Enigma

A programming language Version 0.1

Niels Serup (ns@metanohi.org)

Table of Contents

1 A short introduction

Today, in the world of programming, the commonly used programming languages have — to some extent — adapted the same basic syntax. Both C, C++, Java, Python and Ruby (and others) all share several paradigms, even though they are still very different. Enigma, on the other hand, is very different from the "common" languages. See this small program to get an idea of Enigma:

```
{/a,b/
    a 2 ! multiply = c;
    b c ! add = return;
} = odd-add;

9 4 ! odd-add | stdout temp ! write;
```

Here, 9 is first multiplied with 2 and then added to 4. The program then prints the result (22).

With only a few constructs and built-in variables, Enigma is quite small and relatively simple. Any logic can be expressed in Enigma. The above example may seem odd, but its aspects will be dealt with in the next chapter.

Enigma is *not* the kind of programming language that likes to control programmers. When programming in Enigma, one creates functions that call other functions — instead of creating functions that are called by built-in functions.

On the other hand, Enigma is very restrictive. There are limits to certain things. If these limits were not present, chaos would ensue. Perhaps.

Like many other languages, Enigma is quite fond of pointers. In Enigma, all variables are pointers. When `x` is assigned to `a`, and `b` is assigned to `a`, changing `b` also changes `a`. As a matter of fact, when two pointers points to the same place, they are **forever** linked together.

Enigma is not recommended for serious use. Use with **caution**!

2 The language

Enigma consists of functions. Any file being parsed is a function. Within functions there can be local variables. These variables are to disappear when the function exits.

Functions have return values. These values can be determined by the programmer.

Within functions, there are commands. These commands constitute the very base of Enigma. Commands can interact with built-in functions, meaning that writing to and reading from files, calculating, as well as manipulating strings and lists, is possible.

There are no simple types. Enigma lies in the world of objects. Numbers, strings, lists.. It's just objects. Also, all commands must end with a semicolon`;`

2.1 Operators

Enigma has 4 operators: `!`, `=`, `|`, and `*`. `!` and `=` are the most important ones.

To assign values to variables, use the `=` operator. For example:

```
obj = hello-string;
```

Here, the variable `hello-string` receives the object `obj`. Note the syntax. The objects comes first. Unless `hello-string` is already defined, it is created as a local variable.

Now, say we have a function called `think` and we want to send objects to it. This is how that's done:

```
obj1 obj2 obj3 ! think;
```

There are no limits to the amount of objects that can be sent to a function. Whitespace characters are used to separate objects, also when assigning them to a variable. There *is* a difference between using one object and more objects as arguments, though. When using only one object, that object is simply used, but when using more objects, a list containing all objects is created and transmitted instead. So, in reality, when assigning two or more objects to a variable, the variable points to a list with the objects. More about lists later.

The `|` operator is merely a shortcut operator. It can merge several commands into one, long command. See the following example:

```
# Long version
a ! b = c;
d c ! e = c;
# Short version
a ! b | d temp ! e = c;
```

Here, `a` is sent to `b`, which is then — together with `d` — sent to `e` and saved in `c`. The `|` acts like the semicolon, but it copies the return value of the last function (in this case `b`) and stores the value in the local variable `temp`. Hackers are encouraged to use this operator.

The final operator, `*`, has the same function as `=`, except that it works on a global level. So, if the code is `b = a;` and `a` does not exist, it is created — in the global space.

2.2 Numbers

Numbers in Enigma can be arbitrarily large. They are written as one usually writes numbers. Currently only the decimal system is supported.

```
7587 = age;
2556 ! numfunc;
```

2.3 Strings

Strings are defined using the " character.

```
"This is a string!" " This too!" ! add;
"A" = A;
"Now comes
a newline" = nl;
"\"\\\n\t\
" = escaped;
```

2.4 Lists

Lists are special in Enigma. They are used a lot, but unlike eg. numbers and strings they cannot be defined in their own syntax. When one writes this:

```
123 "witch" = lst;
```

..one creates a list. Lists can be nested.

2.5 Booleans

Booleans are used a lot in Enigma. The predefined variables `true` and `false` can be used.

2.6 Files

Files can only be created using the built-in function `open`. Files are either readable or writeable. If a file is writeable, text written to the file may optionally be appended to any existing text.

```
"a" "r" ! open = f1;
"b" "w" ! open = f2;
"c" "a" ! open = f3;

# Read one character
f1 ! read | stdout temp ! write;
# Write a string
f2 "Aaar" ! write;
# Append a string
f3 "Ouu" ! write;

f1!close;f2!close;f3!close;
```

To read an entire file, one must use a loop construct. See Section 2.9 [Loops], page 5. When there are no more characters left, the `read` function returns an empty string. Using equality testing, reading a complete file is thus possible. See Section 2.8 [Conditionals], page 4.

```
filename "r" ! open = f;
{f ! read = c; stdout c ! write; c "" ! equal ! not = return;} ! loop;
```

At this point, it may not be obvious why this works. This will be explained in the next sections.

2.7 Functions

Functions are defined and called using the following syntax:

```
# Defining them
{stdout "Hello" ! write;} = func;
{/x,y/stdout "x=" x ", y=" y!write;} = coorprint;
{stdout args ! write;} = aprint;
```

```
# Calling them
!func;                  # Outputs "Hello"
52 12 ! coorprint;      # Outputs "x=52, y=12"
999 "abc" 21 ! aprint; # Outputs "999 abc 21"
```

Arguments can thus be accessed either by using the `/.../` syntax in the beginning of the function or by using the local `args` variable. Furthermore, to make *all* new variables save in the global space, prepend the function content with a `*`. This will have the same effect as replacing all `=` signs in the function with the `*` operator.

Remember that your file is also a function, just without the curly brackets. This means that when working on the top level of your program, `args` actually refer to the arguments specified on the command line. All arguments are strings, but built-in functions that can convert them to numbers exist.

The previous example functions did not have any return values. This is merely because they didn't need such things. To define a return value, the return value must be sent to the special variable `return`.

```
{/a,b/a b ! multiply = return;} = 2mul;
33 3 ! 2mul = 2mult;
stdout 2mult ! write;
```

The execution of a function does *not* stop when a value is assigned to `return`. This has both advantages and disadvantages.

Many things are almost impossible to do without the built-in functions. Some are completely impossible. See Chapter 6 [Built-In], page 10. As all objects in Enigma are pointers, several built-in functions actually modify the objects sent to them as arguments. To solve this problem, one must use the `clone` function.

2.8 Conditionals

Any value can be related to another value. 78 is less than 85. 2 equals 2. "Hi" is not "Bye". Enigma comes with a basic set of functions that can be used to test these relations. The functions `greater`, `lesser`, `equal`, `not`, `or` and `and` are available. These functions return either true or false. To run a function on the condition that a relational function has returned true, use the function `act`.

While `lesser` and `greater` are aimed at numbers only, and `and`, `or` and `not` are aimed at booleans only, `equal` can be used on all objects. It tests for all equalities.

```
2 = a;
4 = b;

a 2 ! add | temp b ! equal = cond1;
"47" ! num | temp 48 ! greater = cond2;

cond1 cond2 ! and | temp write stdout "4 = 4 and 47 > 48\n" ! act;
cond1 cond2 ! or | temp write stdout "4 = 4 or 47 > 48\n" ! act;
cond2 ! not | cond1 temp ! and | temp write stdout "4 = 4 and !(47 >
48)\n" ! act;
```

Even simpler, one can do this (though there's no real need for it):

```
# Instead of stdout "Hello" !write;
true write stdout "Hello" !act;
```

2.9 Loops

There are several ways to loop pieces of code in Enigma. One can create a function that calls itself, for example. It is, however, advised to use the `loop` function.

```
# Infinite loop
{stdout "Forever\n" ! write; true = return;} ! loop;
```

`loop` works a bit like `act`, except that it runs on the premise of the return value of the function it calls. If it returns true, or any other value that can be considered true (non-zero numbers, non-empty strings, files, etc.), it runs the function again. If it returns false, the loop stops. It naturally runs the first time no matter what (no return value has been created yet).

2.10 Undefined behaviour

Enigma is still new. The way built-in functions act, the way error messages appear, and the way some odd details of the language works are still undefined. In general, when something seems likely to work, it will work. But it's not necessarily defined to work.

3 Future aspects

As Enigma is right now, it is quite limited. The only way that it can interact with the rest of one's system is by reading and writing files and by executing shell commands using the `system` built-in function. This obviously needs to be improved. A foreign function interface system must be implemented in version 0.2. It should be possible for Enigma to do everything C can.

The number of built-in functions seems reasonable, but it may be a good idea to implement a few more. These eventual functions should be focused on making things easier. Specifically, an import function should be considered. It is possible to create an import function in Enigma directly, but it's not exactly fast. Introducing a "compile" function should solve that.

4 Implementations

As of right now (beginning of June 2010) there is only one implementation. It is written in Java and is called enigma (with a lowercase e). The code is bulky and was not written by an experienced Java hacker. For v0.2 it may be a good idea to write the interpreter in C instead. Implementing a foreign function interface should be easier that way.

The current implementation can be downloaded at `http://metanohi.org/projects/enigma`.

5 Comparison with other languages

5.1 Assigning values

C:

```
type var = value;
```

Python:

```
var = value
```

Enigma:

```
value = var;
```

5.2 Conditionals

C:

```
if ((a == 2 && b < 5) || (c != 4 && !d)) do_something();
```

Python:

```
if (a == 2 and b < 5) or (c != 4 and not d): do_something()
```

Enigma:

```
a 2 ! equal = cond-a;
b 5 ! lesser = cond-b;
c 4 ! equal ! not = cond-c;
d ! not = cond-d;
cond-a cond-b ! and = ncond-ab;
cond-c cond-d ! and = ncond-cd;
ncond-ab ncond-cd ! or = actcond;
actcond do-something ! act;
```

5.3 Looping

C:

```
while (thinking) run();

for (int i = 0; i < 18; i++) run(i);
```

Python:

```
while thinking: run()

for i in range(18): run(i)
```

Enigma:

```
{!run; thinking ! clone = return;} ! loop;

0 = i;
{i ! run; i 1 ! add; i 18 ! equal = return;} ! loop;
```

5.4 Functions

C:

```
int time(float space) {
    return (int) space / 3;
}

t = time(81.65);
```

Python:

```
def time(space):
    return int(space / 3)

t = time(81.65)
```

Enigma:

```
{/space/space 3 ! divide ! clone ! round = return;} = time;

81.65 ! time = t;
```

6 Built-in values and functions

6.1 Values

The following variables are special and can be accessed anytime. They can even be overwritten.

`args, return, temp, stdin, stderr, stdout, zero, true, false, none, @pi, @e, cwd, cpd, fnm`

`zero` is a fake writable file that does nothing. `none` is an abstract variable with an internal value of.. none. `@pi` is Pi. `@e` is Euler's number. `cwd` is the directory from which Enigma is run, `cpd` is the directory in which the current program file resides, and `fnm` is the filename of the current program file. The rest are self-explanatory — and `args`, `return` and `temp` are extra special.

6.2 Functions

Enigma has few built-in functions. But they do exist.

`str(OBJECT...)`: Converts all supplied values to strings.

`num(OBJECT...)`: Converts all supplied values to numbers.

`list(OBJECT...)`: Puts all supplied values in a list and destroys nested lists at one level.

`bool(OBJECT...)`: Converts all supplied values to booleans.

`code(OBJECT...)`: Converts all supplied values to code strings.

`repr(OBJECT...)`: Converts all supplied values to representational strings. Strings get prepended and appended by `"` characters, code objects are surrounded by `{` and `}`, and so on.

`type(OBJECT...)`: Converts all supplied values to type strings (string, number, file, etc.).

`len(OBJECT...)`: Converts all supplied values to their length. This can be used for both strings and lists.

`clone(OBJECT...)`: Clones all supplied values. When an object is cloned, its value is assigned to a new variable. The list of clones is returned.

`slice(LIST|STRING, NUMBER, NUMBER)`: Returns a list value sliced according to the first and second number if the first variable is a list variable, or a substring if the first variable is a string variable.

`loop(FUNCTION, [OBJECT]...)`: Executes function until it returns false. Sends the rest of the specified variables (if any) to the function as arguments.

`open(STRING, STRING)`: Opens the file by the name of the first string, using the second string as its guide as to how it should be opened. `"r"` means to read, `"w"` means to write, and `"a"` means to append. Returns a file object.

`close(FILE...)`: Closes files.

`read(FILE)`: Reads one character of a file and returns it. If no more characters are present, an empty string is returned.

`write(FILE, STRING)`: Writes a string to a file.

`greater(NUMBER, NUMBER...`: Checks if the first number is greater than the rest and returns true or false.

`lesser(NUMBER, NUMBER...`: Checks if the first number is lesser than the rest and returns true or false.

`equal(OBJECT, OBJECT...`: Checks if the objects are equal and returns true or false.

`and(BOOLEAN...)`: Checks if all booleans are true and returns true or false.

`or(BOOLEAN...)`: Checks if at least one boolean is true and returns true or false.

`not(BOOLEAN...)`: Converts values of true to values of false, and vice-versa.

`act(BOOLEAN, FUNCTION, [OBJECT]...)`: Run the function with the optional objects as arguments if the boolean is true.

`system(STRING...)`: Join the strings and run the result as a system command.

`add(NUMBER|STRING|CODE|LIST...)`: Add objects together. Numbers, strings, code strings and lists can be used, but only with similar types. The type of the first object determines what types the rest must be. The result is stored in the first object and also returned.

`subtract(NUMBER...)`: Subtract numbers from each other. The first object receives the final number. It is also returned.

`multiply(NUMBER...)`: Multiply numbers with each other. The first object receives the final number. It is also returned.

`divide(NUMBER...)`: Divide numbers with each other. The first object receives the final number. It is also returned.

`mod(NUMBER, NUMBER)`: Finds the remainder of the first number divided with the second number and returns it as a new variable.

`pow(NUMBER, NUMBER)`: Returns first number^second number.

`log(NUMBER, [NUMBER])`: Returns the logarithm of the first number. If the second number is not specified, the natural logarith is used.

`random()`: Returns a random number between 0 and 1.

`abs(NUMBER...)`: Sets and returns absolute values of numbers.

`round(NUMBER, NUMBER)`: Rounds the first number with x decimals, where x is the second number.

`floor(NUMBER, NUMBER)`: Floors the first number with x decimals, where x is the second number.

`ceil(NUMBER, NUMBER)`: "Ceils" the first number with x decimals, where x is the second number.

`sin(NUMBER...)`: Sets and returns sine values.

`cos(NUMBER...)`: Sets and returns cosine values.

`tan(NUMBER...)`: Sets and returns tangent values.

`asin(NUMBER...)`: Sets and returns arcsine values.

`acos(NUMBER...)`: Sets and returns arccosine values.

`atan(NUMBER...)`: Sets and returns arctangent values.

`sinh(NUMBER...)`: Sets and returns hyperbolic sine values.

`cosh(NUMBER...)`: Sets and returns hyperbolic cosine values.

`tanh(NUMBER...)`: Sets and returns hyperbolic tangent values.

Appendix A GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index