace identifier has local scope. It is known only within the file that declares it, i.e. the file that contains the include statement. Different files might define different namespace identifiers to refer to the same included file.

Euphoria encourages you to restrict the scope of symbols. If all symbols were automatically global to the whole program, you might have a lot of naming conflicts, especially in a large program consisting of files written by many different programmers. A naming conflict might cause a compiler error message, or it could lead to a very subtle bug, where different parts of a program accidentally modify the same variable without being aware of it. Try to use the most restrictive scope that you can. Make variables private to one routine where possible, and where that isn't possible, make them local to a file, rather than global to the whole program.

When Euphoria looks up the declaration of a symbol, it first checks the current routine, then the current file, then globals in other files. Symbols that are more local will override symbols that are more global. At the end of the scope of the local symbol, the more global symbol will be visible again.

Constant declarations must be outside of any subroutine. Constants can be global or local, but not private.

Variable declarations inside a subroutine must all appear at the beginning, before the executable statements of the subroutine.

Declarations at the top level, outside of any subroutine, must not be nested inside a loop or if-statement.

The controlling variable used in a for-loop is special. It is automatically declared at the beginning of the loop, and its scope ends at the end of the for-loop. If the loop is inside a function or procedure, the loop variable is a private variable and may not have the same name as any other private variable. When the loop is at the top level, outside of any function or procedure, the loop variable is a local variable and may not have the same name as any other local variable in that file. You can use the same name in many different for-loops as long as the loops aren't nested. You do not declare loop variables as you would other variables. The range of values specified in the for statement defines the legal values of the loop variable - specifying a type would be redundant and is not allowed.

2.4.3 Specifying the Type of a Variable

So far you've already seen some examples of variable types but now we will define types more precisely.

Variable declarations have a type name followed by a list of the variables being declared. For example,

```
        object a

        global integer x, y, z

        procedure fred(sequence q, sequence r)
```

The types: object, sequence, atom and integer are predefined. Variables of type object may take on *any* value. Those declared with type sequence must always be sequences. Those declared with type atom must always be atoms. Those declared with type integer must be atoms with integer values from -1073741824 to +1073741823 inclusive. You can perform exact calculations on larger integer values, up to about 15 decimal digits, but declare them as atom, rather than integer.

Note:

In a procedure or function parameter list like the one for fred() above, a type name may only be followed by a single parameter name.

Performance Note:

Calculations using variables declared as integer will usually be somewhat faster than calculations involving variables declared as atom. If your machine has floating-point hardware, Euphoria will use it to manipulate atoms that aren't representable as integers. If your machine doesn't have floating-point hardware, Euphoria will call software floating-point arithmetic routines contained in ex.exe (or in Windows). You can force ex.exe to bypass any floating-point hardware, by setting an environment variable:

```
            SET NO87=1
```

The slower software routines will be used, but this could be of some advantage if you are worried about the floating-point bug in some early Pentium chips.

To augment the predefined types, you can create user-defined types. All you have to do is define a single-parameter function, but declare it with type ... end type instead of function ... end function. For example,

```
        type hour(integer x)
            return x >= 0 and x <= 23
        end type

        hour h1, h2

        h1 = 10      -- ok
        h2 = 25      -- error! program aborts with a message
```

Variables h1 and h2 can only be assigned integer values in the range 0 to 23 inclusive. After each assignment to h1 or h2 the interpreter will call hour(), passing the new value. The value will first be checked to see if it is an integer (because of "integer x"). If it is, the return statement will be executed to test the value of x (i.e. the new value of h1 or h2). If hour() returns true, execution continues normally. If hour() returns false then the program is aborted with a suitable diagnostic message.

"hour" can be used to declare subroutine parameters as well:

```
        procedure set_time(hour h)
```

set_time() can only be called with a reasonable value for parameter h, otherwise the program will abort with a message.

A variable's type will be checked after each assignment to the variable (except where the compiler can predetermine that a check will not be necessary), and the program will terminate immediately if the type function returns false. Subroutine parameter types are checked each time that the subroutine is called. This checking guarantees that a variable can never have a value that does not belong to the type of that variable.

Unlike other languages, the type of a variable does not affect any calculations on the variable. Only the value of the variable matters in an expression. The type just serves as an error check to prevent any "corruption" of the variable.

User-defined types can catch unexpected logical errors in your program. They are not designed to catch or correct user input errors.

Type checking can be turned off or on between subroutines using the "with type_check" or "without type_check" special statements. It is initially on by default.

Note to Benchmarkers:

> When comparing the speed of Euphoria programs against programs written in other languages, you should specify without type_check at the top of the file. This gives Euphoria permission to skip run-time type checks, thereby saving some execution time. All other checks are still performed, e.g. subscript checking, uninitialized variable checking etc. Even when you turn off type checking, Euphoria reserves the right to make checks at strategic places, since this can actually allow it to run your program *faster* in many cases. So you may still get a type check failure even when you have turned off type checking. Whether type checking is on or off, you will never get a machine-level exception. You will always get a meaningful message from Euphoria when something goes wrong. (This might not be the case when you poke directly into memory, or call routines written in C or machine code.)

Euphoria's method of defining types is simpler than what you will find in other languages, yet Euphoria provides the programmer with *greater* flexibility in defining the legal values for a type of data. Any algorithm can be used to include or exclude values. You can even declare a variable to be of type "object" which will allow it to take on *any* value. Routines can be written to work with very specific types, or very general types.

For many programs, there is little advantage in defining new types, and you may wish to stick with the four predefined types. Unlike other languages, Euphoria's type mechanism is optional. You don't need it to create a program.

However, for larger programs, strict type definitions can aid the process of debugging. Logic errors are caught close to their source and are not allowed to propagate in subtle ways through the rest of the program. Furthermore, it is easier to reason about the misbehavior of a section of code when you are guaranteed that the variables involved always had a legal value, if not the desired value.

Types also provide meaningful, machine-checkable documentation about your program, making it easier for you or others to understand your code at a later date. Combined with the subscript checking, uninitialized variable checking, and other checking that is always present, strict run-time type checking makes debugging much easier in Euphoria than in most other languages. It also increases the reliability of the final program since many latent bugs that would have survived the testing phase in other languages will have been caught by Euphoria.

Anecdote 1:

In porting a large C program to Euphoria, a number of latent bugs were discovered. Although this C program was believed to be totally "correct", we found: a situation where an uninitialized variable was being read; a place where element number "-1" of an array was routinely written and read; and a situation where something was written just off the screen. These problems resulted in errors that weren't easily visible to a casual observer, so they had survived testing of the C code.

Anecdote 2:

The Quick Sort algorithm presented on page 117 of Writing Efficient Programs by Jon Bentley has a subscript error! The algorithm will sometimes read the element just before the beginning of the array to be sorted, and will sometimes read the element just after the end of the array. Whatever garbage is read, the algorithm will still work - this is probably why the bug was never caught. But what if there isn't any (virtual) memory just before or just after the array? Bentley later modifies the algorithm such that this bug goes away -- but he presented this version as being correct. Even the experts need subscript checking!

Performance Note:

When typical user-defined types are used extensively, type checking adds only 20 to 40 percent to execution time. Leave it on unless you really need the extra speed. You might also consider turning it off for just a few heavily-executed routines. Profiling can help with this decision.

2.5 Statements

The following kinds of executable statements are available:

* assignment statement
* procedure call
* if statement
* while statement
* for statement
* return statement
* exit statement

Semicolons are not used in Euphoria, but you are free to put as many statements as you like on one line, or to split a single statement across many lines. You may not split a statement in the middle of an identifier, string, number or keyword.

2.5.1 assignment statement

An assignment statement assigns the value of an expression to a simple variable, or to a subscript or slice of a variable. e.g.

```
x = a + b

y[i] = y[i] + 1

y[i..j] = {1, 2, 3}
```

The previous value of the variable, or element(s) of the subscripted or sliced variable are discarded. For example, suppose x was a 1000-element sequence that we had initialized with:

```
object x

x = repeat(0, 1000)  -- a sequence of 1000 zeros
```

and then later we assigned an atom to x with:

```
x = 7
```

This is perfectly legal since x is declared as an object. The previous value of x, namely the 1000-element sequence, would simply disappear. Actually, the space consumed by the 1000-element sequence will be automatically recycled due to Euphoria's dynamic storage allocation.

Note that the equals symbol '=' is used for both assignment and for equality testing. There is never any confusion because an assignment in Euphoria is a statement only, it can't be used as an expression (as in C).

Assignment with Operator

Euphoria also provides some additional forms of the assignment statement.

To save typing, and to make your code a bit neater, you can combine assignment with one of the operators:

```
     +  -  /  *  &
```

For example, instead of saying:

```
     mylongvarname = mylongvarname + 1
```

You can say:

```
     mylongvarname += 1
```

Instead of saying:

```
     galaxy[q_row][q_col][q_size] = galaxy[q_row][q_col][q_size] * 10
```

You can say:

```
     galaxy[q_row][q_col][q_size] *= 10
```

and instead of saying:

```
     accounts[start..finish] = accounts[start..finish] / 10
```

You can say:

```
     accounts[start..finish] /= 10
```

In general, whenever you have an assignment of the form:

```
     left-hand-side = left-hand-side op expression
```

You can say:

```
     left-hand-side op= expression
```

where op is one of: + - * / &

When the left-hand-side contains multiple subscripts/slices, the "op=" form will usually execute faster than the longer form. When you get used to it, you may find the "op=" form to be slightly more readable than the long form, since you don't have to visually compare the left-hand-side against the copy of itself on the right side.

2.5.2 procedure call

A procedure call starts execution of a procedure, passing it an optional list of argument values. e.g.

```
       plot(x, 23)
```

2.5.3 if statement

An if statement tests a condition to see if it is 0 (false) or non-zero (true) and then executes the appropriate series of statements. There may be optional elsif and else clauses. e.g.

```
        if a < b then
            x = 1
        end if


        if a = 9 and find(0, s) then
            x = 4
            y = 5
        else
            z = 8
        end if


        if char = 'a' then
            x = 1
        elsif char = 'b' or char = 'B' then
            x = 2
        elsif char = 'c' then
            x = 3
        else
            x = -1
        end if
```

Notice that "elsif" is a contraction of "else if", but it's cleaner because it doesn't require an "end if" to go with it. There is just one "end if" for the entire if statement, even when there are many elsif's contained in it.

The if and elsif conditions are tested using short-circuit evaluation.

2.5.4 while statement

A while statement tests a condition to see if it is non-zero (true), and while it is true a loop is executed. e.g.

```
        while x > 0 do
            a = a * 2
            x = x - 1
        end while
```

Short-Circuit Evaluation

When the condition tested by "if", "elsif", or "while" contains "and" or "or" operators, *short-circuit* evaluation will be used. For example,

```
if a < 0 and b > 0 then ...
```

If a < 0 is false, then Euphoria will not bother to test if b is greater than 0. It will assume that the overall result is false. Similarly,

```
if a < 0 or b > 0 then ...
```

if a < 0 is true, then Euphoria will immediately decide that the result true, without testing the value of b.

In general, whenever we have a condition of the form:

```
A and B
```

where A and B can be any two expressions, Euphoria will take a short-cut when A is false and immediately make the overall result false, without even looking at expression B.

Similarly, with:

```
A or B
```

when A is true, Euphoria will skip the evaluation of expression B, and declare the result to be true.

If the expression B contains a call to a function, and that function has possible side-effects, i.e. it might do more than just return a value, you will get a compile-time warning. Older versions (pre-2.1) of Euphoria did not use short-circuit evaluation, and it's possible that some old code will no longer work correctly, although a search of the Euphoria archives did not turn up any programs that depend on side-effects in this way.

The expression, B, could contain something that would normally cause a run-time error. If Euphoria skips the evaluation of B, the error will not be discovered. For instance:

```
if x != 0 and 1/x > 10 then  -- divide by zero error avoided

while 1 or {1,2,3,4,5} do    -- illegal sequence result avoided
```

B could even contain uninitialized variables, out-of-bounds subscripts etc.

This may look like sloppy coding, but in fact it often allows you to write something in a simpler and more readable way. For instance:

```
if atom(x) or length(x)=1 then
```

Without short-circuiting, you would have a problem when x was an atom, since length is not defined for atoms. With short-circuiting, length(x) will only be checked when x is a sequence. Similarly:

```
-- find 'a' or 'A' in s
i = 1
while i <= length(s) and s[i] != 'a' and s[i] != 'A' do
     i += 1
end while
```

In this loop the variable i might eventually become greater than length(s). Without short-circuit evaluation, a subscript out-of-bounds error will occur when s[i] is evaluated on the final iteration. With short-circuiting, the loop will terminate immediately when i <= length(s) becomes false. Euphoria will not evaluate s[i] != 'a' and will not evaluate s[i] != 'A'. No subscript error will occur.

Short-circuit evaluation of "and" and "or" takes place for "if", "elsif" and "while" conditions only. It is not used in other contexts. For example, the assignment statement:

```
       x = 1 or {1,2,3,4,5}  -- x should be set to {1,1,1,1,1}
```

If short-circuiting were used here, we would set x to 1, and not even look at {1,2,3,4,5}. This would be wrong. Short-circuiting can be used in if/elsif/while conditions because we only care if the result is true or false, and conditions are required to produce an atom as a result.

2.5.5 for statement

A for statement sets up a special loop with a controlling loop variable that runs from an initial value up or down to some final value. e.g.

```
       for i = 1 to 10 do
           ? i   -- ? is a short form for print()
       end for

       -- fractional numbers allowed too
       for i = 10.0 to 20.5 by 0.3 do
           for j = 20 to 10 by -2 do    -- counting down
               ? {i, j}
           end for
       end for
```

The loop variable is declared automatically and exists until the end of the loop. Outside of the loop the variable has no value and is not even declared. If you need its final value, copy it into another variable before leaving the loop. The compiler will not allow any assignments to a loop variable. The initial value, loop limit and increment must all be atoms. If no increment is specified then +1 is assumed. The limit and increment values are established when the loop is entered, and are not affected by anything that happens during the execution of the loop. See also the scope of the loop variable in 2.4.2 Scope.

2.5.6 return statement

A return statement returns immediately from a subroutine. If the subroutine is a function or type then a value must also be returned. e.g.

```
       return

       return {50, "FRED", {}}
```

2.5.7 exit statement

An exit statement may appear inside a while-loop or a for-loop. It causes immediate termination of the loop, with control passing to the first statement after the loop. e.g.

```
       for i = 1 to 100 do
           if a[i] = x then
               location = i
               exit
           end if
       end for
```

It is also quite common to see something like this:

```
       constant TRUE = 1

       while TRUE do
           ...
           if some_condition then
               exit
           end if
           ...
       end while
```

i.e. an "infinite" while-loop that actually terminates via an exit statement at some arbitrary point in the body of the loop.

Performance Note:

Euphoria optimizes this type of loop. At run-time, no test is performed at the top of the loop. There's just a simple unconditional jump from end while back to the first statement inside the loop.

With ex.exe, if you happen to create a real infinite loop, with no input/output taking place, there is no easy way to stop it. You will have to type Control-Alt-Delete to either reboot, or (under Windows) terminate your DOS prompt session. If the program had files open for writing, it would be advisable to run scandisk to check your file system integrity. Only when your program is waiting for keyboard input, will control-c abort the program (unless allow_break(0) was used).

With exw.exe or exu, control-c will always stop your program immediately.

2.6 Special Top-Level Statements

Euphoria processes your .ex file in one pass, starting at the first line and proceeding through to the last line. When a procedure or function definition is encountered, the routine is checked for syntax and converted into an internal form, but no execution takes place. When a statement that is outside of any routine is encountered, it is checked for syntax, converted into an internal form and then immediately executed. A common practice is to immediately initialize a global variable, just after its declaration. If your .ex file contains only routine definitions, but no immediate execution statements, then nothing will happen when you try to run it (other than syntax checking). You need to have an immediate statement to call your main routine (see 1.1 Example Program). It is quite possible to have a .ex file with nothing but immediate statements, for example you might want to use Euphoria as a simple calculator, typing in just one print (or ?) statement into a file, and then executing it.

As we have seen, you can use any Euphoria statement, including for-loops, while-loops, if statements etc. (but not return), at the top level i.e. *outside* of any function or procedure. In addition, the following special statements may *only* appear at the top level:

* include
* with / without

2.6.1 include

When you write a large program it is often helpful to break it up into logically separate files, by using include statements. Sometimes you will want to reuse some code that you have previously written, or that someone else has written. Rather than copy this code into your main program, you can use an include statement to refer to the file containing the code. The first form of the include statement is:

include filename

This reads in (compiles) a Euphoria source file.

For example:

```
        include graphics.e
```

Any top-level code in the included file will be executed.

Any global symbols that have already been defined in the main file will be visible in the included file.

N.B. Only those symbols defined as "global" in the included file will be visible (accessible) in the remainder of the program.

If an absolute filename is given, Euphoria will use it. When a relative filename is given, Euphoria will first look for it in the same directory as the main file given on the ex (or exw or exu) command-line. If it's not there, and you've defined an environment variable, EUINC, it will search each directory listed in EUINC (from left to right). Finally, if it still hasn't found the file, it will search euphoria\include. This directory contains the standard Euphoria include files. The environment variable EUDIR tells ex.exe/exw.exe/exu where to find your euphoria directory. EUINC should be a list of directories, separated by semicolons (colons on Linux), similar in form

to your PATH variable. It can be added to your AUTOEXEC.BAT file, e.g.
SET EUINC=C:\EU\MYFILES;C:\EU\WIN32LIB
This lets you organize your include files according to application areas, and avoid adding numerous unrelated files to euphoria\include.

An included file can include other files. In fact, you can "nest" included files up to 10 levels deep.

Include file names typically end in .e, or sometimes .ew or .eu (when they are intended for use with Windows or Linux). This is just a convention. It is not required.

If your filename (or path) contains blanks, you must enclose it in double-quotes, otherwise quotes are optional. For example:

```
            include "c:\program files\myfile.e"
```

Other than possibly defining a new namespace identifier (see below), an include statement will be quietly ignored if a file with the same name has already been included.

An include statement must be written on a line by itself. Only a comment can appear after it on the same line.

The second form of the include statement is:

include filename as namespace_identifier

This is just like the simple include, but it also defines a namespace identifier that can be attached to global symbols in the included file that you want to refer to in the main file. This might be necessary to disambiguate references to those symbols, or you might feel that it makes your code more readable. See Scope Rules for more.

2.6.2 with / without

These special statements affect the way that Euphoria translates your program into internal form. They are not meant to change the logic of your program, but they may affect the diagnostic information that you get from running your program. See 3. Debugging and Profiling for more information.

with

This turns on one of the options: profile, profile_time, trace, warning or type_check . Options warning and type_check are initially on, while profile, profile_time and trace are initially off.

Any warnings that are issued will appear on your screen after your program has finished execution. Warnings indicate very minor problems. A warning will never stop your program from executing.

without

This turns off one of the above options.

There is also a rarely-used special with option where a code number appears after with. In previous releases this code was used by RDS to make a file exempt from adding to the statement count in the Public Domain Edition.

You can select any combination of settings, and you can change the settings, but the changes must occur *between* subroutines, not within a subroutine. The only exception is that you can only turn on one type of profiling for a given run of your program.

An included file inherits the with/without settings in effect at the point where it is included. An included file can change these settings, but they will revert back to their original state at the end of the included file. For instance, an included file might turn off warnings for itself and (initially) for any files that it includes, but this will not turn off warnings for the main file.

3. Debugging and Profiling

3.1 Debugging
=============

Debugging in Euphoria is much easier than in most other programming languages. Extensive run-time checking provided by the Euphoria interpreter catches many bugs that in other languages might take hours of your time to track down. When the interpreter catches an error, you will always get a brief report on your screen, and a detailed report in a file called "ex.err". These reports include a full English description of what happened, along with a call-stack traceback. The file ex.err will also have a dump of all variable values, and optionally a list of the most recently executed statements. For extremely large sequences, only a partial dump is shown. If ex.err is not convenient, you can choose another file name, anywhere on your system, by calling crash_file().

In addition, you are able to create user-defined types that precisely determine the set of legal values for each of your variables. An error report will occur the moment that one of your variables is assigned an illegal value.

Sometimes a program will misbehave without failing any run-time checks. In any programming language it may be a good idea to simply study the source code and rethink the algorithm that you have coded. It may also be useful to insert print statements at strategic locations in order to monitor the internal logic of the program. This approach is particularly convenient in an interpreted language like Euphoria since you can simply edit the source and rerun the program without waiting for a re-compile/re-link.

The interpreter provides you with additional powerful tools for debugging. Using trace(1) you can trace the execution of your program on one screen while you witness the output of your program on another. trace(2) is the same as trace(1) but the trace screen will be in monochrome. Finally, using trace(3), you can log all executed statements to a file called ctrace.out.

The full trace facility is part of the Euphoria Complete Edition, but it's enabled in the Public Domain Edition for programs up to 300 statements.

with trace / without trace special statements select the parts of your program that are available for tracing. Often you will simply insert a "with trace" statement at the very beginning of your source code to make it all traceable. Sometimes it is better to place the first "with trace" after all of your user-defined types, so you don't trace into these routines after each assignment to a variable. At other times, you may know exactly which routine or routines you are interested in tracing, and you will want to select only these ones. Of course, once you are in the trace window, you can skip viewing the execution of any routine by pressing down-arrow on the keyboard rather than Enter.

Only traceable lines can appear in ctrace.out or in ex.err as "Traced lines leading up to the failure" should a run-time error occur. If you want this information and didn't get it, you should insert a "with trace" and then rerun your program. Execution will be slower when lines compiled "with trace" are executed, especially when trace(3) is used.

After you have predetermined the lines that are traceable, your program must then dynamically cause the trace facility to be activated by executing a trace() statement. You could simply say:

```
       with trace
       trace(1)
```

at the top of your program, so you can start tracing from the beginning of execution. More commonly, you will want to trigger tracing when a certain routine is entered, or when some condition arises. e.g.

```
       if x < 0 then
           trace(1)
       end if
```

You can turn off tracing by executing a trace(0) statement. You can also turn it off interactively by typing 'q' to quit tracing. Remember that "with trace" must appear outside of any routine, whereas trace() can appear inside a routine or outside.

You might want to turn on tracing from within a type. Suppose you run your program and it fails, with the ex.err file showing that one of your variables has been set to a strange, although not illegal value, and you wonder how it could have happened. Simply create a type for that variable that executes trace(1) if the value being assigned to the variable is the strange one that you are interested in. e.g.

```
       type positive_int(integer x)
           if x = 99 then
               trace(1) -- how can this be???
               return 1 -- keep going
           else
               return x > 0
           end if
       end type
```

When positive_int() returns, you will see the exact statement that caused your variable to be set to the strange value, and you will be able to check the values of other variables. You will also be able to check the output screen to see what has happened up to this precise moment. If you define positive_int() so it returns 0 for the strange value (99) instead of 1, you can force a diagnostic dump into ex.err.

3.1.1 The Trace Screen

When a trace(1) or trace(2) statement is executed by the interpreter, your main output screen is saved and a trace screen appears. It shows a view of your program with the statement that will be executed next highlighted, and several statements before and after showing as well. Several lines at the bottom of the screen are reserved for displaying variable names and values. The top line shows the commands that you can enter at this point:

F1	- display main output screen - take a look at your program's output so far
F2	- redisplay trace screen. Press this key while viewing the main output screen to return to the trace display.
Enter	- execute the currently-highlighted statement only
down-arrow	- continue execution and break when any statement coming after this one in the source listing is about to be executed. This lets you skip over subroutine calls. It also lets you stop on the first statement following the end of a for-loop or while-loop without having to witness all iterations of the loop.
?	- display the value of a variable. After hitting ? you will be prompted for the name of the variable. Many variables are displayed for you automatically as they are assigned a value. If a variable is not currently being displayed, or is only partially displayed, you can ask for it. Large sequences are limited to one line on the trace screen, but when you ask for the value of a variable that contains a large sequence, the screen will clear, and you can scroll through a pretty-printed display of the sequence. You will then be returned to the trace screen, where only one line of the variable is displayed. Variables that are not defined at this point in the program cannot be shown. Variables that have not yet been initialized will have "< NO VALUE >" beside their name. Only variables, not general expressions, can be displayed. As you step through execution of the program, the system will update any values showing on the screen. Occasionally it will remove variables that are no longer in scope, or that haven't been updated in a long time compared with newer, recently-updated variables.
q	- quit tracing and resume normal execution. Tracing will start again when the next trace(1) is executed.

```
Q               - quit tracing and let the program run freely to its normal
                  completion. trace() statements will be ignored.

!               - this will abort execution of your program. A traceback and
                  dump of variable values will go to ex.err.
```

As you trace your program, variable names and values appear automatically in the bottom portion of the screen. Whenever a variable is assigned-to, you will see its name and new value appear at the bottom. This value is always kept up-to-date. Private variables are automatically cleared from the screen when their routine returns. When the variable display area is full, least-recently referenced variables will be discarded to make room for new variables. The value of a long sequence will be cut off after 80 characters.

For your convenience, numbers that are in the range of printable ASCII characters (32-127) are displayed along with the ASCII character itself. The ASCII character will be in a different color (or in quotes in a mono display). This is done for all variables, since Euphoria does not know in general whether you are thinking of a number as an ASCII character or not. You will also see ASCII characters (in quotes) in ex.err. This can make for a rather "busy" display, but the ASCII information is often very useful.

The trace screen adopts the same graphics mode as the main output screen. This makes flipping between them quicker and easier.

When a traced program requests keyboard input, the main output screen will appear, to let you type your input as you normally would. This works fine for gets() (read one line) input. When get_key() (quickly sample the keyboard) is called you will be given 8 seconds to type a character, otherwise it is assumed that there is no input for this call to get_key(). This allows you to test the case of input and also the case of no input for get_key().

3.1.2 The Trace File

When your program calls trace(3), tracing to a file is activated. The file, ctrace.out will be created in the current directory. It contains the last 500 Euphoria statements that your program executed. It is set up as a circular buffer that holds a maximum of 500 statements. Whenever the end of ctrace.out is reached, the next statement is written back at the beginning. The very last statement executed is always followed by "=== THE END ===". Because it's circular, the last statement executed could appear anywhere in ctrace.out. The statement coming after "=== THE END ===" is the 500th-last.

This form of tracing is supported by both the interpreter and the Complete Edition of the Euphoria To C Translator. It is particularly useful when a machine-level error occurs that prevents Euphoria from writing out an ex.err diagnostic file. By looking at the last statement executed, you may be able to guess why the program crashed. Perhaps the last statement was a poke() into an illegal area of memory. Perhaps it was a call to a C routine. In some cases it might be a bug in the interpreter or the Translator.

The source code for a statement is written to ctrace.out, and flushed, just before the statement is performed, so the crash will likely have happened during execution of the final statement that you see in ctrace.out.

3.2 Profiling (Complete Edition Only)

=============

If you specify "with profile" (DOS32, WIN32 or Linux), or "with profile_time" (DOS32 only) then a special listing of your program, called a "profile", will be produced by the interpreter when your program finishes execution. This listing is written to the file "ex.pro" in the current directory.

There are two types of profiling available: execution-count profiling, and time profiling. You get execution-count profiling when you specify "with profile". You get time profiling when you specify "with profile_time". You can't mix the two types of profiling in a single run of your program. You need to make two separate runs.

We ran the sieve8k.ex benchmark program in demo\bench under both types of profiling. The results are in sieve8k.pro (execution-count profiling) and sieve8k.pro2 (time profiling).

Execution-count profiling shows precisely how many times each statement in your program was executed. If the statement was never executed the count field will be blank.

Time profiling (DOS32 only) shows an estimate of the total time spent executing each statement. This estimate is expressed as a percentage of the time spent profiling your program. If a statement was never sampled, the percentage field will be blank. If you see 0.00 it means the statement was sampled, but not enough to get a score of 0.01.

Only statements compiled "with profile" or "with profile_time" are shown in the listing. Normally you will specify either "with profile" or "with profile_time" at the top of your main .ex file, so you can get a complete listing. View this file with the Euphoria editor to see a color display.

Profiling can help you in many ways:

* it lets you see which statements are heavily executed, as a clue to speeding up your program
* it lets you verify that your program is actually working the way you intended
* it can provide you with statistics about the input data
* it lets you see which sections of code were never tested - don't let your users be the first!

Sometimes you will want to focus on a particular action performed by your program. For example, in the Language War game, we found that the game in general was fast enough, but when a planet exploded, shooting 2500 pixels off in all directions, the game slowed down. We wanted to speed up the explosion routine. We didn't care about the rest of the code. The solution was to call profile(0) at the beginning of Language War, just after "with profile_time", to turn off profiling, and then to call profile(1) at the beginning of the explosion routine and profile(0) at the end of the routine. In this way we could run the game, creating numerous explosions, and logging a lot of samples, just for the explosion effect. If samples were charged against other lower-level routines, we knew that those samples occurred during an explosion. If we had simply profiled the whole program, the picture would not have been clear, as the lower-level routines would also have been used for moving ships, drawing phasors etc. profile() can help in the same way when you do execution-count profiling.

3.2.1 Some Further Notes on Time Profiling

With each click of the system clock, an interrupt is generated. When you specify "with profile_time" Euphoria will sample your program to see which statement is being executed at the exact moment that each interrupt occurs.

These interrupts normally occur 18.2 times per second, but if you call tick_rate() you can choose a much higher rate and thus get a more accurate time profile, since it will be based on more samples. By default, if you haven't called tick_rate(), then tick_rate(100) will be called automatically when you start profiling. You can set it even higher (up to say 1000) but you may start to affect your program's performance.

Each sample requires 4 bytes of memory and buffer space is normally reserved for 25000 samples. If you need more than 25000 samples you can request it:

```
       with profile_time 100000
```

will reserve space for 100000 samples (for example). If the buffer overflows you'll see a warning at the top of ex.pro. At 100 samples per second your program can run for 250 seconds before using up the default 25000 samples. It's not feasible for Euphoria to dynamically enlarge the sample buffer during the handling of an interrupt. That's why you might have to specify it in your program. After completing each top-level executable statement, Euphoria will process the samples accumulated so far, and free up the buffer for more samples. In this way the profile can be based on more samples than you have actually reserved space for.

The percentages shown in the left margin of ex.pro, are calculated by dividing the number of times that a particular statement was sampled, by the total number of samples taken. e.g. if a statement were sampled 50 times out of a total of 500 samples, then a value of 10.0 (10 per cent) would appear in the margin beside that statement. When profiling is disabled with profile(0), interrupts are ignored, no samples are taken and the total number of samples does not increase.

By taking more samples you can get more accurate results. However, one situation to watch out for is the case where a program synchronizes itself to the clock interrupt, by waiting for time() to advance. The statements executed just after the point where the clock advances might *never* be sampled, which could give you a very distorted picture. e.g.

```
       while time() < LIMIT do
       end while
       x += 1 -- This statement will never be sampled
```

Sometimes you will see a significant percentage beside a return statement. This is usually due to time spent deallocating storage for temporary and private variables used within the routine. Significant storage deallocation time can also occur when you assign a new value to a large sequence.

If disk swapping starts to happen, you may see large times attributed to statements that need to access the swap file, such as statements that access elements of a large swapped-out sequence.