
ENiX

ENiX 2 Handbook

Purpose of ENiX 2:

The point of ENiX 2 is to expand on ENiX 1 vocabulary of algoithms for solving problems and to seemlessly integrate these algorithms into one system, and for ENiX to have better awareness of its limits. This document assumes that you have read ENiX1.pdf documenting the basics of ENiX 1. **ENiX is still an experiment** more than anything ENiX is an experimental framework for clearly highlighting concepts that will be influencial for future technologies. Also this documentation is as much of a journel for these concepts as it is a user manual so that I do not forget. Areas for improvements include:

- HTML compatible program output to enable the output from ENiX 2 to be fed through a Perl / Apache interface and used on a website,
- Greater integer processing ranges because ENiX 1 could only process integers in the range $\boldsymbol{Z}\epsilon[0, 255]$ which was fine for purposes of demonstrations and experiments but in terms of actual problem solving doesn't provide us with much of a framework for dealing with more complex problems that human reasoning would be slow at processing.
- Clearer explanations of methods for explaining the observed data. In particular, a graphical (via HTML) method of displaying a logic circuit-like system for reproducing the observed output.
- Because ENiX 2 is also very experimental, we need more tools for debugging problems so that we can have a better idea of the types of problems we encounter. In particular at this stage a visual dependancy checker would be great for illustrating graphically what scenerios the observed data is affected by. Also a trivial but useful feature would be a module for expanding 1D number sequence vectors into 2D matrices of binary, because this is a feature which was sorely lacked in ENiX 1 and examining binary output is something I regularly find I have to do with pen and paper in order to manually check ENiX output.
- Algorithm for dealing with logical number patterns, polynomial based number patterns as well as recurrence relations.
- Basic feature for manually defining solutions to problems and the ability to save and load problem solutions from memory files.
- A new algorithm inside ENiX which can choose between logical / polynomial / recurrence based problem solving algorithms automatically in an efficient time frame.
- Experimentation with a new standard form for solving problems, the idea that this standard form can be constructed from information obtained from the dependency checking algorithm, and that we can use this standard form to accelerate the speed that we compute the method of the observed data.

General Differences In ENiX 2 From ENiX 1:

Firstly this is a complete ENiX rewrite, so all the commands in ENiX 1 don't necessarily occur in ENiX 2, although some of the pattern recognition algorithms share things in common. From this point in the document ENiX will be used to specifically refer to ENiX 2 to simplify matters. ENiX can now use multiple commands at once for the first time, meaning that previous information can be processed further and there is a greater flexibility about how the user can choose the information to be displayed and in what form. ENiX now features built in voice synthesis, although this has been only tested on OS X using the "say" command, however with minor modification to the source code, ENiX can be made to use other voice synthesis systems such as for example "Flight" or "Festival" or even "Cepstral". The syntax for running commands in ENiX has changed slightly in anticipation with future linguistic developments in ENiX it was considered necessary to introduce a unique way of identifying commands from articles of natural language. We do this by prefixing the command (still in capitols) with a colon. So for example to get a list of help topics in this generation of ENiX, you now need to use ":HELP" rather than "HELP" in previous generations.

Central Concepts:

- **ENiX Light (:FORCE)** logic analyser is a completely rewritten algorithm which focuses in a new direction from previous. The idea previously was to cover all possibilities by blindly fusing together complicated clusters of logic gates to provide a brute force thoroughness at solving simple problems. This ENiX shelves that direction temporarily and looks at what happens when you sacrifice the depth in favour of simpler more clinical efficiency which is more orientated around for instance designing the solution for the problems rather start brute forcing. The idea behind it is that it will no longer focus so much on mathematical problems, offloading this responsibility on other more efficient algorithms orientated around mathematics. I noticed in previous ENiX experiments with language that majority of the computations involved in determining appropriate output for word problems had quite simple logical explanations and so here what we are trying to do is get a far more linear relationship to problem complexity than exponential, so this is equally revelent direction to persue, particularly as I plan to start looking at language soon. The benefit is that with the new dependency checker, ENiX can now handle bigger numbers and longer sequences than before.
- **Polynomial Finder (:POLY)** this algorithm is one of the few things from previous ENiX that has remained the same. :POLY command is the new EQUATION function. Once again it uses difference equations to work out coefficients and exponents. And stores the coefficients and exponents in an appropriate form.
- **Iterative Problems (:RECURSION)** a totally new player to ENiX. The point of this algorithm is to use linear algebra to take the weight of the more unusual problems like exponential sequences and fibonacci sequences. The underlying principles are orientated around Gaussian Elimination which will be explained in greater detail later on in this document.

- **Auto Solver (:LOGIC)** this command is a front end for all three of the above algorithms and uses "sequence folding" and bit "dependency checking" to determine algorithms likely to solve the sequence. Its job is simply to choose or if necessary try out the algorithms automatically without use intervention. Previously ENiX had automatic problem solvers which did fancy things to simply the observed data to canonical form, you may well have found that I had to have two types of automatic problem solvers and there were problems that one would solve but not the other, to make matters more complicated the polynomial finder was implimented into the study as a separate command. This was a problem because users want to get to the bottom of number sequences and seldom knew in advance the origin of the sequence, so they often had to try by trial and error various different ENiX commands for solving the problems. So we want to end that problem right now by giving the user a consistant automatic problem solver and also providing the user with manual commands if the user has reason to have preference for any reason, including for study.
- **Specifying Time Parameters (:TPI)** this command offers us for the first time the ability to wildly extrapolate from the methods learnt what will happen well outside the observed values. This gives us a clearer idea of why models fail and also allows us sometimes to get useful predictions which would be tedius to obtain without this command.

ENiX Syntax:

ENiX automatically renames itself to ENiX_II, the reason for this is that it needs to be capable of running itself for some functions so it needs to ensure that the command it is running is actually ENiX_II. This section provides some examples of the features of ENiX and how to access them, commands run are in **bold** and ENiX output is in `typewriter` fonts. Items marked with * are needed for command line execution only. The number before the command in the output represents the TPI of the command.

- ***[./ENiX_II] :BINARY 2 3 5 7 11 13 17 19 ?**

```
0000:  :BINARY [2, 3, 5, 7, 11, 13, 17, 19, ?]
00|01111111?
01|11011001?
02|00110100?
03|00001100?
04|00000011?
```

The :BINARY function accepts a list of 32bit integers or "?"s after it. It seems stable for the range $n\epsilon[0, 1000000000]$. It prints out a table (above) of binary, from the LSB to the MSB vertically, and from TPI 0 to 8 horizontally.

- ***[./ENiX_II] :HELP [:NULL ...]**

```
0000:  :HELP :NULL
Welcome to ENiX 2!
On this page is a list of currently available commands:
:BEHAVIOR :BINARY :BY :CLOAK :DEFINES :DEPENDS :FORCE :HELP
:HTML :INSIDE :LOGIC :METHOD :MODEL :MTD :NORM :NULL :PAR
:POLY :RECURSION :SAY :SEQ :TPI :TYPE :VERSION

Use context sensitive help for more detailed information for example:
":HELP :NORM" gives syntax and information on the command ":NORM".

Note that :NULL is a void command, necessary to the function of ENiX.
therefore :HELP :NULL details have been dissabled to prevent Bertrand
Russels Paradox.  Eg understand ":HELP :NULL ..." to be equivalent to
"(:HELP) :NULL ..." rather than "(:HELP :NULL) ...".
```

***[./ENiX_II] :HELP :TYPE**

```
0001:  :HELP :TYPE
Syntax:  ":TYPE" numbers.  Assesses the most suitable way of solving the
problem.
```

In the first example, the command prints out a list of commands available in ENiX note that the :NULL command is only necessary if :HELP is followed by another command after it and the :HELP was for general information rather than context sensitive help as in the second example. So for example :HELP :NULL :TYPE would provide the general help menu and then run :TYPE rather than run :HELP on :TYPE.

- ***[./ENiX_II] :VERSION**

```
0000:  :VERSION

ENiX Behavioural Inhibitor Collection 2.
ENiX Kernel:  Kernel Rewrite Attempt 5, Release 2.
Logic Net Designer, Trial and Error Learning, Multiple-Lamination, 32-Bit,
Evolutionary Learning, Fast Self-Assessment, Problem Optimizer,
ENiX Legacy Support.
ENiX Equation Comprehension:  Rewrite 2.
ENiX Inductive Reasoning:
Order 1 Poly + Recursion Hybreeding, Simplifier, TPI Extrapolation.
ENiX Context Sensitive Instruction Set:
Parallel + Sequential Processing, Memory Management,
Method Programming, Context Sensitive Method Switching.
Developed by M.S.Braithwaite.  GPL 2006.  [OvO]wl Technology.
```

:VERSION boasts of the internal features of ENiX... These things will be explained in greater detail later.

- ***[./ENiX_II] :POLY 0 0 12 72 240 600 :METHOD**

```
0000:  :POLY [0, 0, 12, 72, 240, 600]
Concept understood.
0007:  :METHOD
```
$F[x] = x^4 - x^2$

:POLY accepts 32bit integers, and seems stable for $n\epsilon[-1000000000, 1000000000]$ it also accepts unknowns, (?)s . As in previous ENiX versions, it's capable of calculating the polynomial of the observed data.

- ***[./ENiX_II] :TYPE 1 1 2 3 5 8 13 :TYPE 5 5 0 2 5 5 0 2 5 5 :TYPE 1 4 9 16 25 36 49 64**

```
0000:  :TYPE [1, 1, 2, 3, 5, 8, 13]
This problem is recursively modelled.
0008:  :TYPE [5, 5, 0, 2, 5, 5, 0, 2, 5, 5]
This problem is logically modelled.
0019:  :TYPE [1, 4, 9, 16, 25, 36, 49, 64]
This problem is polynomial modelled.
```

The :TYPE function analyses automatically the number sequence and comes to a conclusion about the sequnce classification and makes a recommendation on which algorithm is most suitable to use to calculate the method behind the observed data. There are three problem types supported in this version of ENiX and these are polynomial, logical and recursive.

- ***[./ENiX_II] :RECURSION 1 1 2 4 7 13 24 44 :METHOD :RECURSION 1 2 4 8 16 32 :METHOD**

```
0000:  :RECURSION [1, 1, 2, 4, 7, 13, 24, 44]
Concept understood.
0009:  :METHOD
F[n] = F[n-1]+F[n-2]+F[n-3] Where F[0] = 1.
0010:  :RECURSION [1, 2, 4, 8, 16, 32]
Concept understood.
0017:  :METHOD
F[n] = 2*F[n-1] Where F[0] = 1.
Behavioural inhibitor used:  f[n+1]=2.000000*f[n]+[0.000000].
```

The :RECURSION function specializes in number sequences which are generated by a formula which is expressed as a combination of sums and negations of previous terms relative to the current. In the first example I extended the Fibonacci sequence to a more advanced example for ENiX to learn. In the second example I

have looked at a function generated by powers of 2 which is something that previous ENiX generations couldn't cope with.

- ***[./ENiX_II] :FORCE 2 3 5 7 11 13 17 19 :METHOD :DEPENDS**

```
0000:  :FORCE [2, 3, 5, 7, 11, 13, 17, 19]
Concept understood.  Time taken:  less than 1 seconds.
0009:  :METHOD
I-O . 00000 00001 00002 00003 00004
.................................
000 .  WIRE WIRE WIRE ----- -----
001 .  OR ERROR ERROR WIRE WIRE
002 .  OR ERROR ERROR N-AND AND
0010:  :DEPENDS
I-O . 00000 00001 00002 00003 00004
.................................
000 .  TRUE TRUE TRUE FALSE FALSE
001 .  TRUE TRUE TRUE TRUE TRUE
002 .  TRUE TRUE TRUE TRUE TRUE
```

The :FORCE function runs logic pattern analyzer. With the :FORCE command, we can use the :DEPENDS command which shows output bit dependencies. Vertically from LSB of the TPI down to the MSB of the TPI, and horizontally from minimum TPI to max, in this case the interval $[0, 4]$. I chose this example to show the limitations of this ENiX version. In the first part, we can see ERROR. What this means is that logic is too densely packed for ENiX Lite, and there is more than one logic operation going on inside the box with the ERROR. As with :DEPENDS, for :FORCE the :METHOD commands runs horizontally from min to max TPI, then vertically goes from LSB of the TPI to the MSB. Say we want to work out what happens at TPI=6, on the LHS, we feed $[0, 1, 1]$ binary vector (for 6), the output vector will be $[(0 \vee 1 \vee 1), (0 \odot_1 1 \odot_2 1), (0 \odot_3 1 \odot_4 1), ((\neg 1) \wedge 1), (1 \wedge 1)]$ where $\odot$ represents the logic operation denoted by ERROR. Of course in this example we don't know what $\odot$ is and neither does ENiX, but in a working example it would be one of the logical symbols and so when we had the evaluation of the output vector, we can sum the appropriate powers of two to get a number for what would be at position 6 in the sequence.

- ***[./ENiX_II] :LOGIC 555 444 333 222 111 0 :METHOD :LOGIC 70 10 50 30 30 50 10 70 :METHOD :LOGIC 1 3 8 21 55 144 :METHOD**

```
0000:  :LOGIC [555, 444, 333, 222, 111, 0]
Concept understood.
0007:  :METHOD
F[x] = -111x+555
0008:  :LOGIC [70, 10, 50, 30, 30, 50, 10, 70]
Concept understood.  Time taken:  less than 1 seconds.
0017:  :METHOD
0000 INP00 ONE -> SOL01 SOLUTION
0001 INP00 XOR INP01 -> TOK00 THEN
0002 INP02 NOT -> SOL02 SOLUTION
0003 INP00 XOR INP02 -> SOL03 SOLUTION
0004 INP01 XOR INP02 -> SOL04 SOLUTION
0005 INP01 XOR INP02 -> TOK00 THEN
0006 INP00 XOR INP01 -> TOK01 THEN
0007 TOK00 AND TOK01 -> SOL05 SOLUTION
0008 INP00 XOR INP02 -> TOK00 THEN
0009 INP00 XOR INP01 -> TOK01 THEN
0010 TOK00 NOR TOK01 -> SOL06 SOLUTION
Input Boundaries:  3
0018:  :LOGIC [1, 3, 8, 21, 55, 144]
Concept understood.
0025:  :METHOD
F[n] = 3*F[n-1]-1*F[n-2] Where F[0] = 1.
```

:LOGIC switches between :POLY, :RECURSION, :FORCE automatically. But if you look carefully in the second example we gave ENiX the list of the first 8 primes multiplied by 10, and it decided not to use ENiX Lite because it isn't (as we know) powerful enough to solve the problem and used the brute force logic analyzer from the previous generation of ENiX because brute force was the only way to solve this problem. With the first example, it worked out that the sequence was a polynomial and used that algorithm. In the third it saw the observed data was a candidate for :RECURSION and used that algorithm to solve the problem. This illustrates from a user's point of view just how easy it can be to deploy 4 algorithms together so they seemlessly operate together from just one command.

- ***[./ENiX_II] :MTD squares :TPI 0 1 2 3 :METHOD**

```
0000:  :MTD squares
Loading method in current scope...File Exists...Loaded.
Environmental Control set to polynomial mode.
Model boundaries set.
0002:  :TPI [0, 1, 2, 3]
The respective solutions are:  [1, 4, 9, 16]
0009:  :METHOD
```

$F[x] = x^2 + 2x + 1$

:MTD squares loads the file named squares from the memory file directory and loads it into current scope. Once it has read the file, it loads it and you can operate on the squares concept as if it were a sequence of squares freshly learnt from :LOGIC or :POLY. In this case we managed to work out the first few numbers in the sequence and displayed the polynomial that generates the sequence. Concepts are saved using the :DEFINES.

- ***[./ENiX_II] :MODEL :LEGACY :INSIDE 11 :BY 1 2 4 0 0 3 4 0 1 2 4 1 0 3 7 1 1 4 6 1 1 2 6 2 0 1 4 2 3 4 3 2 1 3 2 3 2 3 3 3 1 2 3 4 3 :NULL :METHOD :MODEL :POLY :INSIDE 5 :BY 8 -3 3 -1 :NULL :METHOD :MODEL :RECURSION :INSIDE 1 :BY 2 :BEHAVIOR 1 2 1 :NULL :TPI 0 1 2 3 4 5 6 7 8 9 10**

```
0001:  :MODEL :LEGACY
Environmental Control set to first generation ENiX kernel [Legacy] mode.
0003:  :INSIDE [11]
Model boundaries set.
0005:  :BY [1, 2, 4, 0, 0, 3, 4, 0, 1, 2, 4, 1, 0, 3, 7, 1, 1, 4, 6, 1,
1, 2, 6, 2, 0, 1, 4, 2, 3, 4, 3, 2, 1, 3, 2, 3, 2, 3, 3, 3, 1, 2, 3, 4,
3]
0051:  :NULL
0052:  :METHOD
0000 INP01 OR INP02 -> TOK00 THEN
0001 INP00 OR TOK00 -> SOL00 SOLUTION
0002 INP01 OR INP02 -> TOK00 THEN
0003 INP00 NAND TOK00 -> TOK01 THEN
0004 INP01 XOR TOK01 -> SOL01 SOLUTION
0005 INP01 XOR INP02 -> TOK00 THEN
0006 INP00 OR INP01 -> TOK01 THEN
0007 TOK00 AND TOK01 -> SOL02 SOLUTION
0008 INP01 NOT -> TOK00 THEN
0009 INP02 AND TOK00 -> SOL03 SOLUTION
0010 INP01 AND INP02 -> SOL04 SOLUTION
Input Boundaries:  3
0053:  :MODEL :POLY
Environmental Control set to polynomial mode.
0055:  :INSIDE [5]
Model boundaries set.
0057:  :BY [8, -3, 3, -1]
0062:  :NULL
0063:  :METHOD
F[x] = -x^3 + 3x^2 - 3x + 8
0064:  :MODEL :RECURSION
Environmental Control set to recursion mode.
0066:  :INSIDE [1]
```

```
Model boundaries set.
0068:  :BY ...
0070:  :BEHAVIOR ...
0074:  :NULL
0075:  :TPI [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The respective solutions are:  [1, 3, 7, 15, 31, 63, 127, 255, 511, 1023,
2047]
```

A collection of commands used for defining solutions to problems. :MODEL sets is the command for specifying what algorithm to use, at the moment available models are :POLY, :LOGIC, :LEGACY and :RECURSION. Normally :INSIDE specifies the dimension size of the algorithm. :BY defines the particular attributes of the model. :BEHAVIOUR is a command which is only used in :RECURSION mode and simplifies the method. :NULL is used to clarify the end of the definition, and as you can see from :METHOD the model has been created as expected.

- ***[./ENiX_II] :LOGIC 1 4 9 16 25 :METHOD :DEFINES squares**

```
0000:  :LOGIC [1, 4, 9, 16, 25]
Concept understood.
0006:  :METHOD
F[x] = x^2 + 2x + 1
0007:  :DEFINES squares
:MODEL :POLY :INSIDE 3 :BY 1 2 1 :NULL
Defined as:squares
```

The :DEFINES command saves any problem currently in context to the file name following it. So in this example the filename is squares. Saved concepts can be loaded using the :MTD command. NB as well as saving the commands to a filename called squares.METHOD, it also displays the commands to create the concept for future reference.

- ***[./ENiX_II] :SEQ squares :PAR squares**

```
0000:  :SEQ squares
ENiX II is now spawning:  ./ENiX_II :MODEL :POLY :INSIDE 3 :BY 1 2 1 :NULL
0002:  :PAR squares
ENiX II is now spawning:  ./ENiX_II :MODEL :POLY :INSIDE 3 :BY 1 2 1 :NULL
> /dev/null &
```

As you can see :SEQ and :PAR run scripts! All the commands you give to ENiX can be placed inside a batchfile in the ENiX.DATA directory in a file with a .METHOD extension and ENiX will just run these commands as though they were entered in via the command line. As you may have guessed (if you've looked at Occam), :PAR allows ENiX to continue (i.e. runs the script simultaneously in the background) :SEQ waits for the batch file to finish being processing before continuing.

- ***[./ENiX_II] :LOGIC 1 ? ? 16 25 36 49 :SOLS**

```
0001:  :LOGIC [1, ?, ?, 16, 25, 36, 49]
Concept understood.
0009:  :SOLS
[1, 4, 9, 16, 25, 36, 49]
```

:SOLS display solution set. Useful you only need to fill in the blanks in a number sequence. See also :TPI.

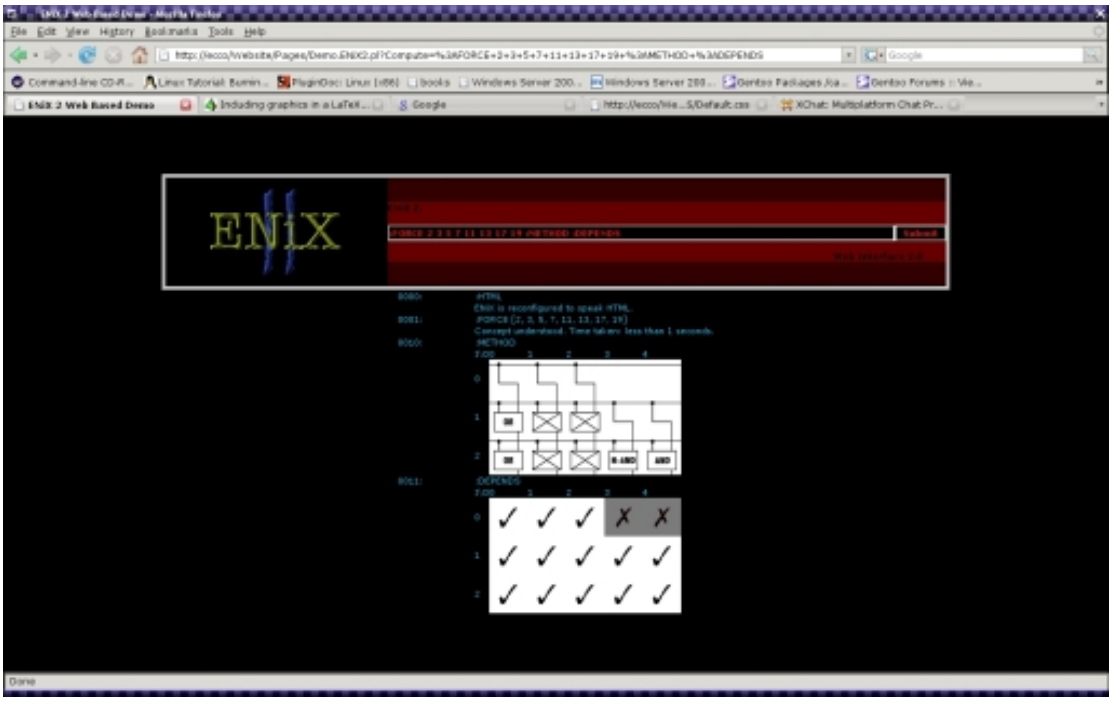
- ***[./ENiX_II] :help :help :cloak :help :help :html :help :help :norm :help :help :say :help :help**

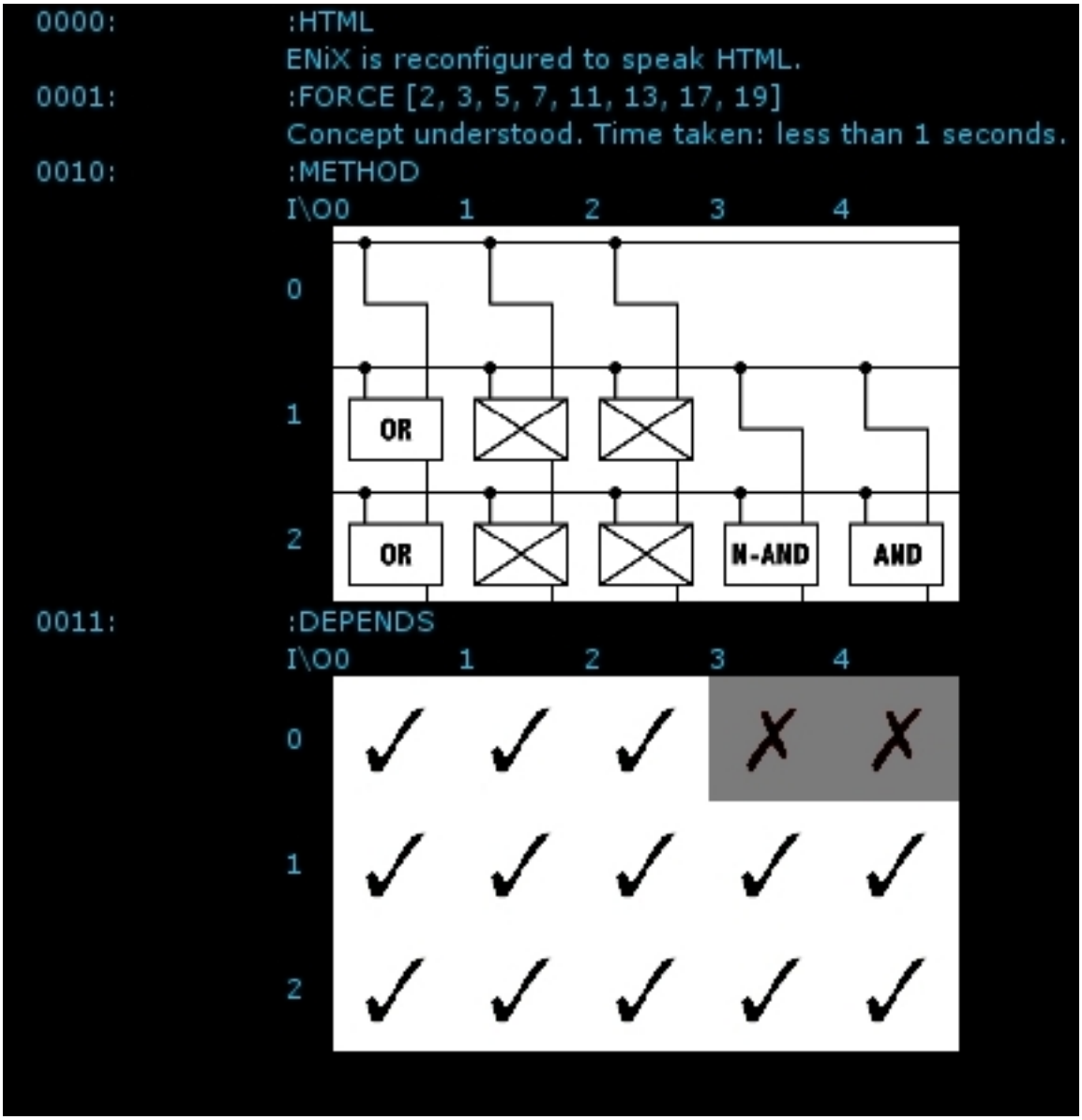
```
0000:  :HELP :help
Syntax:  ":HELP [:command]".  Displays the help menu.
0002:  :CLOAK
ENiX has been cloaked.
<TR><TD>0006:</TD><TD>:HELP :help</TD></TR>
<TR><TD></TD><TD>Syntax:  ":HELP [:command]".  Displays the help menu.</TD></TR>
<TR><TD>0008:</TD><TD>:NORM</TD></TR>
<TR><TD></TD><TD>ENiX is reconfigured to console mode.</TD></TR>
0009:  :HELP :help
Syntax:  ":HELP [:command]".  Displays the help menu.
0011:  :SAY
Voice synthesis online.
0012:  :HELP :help
Syntax:  ":HELP [:command]".  Displays the help menu.
```

:SAY, :CLOAK, :HTML and :NORM are mutually exclusive switches that control the way ENiX displays output. If :SAY is enabled it will parse it through the command line to the voice synthesis command. If :CLOAK is enabled, ENiX will not display any output at all until either the next time it's used, or either :SAY, :HTML or :NORM are executed. :HTML displays the output in :HTML so that the output can be wrapped in HTML table /table commands in a Apache / Perl interface. :NORM is a command that switches the output into the default output mode. NB on the Apache / Perl ENiX interface, :HTML is run by default.

Screenshot of ENiX:

This is a screenshot of the current version of ENiX running through an Apache web server front end, programmed in perl. The webserver provides the client with a Google-like form interface which is posted to a perl script which executes ENiX on the server. The output from ENiX is displayed in a perl generated HTML page which includes very basic graphical explanations.





GPL Martyn Stephen Braithwaite, Sept 2008.