

# Ic - the First Notation Polymorphic Compiler

## Legal

Copyright © 1993-2002 Tom Barbalet. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by Tom Barbalet for the Nervana Project."
4. The names "Nervana" and "Nervana Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact Tom Barbalet.
5. Products derived from this software may not be called "Nervana" nor may "Nervana" appear in their names without prior written permission of Tom Barbalet.
6. Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by Tom Barbalet for the Nervana Project."

THIS SOFTWARE IS PROVIDED BY THE NERVANA PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE NERVANA PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

--- --- /// --- ---

This software and the Nervana Project are a continuing work of Tom Barbalet, begun on 13 June 1996. No apes or cats were harmed in the writing of this software.

# First Notation

Numerics and Expressions  
Dynamic Unit  
Shift  
Input, Output and Whole  
Primary and Secondary Dynamic Units  
Dynamic Addition  
Cell Mathematics  
Data  
Comments  
Dynamic Functions  
Loops  
If  
Loopback  
Stripping  
Filter Functions  
Memory  
Program  
Dynamic Unit Semantics  
Editing  
Event Cycles  
Existence  
Commands To Machine Code  
Variable Character Recognition  
Protection  
Example  
Stripping Suffix  
& Functions  
Concurrency

## NUMERICS and EXPRESSIONS

The numerics of the first notation are hexadecimal (i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). The numerics used in Ic are four hexadecimal characters long (65536 possibilities) however many first notation mathematicians use eight or even sixteen character numerics. In this text any numeric will be referred to as xxxx or yyyy.

The mathematical expressions;

+ addition  
- subtraction  
\* multiplication  
/ division  
. bit and  
, bit or

! mod

As a simple rule;

FFFF=0000-0001  
0000=FFFF+0001  
0000=8000\*0002

yet

0000=0000/0002

This creates a paradox of sorts. The way you escape from this paradox is by using the maximum hexadecimal notation possible. There is also a simple division paradox;

0001=0011/0010

yet

0010=0001\*0010

This problem is unsolved, unless you can create an effective shifting remaindering equation.

Integers are defined by;

g...z for the first character, then a...z, (no numerics) of any length (within reason, antisestablishmentarianism is just an anti-social programming practice.)

## DYNAMIC UNIT

A dynamic unit is a series (array, matrix etc.) of data (integer), which has no fixed length.

#Y is the dynamic unit, Y  
^#Y is the size of the dynamic unit and is an integer  
~#Y is the data held in the dynamic unit  
xxxx~#Y is the data held (cell) in section x of dynamic unit Y where xxxx is integer, xxxx => 0000 and xxxx < ^#Y

Dynamic units are defined by;

G...Z for the first character, then A...Z (no numerics) of any length.

N.B. Dynamic units must always have a specific prefix. If in doubt use the hatch (#) symbol.

## SHIFT

The shift command ( @ ) is the second most powerful piece of first notation maths, second only by stripping (as it will be found later).

yyyy = xxxx @ rabc

advances <- by a  
'retreats' -> by b  
advances <- by c

2340=1234@0100  
0002=1234@0130  
0020=1234@0131

1234@0131  
advances 1 digit => 2340  
retreats 3 digits => 0002  
advances 1 digit => 0020

A good likeness of this command is if one can imagine a cliff, four hexadecimal characters (or how ever many the length of the standard is) long and with the shift command one moves the characters to either side. Characters fall off either end and are replaced by the 'r' hexadecimal, which is usually zero.

## **INPUT, OUTPUT and WHOLE**

#Y is the whole dynamic unit  
]Y is the input of a dynamic unit  
[Y is the output of a dynamic unit

If only one dynamic unit is used then ], [ and # can be used to represent input, output and whole of the dynamic unit.

]=[

after one function cycle.

## **PRIMARY and SECONDARY DYNAMIC UNITS**

After defining ] and [ there are some dynamic units that do not contain ] or [. These are called primary dynamic units, where as those with output and input are called secondary or common dynamic units. If one can imagine the primary dynamic unit as being something the computer does not have a choice about, such as what keys are pressed on the keyboard, and the common being something the computer can manipulate.

A common unit can disguise itself as a primary unit if  
^]=^[

and the memory location of ] is the memory location of [.

# dynamic unit definition (primary and common)

] dynamic unit input (common)

[ dynamic unit output (common)

The key difference between primary and common dynamic units is that primary dynamic units can not be used in functions as they have no input and output. The useful escape from this is by defining a common unit as a primary unit; this is simply done by defining ^] in the same memory location as ^[ and the same location as ^#. Also by defining the beginning of ~] in the same place as the beginning of ~[ and likewise ~#, thus saying that although it is a common dynamic unit it behaves like a primary dynamic unit.

## DYNAMIC ADDITION

length of dynamic unit taking input and output into account

$^{\#} = ^{]} + ^{[ + 0002$

'+0002' is taken by then length of ^] and ^[. This formula assumes that the lengths are not contained in data.

in nearly all cases

$^{\#} = ^{[$

abbreviated notation says that;

$\#X = \#X + \#Y$

thought that isn't technically allowed. It means;

$^{\#}X = ^{\#}X + ^{\#}Y$

$(^{\#}X - ^{\#}Y \dots ^{\#}X) \sim \#X = (^{\#}X - ^{\#}Y \dots ^{\#}X) \sim \#Y$

if #X and #Y are primary dynamic units

$\#X = \#X - \#Y$

is NOT possible ! Yet through an interesting mathematical quirk, if  $^{\#}X = \text{FFFF}$  then any addition to it will be subtracted from it. As shown;

$^{\#}X = \text{FFFF}$

$^{\#}Y = 8888$

$\#X = \#X + \#Y$

$\#X = ?$

using the equation, one would need to find  $^{\#}X + ^{\#}Y$ , or  $\text{FFFF} + 8888$  or  $8887$ , thus producing a

new X that covers the section of #Y that was previously not overlapped by #X.

## CELL MATHEMATICS

^#, ^[ and ^] can have mathematical expressions performed on them, as can xxxx~#, xxxx~[ and xxxx~].

## CHATA

Chaotic data is used to create the 'randomness' found in polymorphism. On the Macintosh, chaotic data comes directly from the EVENT routes. Chata on other systems can come from any external stimuli that is not logical. Chaotic data (chata) is used directly in 'stripping' which replaces existing code with acceptable chata - a very simple principle of polymorphism.

Theoretically, yet never in practice, it is assumed that the chata unit is infinitely long and after one data cycle it is replaced with fresh chata. The notation here is chata is taken with ~), irrespective of the length, or chata position is defined as ^).

^)=^)+0001

replaces ~) with the next chata point, and;

^)=^)-0001

does the same only in the opposite direction.

Incrementing or decrementing the chata variable will also call a break and allow the computer to do what it was doing prior to the previous calls.

You aren't allowed to directly address ^) (i.e. ^)=FFFF ), because in reality it is a complicated stack dynamic unit that can only be address linearly. But don't let this confuse you, suffice to know that you can't directly address

## COMMENTS

Comments are given in non-compiled text. For example;

x=x+y " This is a comment

They tell the reader of the program what is happening.

## DYNAMIC FUNCTIONS

]g[ is a dynamic unit function. The function is ended with ==. For example;

]test[  
#X=#X+#Y

==

One calls a function with and equals to a whole dynamic unit. For example;

#X=]test[

This puts the input and the output of the ]test[ function in #X...

## LOOPS

Loops are always defined as integers. The standard syntax for looping is;

g=xxxx...yyyy

where xxxx is the initial variable and yyyy is the final variable, and g is the current point within the loop. To advance the loop, you will need to use g=g+0001. The loop will come to an END when yyyy<g<xxxx. These can be held like so;

(^#X-^#Y...^#X)~#X=(^#X-^#Y...^#X)~#Y

or as strict variable definition;

a=^#X...(^#X+^#Y)

A loop point is an integer and thus the same rules apply as the dynamic length variables.

The reasoning behind loops is that if the variable leaves the defined loop allocation, the loop ends.

## IF

? defines an 'if' statement. The simple if syntax is;

```
]ifts[
?^]=^[
\\
^#=-^]*0002
\|=
==
```

if statements are ALWAYS bracketed to define one from another and only use =, >, <, =>, <=, <>; as different statements.

? expression , expression

```
\\
\|=
```

is an 'or' statement (i.e. if either or both expressions occur, then perform the intended

command(s).

```
? expression . expression
```

```
\\
```

```
\=
```

is an 'and' statement (i.e. if both expressions occur, then perform the intended command(s)).

NOTE: \\ is the same as { in C or begin in Pascal

\= is the same as } in C or end in Pascal

## LOOPBACK

To add to these possible memory movement functions there is an increment/decrement function called loopback which simply advances until the maximum is reached and then it calls a decrement. Loopback is defined with the notation '+ &L'. For example;

```
]lptest[  
z=0000...^[  
\\  
z=z+&L  
\  
==
```

NOTE: This obviously CANNOT be used with ~) !

And the syntax '- &L' will produce an error !

## STRIPPING

|#|x indicates the 'strip' command. This is one of the most powerful polymorphic concepts, and thus, I shall leave it for later in the text file. Suffice to say that it is very similar to a remainder of division x (only one hexadecimal long), with a subtle twist.

## FILTER FUNCTIONS

Filter functions perform all the needed polymorphic work. It is possible to write many complex filters, with many different stripping forms, but these are the common standards.

]r[ is a primary filter function that strips the semantic and replaces it with the DATA dynamic unit - placing the first possible option in the first place and then holds it there.

```
]r[          " This is primary filter function  
z=0000...^[ " Establish a loop  
\\  
?|z~]|2=|~)2 " If DATA equates to original  
z~[=~)      " New DATA replaces original  
z=z+0001    " Increment <z>, loop variable  
\  
==
```

```

^)=^)+0001      " Advances ~)
\=
==              " End function

```

]s[ is a secondary (sequential) filter function that strips the semantic and replaces it with the chata dynamic unit - placing the last possible option in the first place and then holds it there.

```

]s[              " This is secondary filter function
z=0000...^]     " Establish a loop
\\
?|z~]|2=|~)|2  " If DATA equates to original
\\
z~[=~)         " New DATA replaces original
z=z+0001       " Increment <z>, loop variable
\=
^)=^)-0001     " 'Advances' ~)
\=
==              " End function

```

]t[ is a tertiary filter function replaces any sequential position with the DATA dynamic unit - placing the possible option in the place, only after this point strips the initial semantic. This is a slightly more complicated evolving advantage because it means a virus can be spawned at any time, with similar, or completely changed, code released.

kd = drop constant

```

]t[              " This is tertiary filter function
[=~]            " The final will look like the initial
z=0000...^]     " Establish a loop
\\
?|z~]|2=|~)|2  " If DATA equates to original
\\
z~[=~)         " New DATA replaces original
z=z+&L         " Loopback <z>, loop variable
\=
?~)=kd         " If drop constant
==              " End function
^)=^)+0001     " Advances ~)
\=

```

The concept of starting at the end of something that is infinitely long may sound strange, but it is just as easy as starting at the beginning of something that is infinitely long.

Most compilers will require you to write these filters and include memory placings. Those that don't are non-standard, and thus it is recommended that you do keep to practice with the standard (see STRIPPING SUFFIX).

## MEMORY

Memory is a primary dynamic unit and can be placed into a 'program' with the command;

```
%xxxx%yyyy#Y
```

where xxxx is the first location and yyyy is the final location, and #Y is the primary dynamic unit. The maximum result for yyyy can be found with the command ^%. ^#Y=yyyy-xxxx. yyyy must be > xxxx !

This definition should be used at the beginning of all 'programs' to define memory sections.

**WARNING:** Any change to #Y will then effect the memory DIRECTLY !

Defining semantics of memory can also be done with %. For example;

```
%F36A ^]Y  
%F36B ~]Y
```

This tells the program that ^]Y is located at F36A and the dynamic unit's data held ( ~]Y ) BEGINS at F36B. Loop points and functions can also be declared in memory. For example;

```
%42E2a  
%428D]memme[
```

Memory allocations can also be written to directly with;

```
%xxxx = yyyy
```

or

```
%xxxx = yyyy~#Y
```

Where xxxx and yyyy are integers and #Y is a dynamic unit.

Memory can also be written to with data held variables,

```
%xxxx~#X = yyyy~#Y
```

And finally memory can be dynamic units can be cleared with the syntax;

```
^#X=0000
```

Thus removing the dynamic unit and the memory place.

With this it is also possible to overlay memory and create some strange situations. For example;

```
%3456%3489#X
%3459%3499#Y
```

Here the two dynamic units are placed over each other and thus both effect each other. This has some advantages and disadvantages. Play with this concept and see what one can come up with.

Obviously also you can define a loop point as a dynamic unit length etc. and the list of possibilities goes on !

## **PROGRAM**

As one may have guessed, the 'program' has a very simple structural format.

```
" Comments can go here
```

```
]zfunctions[ " The functions go here
===
```

```
][      " This is the beginning of the program
        " Put program here
===      " This is the end of the program
```

## **EDITING**

One of the most important principals within first notation maths is that if a filter is used on the program, the 'program' changes. The change can be very minor or quite great. The program can also react in many chaotic ways, including (and highly probably) crashing.

There are two types of chaos in first notation polymorphism; data chaos and stripping chaos.

- Data chaos is defined in Ic as chata.
- Stripping chaos is created by the stripping command

Ultimately they culminate in the same thing, near illogical chaos. Theoretically, (but perhaps not so,) in a perfect system there would be no slow or sub-logic mutations. But sub-logic is just a drawback of first notation stripping.

When stripping is performed the notation is changed, but also through the two types of described chaos, also, occasionally, the sense of the program is changed as well, but only if filters are used.

## **EVENT CYCLES**

An interrupt is called to run the polymorphic program and this interrupt is released when ~) is accessed;

```
^)=^)+0001
```

For example, if, however, you just want to let the computer continue what it was doing and return to the same position in one interrupt call, one uses the command;

```
^)=^)
```

Logically enough will call an interrupt without changing chata.

After saying this, it is in fact incorrect to say that the Ic program is calling an interrupt. The Ic program is actually part of a number of interrupt calls - a cycle - if you will. Calling an interrupt in Ic code merely means that the other programs can do what they want to do, and eventually circulate back to the Ic program.

This means that it is possible to have a number of Ic programs running simultaneously - and what's more, they can inter-communicate.

## EXISTENCE

To exist, by definition, you need to have a place in the universe. In order for a polymorphism to exist, it too must have a place in the universe (memory) and it must know where it is. This is achieved with the \$ command, or progpos, (program position), progpos had two forms, the first being; %\$ or the anchor. This command can be used as many times anywhere in the 'program'.

```
][  
%1234k  
%$k  
==
```

Here the program is anchored with the %\$ command, meaning that the program will be resident (or the %\$ part of the program) at the location given by the integer k located at %1234. As another example;

```
][  
%1234 k  
%$k    " ^ Anchor  
^)=^)  
%1234k  
%$k    " ^ Anchor  
^)=^)  
%1234k  
%$ k    " ^ Anchor  
==
```

Here the initial anchor holds the program before and after it - up to the next anchor which would hold up to the next anchor (if there was one). But in this case holds up to the end of the program. It is possible that these two anchors could be in separate areas of memory, and thus divide the program.

\$ is also a dynamic unit containing the 'program' (i.e. programlength = ^\$ ) The location of that exact line of code can be found with ~%\$, and the location of the beginning of the program with ^%\$.

NOTE: It is good programming practice to keep yourself aware of the exact location of the main program, this of course can be done by using

```
%xxxx][
```

which will place the program in your specified ( x ) memory location.

## COMMANDS TO MACHINE CODE

Don't naturally assume that every first notation maths line will translate to two standard hex-blocks. The ? command varies in complexity, as does the &L command. You can't really access individual commands and edit them with \$ while the program is running. You can move functions around if you want, and translate true polymorphism to a Rubics-like encryption.

## PROTECTION

You can actually protect your code from stripping simply by using { and }; this will tell the processor not to perform the strip function on a section of code, for example;

```
{]zfunc[}
```

will protect all of ]zfunc[;

```
z={ 1342 }
```

will protect the numeric 1342;

```
z{=1342}
```

will protect the equaling of the numeric 1342, and so on;

```
{z=1342}
```

will protect the line, 'z=1342' ,

```
?z={0010  
\\  
z=z+}0001  
\\=  
will protect
```

```
'0010
```

```

\|
z=z+'

```

but not the rest of the code. The use of only one { comes because this is in fact expected to be used for line protection. The } is a speciality. The logic of how it works with stripping is quite simple, { operates a flag and } turns this flag off, whilst on the flag tells the stripper that the data is not the same as the original (i.e. |z~]|2=|~)|2 ).

## EXAMPLE

```

" A Simulation of A Simple Polymorphism
" Copyright 1994, Tom Barbalet

```

```

" Commented Program

```

```

]I
    % ~) z

```

```

" Find Program

```

```

    % ( ^%$ ) % ( ^%$ + ^$ ) ]P
    % ( ^%$ ) % ( ^%$ + ^$ ) [P

```

```

" Move Data

```

```

    ^) = ^) + 0001

```

```

" Copy Program

```

```

    % ( ~) ) % ( ~) + ^$ ) = ]P

```

```

" Perform Polymorphism On Existing Program using ]r[

```

```

    z = 0000
    z = 0000 ... ^]          " Establish a loop
\|
    ? | z~]P | 2 = | ~) | 2    " If DATA equates to original
\|
    z~[P = ~)                " New DATA replace original
    z = z + 0001              " Increment <z>, loop variable
\|=
    ^) = ^) + 0001          " Advances ~)
\|=
==

```

```

" Actual Program

```

" One variable, under 200 bytes on most systems

```
]l
%~)z
%(^%$)%(^%$+^$)lP
%(^%$)%(^%$+^$)lP
^)=^)+0001
%(~))%(~)+^$)=lP
z=0000
z=0000...^]
\\
?|z~]P|2=|~)|2
\\
z~[P = ~)
z=z+0001
\=
^)=^)+0001
\=
==
```

## STRIPPING SUFFIX

In the desperate days that led up to the release of the manual, I tried to find the description I had written about stripping some time in the clouded year that was 1993. I searched through my 40 odd disks of polymorphic text, only to find that it no longer existed. All the important stuff still resides in my head, so from memory, I will explain the stripping suffix.

What one needs to understand about first notation stripping is that it is not really done in practice as it is written in notation. In order to understand stripping, one needs to look at the filter function as a whole. What occurs in a filter function, depends greatly upon the processor and compiler being used. And it is really up to the compiler author to document fully exactly what the filter functions predesigned within the compiler do.

The principle behind most filter functions is to look through a piece of processor specific machine code and deduce which blocks of commands can be interchanged. This is done with a series of preexistence logic algorithms which operate on a series of processor specific rules. Although the first notation does not allude to this, it is done in a very holistic/macrollogical manner which can not be fully described with first notation syntax (hence the use of final notation, but we shall talk about that later). In order to combat this problem, the first notation syntax was developed around the simple filter containing stripper in order to keep 99.5% of the possible first notation users happy.

As an example, the compiler for my Z.O.O. polymorphic computer - aptly called ZooIc - documents its filter function like this;

```
]zoof[
    g=0000
```

```

g=0000...^]
\|
    ? |g~] |= |~) |
    \|
        g~[ = ~)
        g = g + 0001
    \|=
    ^) = ^) + 0001
\|=
==

```

Note the lack of stripping suffix ? This is because the Z.O.O. only supports one stripping and filter algorithm. And although in reality it performs the filter function slight differently to this, it makes first notation sense and explains the filter function to the user, using first notation filter logic.

Likewise, the stripping suffix, the number that comes after the second | is processor specific and must be defined by the compiler author. It is foreseeable that an author may have over 10 different types of stripping within their compiler. And no doubt, standards will be produced when compilers become available.

## & FUNCTIONS

The & function has previously been used to to explain the LOOPBACK function that returns a FFFF (equivalent to -0001) or a 0001 depending on the location of a loop. The & function is used to define anything that can not normally be done within first notation and thus requires specific non-first notation instructions that have been " (commented) out. It is functionally defined as most functions, except the name begins with a & and the function name is capitalized.

For example, if you were writing for a compiler that compiled first notation to BASIC, you could include,

```

]&PRINTSTART[
" PRINT "This is my first notation program"
==

```

If you wish to incorporate a first notation variable then use the protection brackets.

```

]&GETFFFFTOBASIC[
" a={%FFFF}
==

```

And likewise with variables from first notation to the normal program.

```

]&PUTAINFFFF[
" {%FFFF}=a

```

==

And this can also be done with the function name,

```
]&IEQUALA[  
" {&IEQUALA}=a
```

==

## CONCURRENCY

One of the features that I assumed would be obvious to the first notation user was the idea of event driven concurrency. Thus through practical polymorphic principles one could create filters that linked two dynamic units with a third. The third or linking dynamic unit, through concurrent event called programming would work like as a program linked data structure - or for those in the know, object oriented.

For example, a video clerk wishes to rewrite the existing catalog/user database system. She writes in C normally, thus she does all the GUI, I/O, mouse handling in C, and writes the data analysing code in the first notation. She defines the primary dynamic units #VIDEOS and #MEMBERS and the common unit #LINKS.

After defining;

```
]LINKS=#MEMBERS  
[LINKS=#VIDEOS
```

She performs the filter function;

```
#LINKS=]links[
```

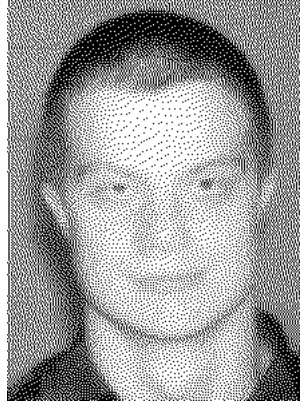
Which links the #VIDEOS and the #MEMBERS concurrently with a set of defined rules. Thus without having to change either #VIDEOS or #MEMBERS, links between the data can be made and broken.

## About the Author

*June, 1994* - At the time of release of Ic 1.00, Tom Barbalet was 17 and finishing the last semester of College. And after three years of private research and study of polymorphic concepts he has earned the prestigious title of 'Australia's last polymorphist'. When he isn't practising Conceptual and Algorithmic Polymorphism, decoding viruses, programming and attending his educational institution; he listens to prophetic Ice Cube, argues with militant intellectuals and writes revolutionary novels about untame losers. At time of release, he was just about to submit his novella Field of Chaos, and Just Call Me Jesus, to yet more publishers.

*July, 1994* - Still failing to find an end (or a plot) to many of my books, the eternal unpublished one reflected upon the localised notoriety the Ic program was winning. The program itself was

trivial. You had to understand the manual before you could use the program. That is where the difficulty came in. People really weren't sure what the hell Ic did, neither did the author. Apart from the fact it was free - which just meant that it had a better right to exist than had it have cost money.



*Tom Barbalet, August 1994. Driver's license photo.*

*August, 1994* - Thanks would not be enough to the staff of the Diffusion Research Unit, Research School of Physical Sciences and Engineering, A.N.U. for the part time job (and a phat InterNet address) in a real institution. Thanks especially to Dr L.A. Woolf, Z.J.D. and Dr R. Mills. Who knows, with a life like this, I may have a book out by the end of the year. But then again, perhaps not.

*October, 1994* - Ic 1.00 was released on the InterNet (sumex-aim) on 25-30 September, 1994. (Originally misclassified as 'Ic 1.00; a game'.) With a growing number of registrations (thanks folks) and a review in the next issue of Mindvirus, not to mention two novellas at the publishers, the future looks bright.



*February, 1995* - So much has happened in the past four and a bit months to do with Ic and location polymorphic programming. Read Ic News 2 for an expansion. I am now living on location in ANU - doing an Arts/Science degree, still working at RSPHysSE and attempting to get published. Oh, and expecting a PHAT Mindvirus III review!

*March, 2002* - Re-released as a talking point for the cognitive simulation component of the Nervana Simulation.