A Tool for Interactive Observation of the Icon Interpreter*

William H. Mitchell

TR 84-21

ABSTRACT

This report describes an observational tool named Ixis that serves as an aid to understanding the operation of the Icon interpreter. Ixis allows the user to run an Icon program and observe internal actions taken by the interpreter. Ixis starts a copy of the Icon interpreter as a child process and controls and examines it with system calls.

November 14, 1984

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants DCR-840131 and DCR-8320138.

A Tool for Interactive Observation of the Icon Interpreter

Introduction

One of the more unique aspects of the Icon programming language [1] is its expression evaluation mechanism. At the conceptual level, Icon's expression evaluation process is easily described and understood, but implementing it is a different matter. In practice, even gaining a thorough understanding of the implementation is hard to accomplish.

Various aspects of further work on Icon require complete understanding of the expression evaluation mechanism in its current instantiation. It is tedious and error-prone to manually trace the operation of the interpreter for anything but trivial programs. Implementations of parts of the interpreter such as that described in reference [2] are worthwhile, but do not really address the need to understand the current operation of the interpreter. Run-time packages [3] of user-callable procedures for examination of the interpreter are useful, but require that the program being examined include provisions for observation. Ixis is a tool that is designed to accurately and unobtrusively observe the Icon interpreter as it executes.

Ixis starts a copy of the interpreter as a separate process and uses system calls to control the actions of the interpreter. This allows very pure observation; no special provisions are necessary in the interpreter and thus it is possible to be confident that observing the interpreter does not cause the interpreter to behave any differently that it would when being run by itself.

1. Operation

The Icon translator and linker convert Icon source files into *icode* files. These files are composed of instructions and data recognized by the virtual machine which the interpreter presents. The command:

Ixis *icode-file* arg_1 ... arg_n

directs Ixis to run the named icode file and to pass the specified arguments on to the main procedure of the program. The result is a "cinematic" display of the execution of the program. Three windows of information are presented on the user's terminal: stack, icode, and source. The stack window presents the interpreter's stack in an abbreviated form. The icode window displays a "disassembled" representation of the portion of the icode where execution is taking place. The source window presents the portion of the source code that corresponds to the icode displayed.

1.1 The Icode Window

Consider the Icon program

```
global x
procedure main()
        x := 1 + "2"
end
```

The initial icode window for this program is:*

```
->    18c2c:   file     "x.icn"
      18c31:   line     1
      18c32:   mark     18c48 (x)
      18c37:   pnull
      18c38:   local    0
      18c39:   pnull
      18c3a:   int      1
      18c3b:   str      "2"
      18c44:   line     2
      18c45:   plus
      18c46:   asgn
      18c47:   unmark   1
      18c48:   pnull
      18c49:   line     3
      18c4a:   pfail
```

Each icode instruction is presented on a separate line, the line being composed of the memory location of the instruction, the opcode, and any operands. A cursor (->) indicates the instruction at which the interpreter program counter (the ipc) points. Operands are decoded in accordance with the instruction and presented in an appropriate format. For instructions such as line, int, and local, which have small integer operands, the operand is presented as an integer. For instructions such as goto and mark, which accept an icode address as their operand, the address is presented in hexadecimal with a trailing (x) to indicate the radix. For the str instruction, the image of the string to be created is given. See [4] for a complete description of icode instructions.

The source window displays the source code for the file containing the current procedure. The source code display is based on information originating from the Icon translator; in particular, the current values of file and line as maintained by the interpreter are used. Because the information produced by the translator is only designed to support accurate reporting of run-time errors, the information is often not up to date. For example, in the icode above, note that the line is not updated to 2 until the plus instruction is to be executed.

The Stack Window

The stack window displays the stack for &current in an abbreviated form. The stack is represented by a string composed of tokens from the following set:

(p *n*)	A procedure frame consisting of *n* words.
(g *n*)	A generator frame consisting of *n* words.
e	An expression frame.
e0	An expression frame created by a mark0 operation.
v	A descriptor for a variable.
t	A descriptor for a trapped variable.
n	A descriptor for &null.
i	A descriptor for an integer.
s	A string qualifier.
d	A value descriptor for an object that is not in any of the above categories.

Above the stack representation are markers that denote the location of various pointers. Markers are supplied for the current values of the procedure, generator, and expression frame pointers and for the stack pointer as well. The letters p, g, e, and s, respectively are used.

*Actually, the icode window would only display an initial portion of this code due to screen size constraints.

The initial stack window representation for the example is:

```
          p
Stack: (p 17)
```

This indicates that the stack is composed soley of a procedure frame, which happens to be for the main procedure. The procedure frame is 17 words in size and the p in the line above indicates that the procedure frame pointer is pointing at this frame. The absence of g and e markers indicates that the generator and expression frame pointers are zero.

After the str instruction at 18c3b is executed, the stack window is:

```
          p     e     s
Stack: (p 17)envnis
```

Note that the null values on the stack are to receive the results of the asgn and plus operations, respectively. Execution of the plus instruction leaves

```
          p     e  s
Stack: (p 17)envi
```

The values 1 and "2" have been added (the string was converted to an integer in the addition routine) and the result (3) has replaced a null value. The next operation is asgn and it takes the 3 resulting from the addition and assigns it to x. (Recall that the variable descriptor on the stack names x.) The stack window is

```
          p     es
Stack: (p 17)ev
```

The result of the assignment is the variable descriptor and it is now on the top of the stack. The unmark instruction removes everything on the stack back to and including the most recent expression frame marker and the stack is then

```
          p
Stack: (p 17)
```

At this point, the expression x := 1 + "2" has been evaluated and the procedure is almost finished. A null value is pushed on the stack and the procedure fails. The failure of the main procedure terminates execution of the program.

2. Interactive Examination

If -s is specified as the first argument, Ixis runs in an interactive mode. In this mode, after the display is updated, a prompt (>) is presented. If the user replys with a carriage-return, the Icon program is single-stepped to the next icode instruction and the display and prompt cycle is repeated. In addition to merely single stepping, the user may issue a variety of commands.

2.1 Detailed Stack Display

One of the most useful operations is to print out the stack in a very detailed format. This is done with the stk command. In the example above, after the str instruction is executed, the stack is:

```
Line: 2, sp: 7fffe750, ipc: 18c45, efp: 7fffe780,
 gfp: 0, ap: 7fffe7b8, pfp: 7fffe794, boundary: 0

Dt  7fffe750:          1  ---> "2"
Dv  7fffe754:      18c67
Dt  7fffe758:   80000001
Dv  7fffe75c:          1
Dt  7fffe760:          0  ---> &null
Dv  7fffe764:          0
Dt  7fffe768:   c0000000
Dv  7fffe76c:    7fffe784
Dt  7fffe770:          0  ---> &null
Dv  7fffe774:          0
e   7fffe778:      18c48
e   7fffe77c:          0
e   7fffe780:          0
Procedure frame for "main"...
Dt  7fffe784:          0  (local "x") ---> &null
Dv  7fffe788:          0
C frame: [7fffe78c...7fffe7bc] (12 words)
Dt  7fffe7c0:   90000006  (procedure "main")
Dv  7fffe7c4:      18c00
```

The first portion of the display shows the current value of various registers and variables in the interpreter. The stack listing begins immediately after these values.

By default, the stack is presented in order from low locations to high locations. Thus, the *top* of the stack is at the *bottom* of the display. The first object on the stack is a procedure frame for the procedure main. At the upper end of procedure frames lie the arguments and in this case, the only argument is the descriptor for the procedure itself. Immediately following the arguments is the C portion of the procedure frame. The structure of this portion of the frame is unknown and may be quite lengthy and it is not displayed. However, the extent and size of the C portion of the frame is given. Following the C portion of the frame are the local variables. lxis determines the number and names of local variables from information in the procedure frame. In this case, there is only one local, x, and its value is &null. Note that for descriptors, such as local variables and arguments, the type and value words are denoted by Dt and Dv respectively.

The next object on the stack is an expression frame. It has zero values for the previous efp and gfp, but does have a goto address.

A descriptor for &null follows the expression frame. This descriptor is later replaced with the result of the asgn operation.

Next is a descriptor for a variable. Note that the v-word of the variable descriptor points to the location allocated to the local variable x, and thus this is descriptor for x. (An obvious enhancement would be to have lxis detect this and note it.)

Another &null appears, this one is later replaced with the result of the plus operation.

Finally, there are descriptors for the integer 1 and the string "2". Note that the t-word for the integer has bits set that indicate the descriptor is not a string qualifier and that the type is integer. For the string, it can be seen from the t-word that the length is 1 and the v-word shows that it lies at location 18c67.

After the evaluation of the expression is complete and the unmark instruction has been performed, the stack is:

```
Line: 2, sp: 7fffe784, ipc: 18c48, efp: 0,
 gfp: 0, ap: 7fffe7b8, pfp: 7fffe794, boundary: 0

Procedure frame for "main"...
Dt 7fffe784:   80000001  (local "x")
Dv 7fffe788:            3
C frame: [7fffe78c...7fffe7bc] (12 words)
Dt 7fffe7c0:   90000006  (procedure "main")
Dv 7fffe7c4:      18c00
```

Only the procedure frame for main is left, and the variable x how has the value of 3.

3. Memory and Register Examination

Another useful command is ex, which examines memory locations. ex accepts an address specification and format as its argument. The general form of the address specification is:

addr [*aop delim*] [: *format*]

addr is the initial address of the range to examine. This address may be specified in a variety of ways. The simplest form is just an integer and this is interpreted as being in base 10. Integers that begin with a zero are interpreted as being in base 16. A symbolic address, such as _globals, may be specified. The binary arithmetic operations of +, -, *, and / may be used to calculate addresses. Parenthesization and precedence are not supported; evaluation is left to right. Unary minus is also supported. Unary * specifies one level of indirection.

delim delimits the locations to be examined. The interpretation of *delim* is dependent on *aop*. If *aop* is .., *delim* is taken to be the ending address. If *aop* is a comma, *delim* is taken to be a count of objects. For example,

```
10..20
```

specifies locations 10 through 20 inclusive, while

```
10,20
```

specifies 20 objects starting at location 10. *Format* specifies the size of objects to be examined. If *format* is w (the default), words are examined. If *format* is d, descriptors are examined. For example,

```
*_globals..*_eglobals:d
```

prints out the descriptors for the global variables. All values printed with the ex command are in hexadecimal. There should be no blanks in the address specification.

As with the stk command, the ex commands prints in order from low addresses to high addresses. The orient command may be used to change this order. The default representation is termed *up growing* because it shows the stack as growing upwards with respect to the terminal screen. The orientation may be changed to *down growing* with the command:

```
orient down
```

the default may be restored with

```
orient up
```

With no operands, orient tells the current orientation.

The reg command prints out register contents. For example,

```
reg pfp efp gfp
```

prints the value of the procedure, expression, and generator frame pointers. If no operands are given, a default set consisting of sp, ap, pfp, efp, gfp, and ipc is printed.

3.1 Execution Control

The execution of the interpreter may be controlled in various ways. The **brk** *addr* command sets a breakpoint at *addr*. *addr* may be specified as for **ex**. From the > prompt, execution to the next breakpoint is accomplished with the **go** command. When a breakpoint is encountered, its location is printed. The **dbrk** command deletes breakpoints. For example, the commands:

```
brk _deref+2
go
dbrk _deref+2
```

set a breakpoint at the entry point of the **deref** subroutine, continues execution until **_deref+2** is reached, and then deletes the breakpoint.

It is often convenient to execute until a certain operation is about to be performed. The **until** *addr* command provides a capability of this sort. This causes all breakpoints to be temporarily removed and execution continued until *addr* is reached. All breakpoints are then replaced. For example,

```
until op_mark
```

continues execution until the code at **op_mark** is reached.

3.2 Miscellaneous Commands

The **rd** command causes the display to redrawn. This is useful if the user desires to examine the state of things but does not yet wish to continue execution. **rd** uses information from the last display rather than current information to avoid problems with the interpreter being in a curious state after a breakpoint.

The command

```
w file cmd
```

executes the command *cmd* and writes the output into the named file. *cmd* can be any **lxis** command. The named file is truncated before the output is written. A variant, **w+**, appends the output to the file rather than truncating it. Further, **w..***file* appends output of following commands to the named file. For example,

```
w s1 stk
```

writes the output of the **stk** command into the file **stack1**. The commands:

```
w.. script
until _invoke+2
stk
until _deref+2
stk
until _Xwrite+2
stk
w..
```

continues the interpreter to entry point of **invoke**, prints the stack, continues to **deref**, prints the stack, continues to **Xwrite**, prints the stack and all the output goes to the file **script**. Note that **w..** with no operands turns off the diversion. The **w+..** variant is like **w..**, but appends to the named file rather than first truncating it.

The **sc** command is designed for use with **w**. **sc** prints out the contents of the screen and thus,

```
w scr1 sc
```

produces a printable image of the windows on the screen in the file **scr1**.

4. Implementation

Ixis is implemented in Icon. Several new builtin procedures — alcblk, exect, format itos, nlist, ptrace, and stoi — were added to provide access to various system facilities not accessible through the standard set of Icon builtin procedures and also to provide satisfactory performance for very frequently performed operations.

4.1 The *ptrace* System Call

Ixis provides access to an instantiation of the Icon interpreter that runs as a child process. Ixis interacts with the child process via the *ptrace(2)* system call. See the UNIX* manual for a complete description of the ptrace call. Briefly, ptrace allows Ixis to perform the following the operations on the child process:

- Read and write words from the address space of the child process. This includes both the code and data segments of the program.
- Read and write words from the system address space of the process. The primary items of interest in the system address space are the user registers.
- Start execution of the child process.
- Single-step the child process.

ptrace calls can only be executed when the child process is stopped. Any signals that are encountered by the child process (which is actually a child process of Ixis) are available to Ixis via the *wait(2)* system call.

4.2 Initiation of Execution

There is an undocumented system call in 4.2BSD named exect that causes a process to be executed with the trace bit set in the processor status word. When a child process is started with exect, a breakpoint trap occurs when the child process begins to execute. This temporarily stops the execution of the child and a signal is presented to the parent process, Ixis, to notify of this change in status of the child.

A new Icon builtin function, exect, provides a convenient way to start up a child process for use with the ptrace facility. A call to exect has the form:

exect(*executable-file, arg0, arg1, ... argn*)

For example,

```
exect("iconx", "-iconx", "hello", "a", "b")
```

executes the file iconx (note that no path search is done) and passes four arguments to it. Assuming that iconx is the Icon interpreter, this exect call starts the interpreter and the interpreter determines that it is to run the icode file hello and pass the list ["a", "B"] to the main procedure of hello. exect returns the process id of the child process. Note that *arg0* is merely passed as the program name; the Icon interpreter makes no use of it.

5. Program Loading and Access to Symbolic Information

It is often necessary to be able to access specific locations, such as the the entry point of a subroutine, in the address space of the child process. UNIX provides, via the *nlist(3)* function, an easy way to map names into addresses for a given object file. A builtin function, nlist, has been added to provide an Icon interface to this system facility. nlist is called via:

nlist(*object-file, name1, name2, ... *)

nlist returns a string of the form:

name1.value1$name2.value2$ ... nameN.valueN$

Note that the dots and dollar signs are literally in the string. This string can then be decomposed using string

*UNIX is trademark of AT&T Bell Laboratories.

scanning operations.

In practice, the user does not need to call nlist directly, but instead can use its facilities in an indirect fashion. Because the name mapping facilities and **exect** are related (through the program being run), the Icon function **loadpgm** starts the child process and performs various initializations. The form is:

loadpgm(*interpreter, argument-list*)

where *interpreter* is the name of a *private* copy of the Icon interpreter (only one process per executable file can be ptraced), *argument-list* is a list of arguments to pass to the Icon interpreter. For example,

```
loadpgm("iconx", ["hello", "a", "b"])
```

has results similar to the **exect** example above. **loadpgm** returns the pid of the child process.

When **loadpgm** returns, the child process is in a wait state and may be examined using the facilities of **ptrace**. The **ptrace** builtin function can be used directly, but higher-level functions exist to perform common tasks involving **ptrace**.

5.1 Process Examination

One of the most common operations is to fetch a word from the child's address space. The function **getwd**(*addr*) returns the word at location *addr* in the child's address space. *addr* may be integer, but it may also be a symbolic address. For example,

```
getwd("_interp")
```

returns the word at the location denoted by the label **_interp**. It is often convenient to access the *n*th word beyond a certain point. An optional second argument of **getwd** (which defaults to zero) specifies the offset in words. Thus,

```
getwd("_globals", 2)
```

returns the word at **_globals+2*Wsize**, where **Wsize** is the size of a word. Negative indexing is permitted. Individual bytes may be accessed with **getb**(*addr,byte-offset*) function. **getwd** has an analogue, **putwd**(*address,value,offset*). As with **getwd**, *offset* defaults to zero.

In some cases, it may be desirable to get only the *address* of a word rather than its contents. The function **address** does this. For example,

```
address("_globals", -1)
```

returns the address of the word prior to **_globals**.

The various functions that require an address of some sort use a common routine, **addreval**, to convert strings into addresses. **addreval** permits addresses of the form described for the **ex** command.

Although the **nlist** function is very slow, symbolic locations are constant throughout the execution of a program and are one-way cached with a table. The function **getsym** accepts a list of symbolic names, looks them up with a single **nlist** call and enters them into the cache.

Register access is through the **getreg**(*name*) and **putreg**(*name*) functions. For example,

```
getreg("r2")
```

returns the contents of **r2**. Similarly,

```
putreg("r2", 1)
```

Deposits a 1 in register 2. The following register names are defined: **r0** through **r11**, **ap**, **fp**, **sp**, **pc**, **ipc**, **pfp**, **gfp**, and **efp**.

5.2 Program Control

Examining the child process is interesting, but if anything other than the initial configuration is to be studied, the child process must be run. There are various means of controlling the execution of the child process, but for most applications, the ability to set and clear breakpoints and resume execution is all that is necessary.

In the trivial case,

```
loadpgm( ... )
resume()
```

loads a program and then runs it. **resume** takes no arguments and fails if the program cannot be restarted. Thus,

```
setbreak("_interp")
while resume() do
    write(getreg("ipc"))
```

sets a breakpoint at _interp and keeps restarting execution until the program terminates, printing the value of the interpreter program counter after each icode instruction. The function **clearbreak** clears the named breakpoint. As with **getwd** and **address**, the named address may be either an integer or a symbolic location.

Breakpoints are maintained with a table of records. When a breakpoint is hit, the global variable **Currentbreak** is pointed to the record for the breakpoint that has been encountered. Breakpoint records have three fields: **addr**, **loc**, and **byte**. **addr** is the address given to **setbreak**, it may be symbolic or numeric; **loc** is the program location where the breakpoint is set (the same as **addr** if **addr** is numeric); and **byte** is the original instruction byte that has now been replaced with a breakpoint instruction (a byte with the value 3).

For finer control over the child process, the **ptrace** and **syswait** builtin functions provide direct access to the **ptrace** and **wait** system calls.

5.3 Windowing

Ixis uses a primitive windowing system. At the highest level, there is a set of three procedures with which the user may use windows.

w_create(*top*, *bottom*, *dlist*)
: creates a window whose top screen line is *top* and whose bottom screen line is *bottom*. *dlist* is a list of data lines to be mapped into the window. A **window** record is returned which should be used in subsequent window operations.

w_shdl(*w*, *ln*)
: causes data line *ln* of window *w* to appear on the screen and move the data cursor to it. The window package determines how to place the line and consequently, there is no control over where in the window the line appears.

w_trash(*w*)
: causes window *w* to be "trashed". This tells the window system that *w* is now in an unknown state and that it must be redrawn when text in it is next displayed.

5.4 Miscellaneous Operations

Several builtin data manipulation functions have been added:

bitop(*x*,*op*,*y*)

performs the specified bit operation on *x* and *y*. *op* is the string name of the operation to be performed. For example,

```
bitop(16rf0f0f0f0, "|", 16r0f0f0f0f)
```

produces a word with all the bits turned on (-1). The operators supported are: >>, <<, &, |, and ^. Unary complementation is also available via (for example) **bitop**(16rff, "~").

stoi(*s*)

takes the string *s*, which must be 1 to 4 characters long, and returns it as an integer. For example, stoi("\x03") returns 3.

itos(*i*,*l*)

takes the integer *i* and turns it into a string whose length is bounded by the optional argument *l* (default 4).

fchstring(*addr*)

returns a copy of the string whose descriptor lies at at address *addr* in the child process. This is implemented as an Icon procedure that does successive itos operations on words fetched from the child process.

alcblk(*s*)

takes the string pointed at by *s*, copies it into the heap (thus making it a data object) and returns a descriptor of the appropriate type. Thus, assuming that the expression

```
main := 2.0
```

has just been executed in the child process,

```
rblk := getwd("_globals", 1)
write(image(alcblk(getwd(rblk) || getwd(rblk,1) || getwd(rblk,2))))
```

prints 2.0.

format(*fmt*,arg1,arg2,..., argn)

provides an interface to the formatting facilities used by the *printf(3)* function. *fmt* is a format string with exactly the same semantics as used by printf. The arguments are scanned in turn. For strings, integers, and reals, their C equivalents are pushed on the stack. For integers and strings, the second word of the descriptor is pushed on the stack. For reals, the first word of the double word value is pushed on the stack (thus, reals are output in single precision). For other objects, the address is pushed on the stack as an integer. Note that format returns the formatted string rather than printing it. The Icon function writef takes the same arguments as format and actually writes the string out. Note that format makes the assumption that the format string is correct; various problems may occur if this assumption is not valid.

6. Further Work

6.1 Enhancements to Ixis

There are numerous simple improvements that can be made to Ixis. Some of these are described in the following paragraphs.

There need to be greater variety of, and more intelligent output formats. For example, all descriptors could be analyzed, rather than merely stopping in many cases with just the contents of the words of the descriptor. All output values are in one format; it would be much better to allow the user to select the output format desired. It would also be good to allow specification of output formats using a simple specification system perhaps modeled after the mode line composition system of Emacs [5].

The capability to deal with source-level objects and work in terms of them is needed. Examples are examining the value of a variable or setting a breakpoint at the entry of a specific Icon procedure.

More elaborate display management would be useful, such as user specifiable positioning and sizing of windows, support for multiple stack and icode windows, and movement inside of windows. The right side of screen could well be used for a horizontally split window that could display the upper portion of the stack.

The use of a bit-mapped output device would produce much more aesthetically pleasing output. In addition to the increase in resolution such a device would provide, it would also allow graphical representation of various data.

The capability to replace icode operations built into the interpreter with Icon procedures written by the user is an interesting possibility. For example, imagine a version of esusp written in Icon that would work through getwd and putwd operations to take the place of the assembly code that now implements esusp.

It is interesting to note that an alternative and perhaps better way to implement the two process model would be to open a two-way pipe to *adb*, and rather than using *ptrace* calls, send commands to *adb*. This would probably be slower in execution and somewhat trickier to implement, but it would make Ixis a superset of *adb* because commands can either be interpreted by Ixis or passed on to *adb*.

6.2 Instrumentation

The basic paradigm employed by Ixis can be used to provide instrumentation facilities. For example, to count the number of icode instructions executed by a program, the programmer could set a breakpoint at _interp and count the number of calls to resume that are required until the program terminates. As a less trivial example, consider the following program:

```
link prim, funcs, image
procedure main(argv)
   primInit()
   btab := table(0)
   loadpgm("piconx",[get(argv)])
   every bpt := !argv do
      setbreak(bpt)
   while resume() do
      btab[Currentbreak.addr] +:= 1
   every e := !sort(btab) do
      writef("%s:\t%4d", e[1], e[2])
end
```

This program accepts a program name followed by a list of addresses, and produces a count of the number of times each address was encountered during the course of execution. It might be invoked with:

```
countbkpts myprog _interp _esusp+2 _deref+2 _invoke+2
```

to produce a count of the number of icode instructions executed, and also count the number of entries to the routines that implement expression suspension, dereferencing, and procedure invocation. With this system, it is not necessary to modify the interpreter to include code for instrumentation. On the other hand, this method would be much slower than built-in instrumentation.

Another instrumentation tack is to study the data areas. It is easy to imagine a program that would start at _hpbase and work its way through the heap, tallying the various objects encountered. (Note that there would be no need to "wire in" constants for the various data block sizes because the program can examine the _bsizes table (defined in rt/dblocks.c) to determine this information.)

The great advantage of using observation-based instrumentation is that it has no effect whatsoever on the program being observed. One problem with instrumentation is that it's often desirable to study only part of a program, rather than the whole, but this usually is difficult. Imagine an interactive interface that allows the user to mark a portion of a program that is to be studied. By noting line opcodes, it is possible to determine when the portion in question is entered and exited.

6.3 Interpreter Engineering

While Ixis currently only works on VAXs, it would not be difficult to completely parameterize it for other machines. It is certainly possible that Ixis could be used to discover bugs in an implementation, but there is a predecessor problem because Ixis is written in Icon. However, consider a site that has a machine H which has a working Icon system, and a machine T to which Icon is to be ported and a high-speed network between them capable of inter-machine inter-process communication. Imagine a version of Ixis on H that does ptrace and nlist operations over the network to a version of the interpreter under development on T. Thus, the version of the interpreter under development on T can be observed from H.

Acknowledgements

The original idea for this system appeared during a conversation with Ralph Griswold. Lectures by David Hanson on debuggers introduced the author to the concepts of one- and two-process debugging and it was from these concepts that ideas for the present form of the system originated. In addition to supervising the project, Ralph Griswold provided editorial assistance in generous amounts during the preparation of this report.

References

1. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.
2. Wampler, Stephen B. *Control Mechanisms for Generators in Icon.* Ph.D. Dissertation, Technical Report TR 81-18, Department of Computer Science, The University of Arizona. December 1981.
3. Griswold, Ralph E. *The Icon Program Library; Version 5.9.* Technial Report TR 84-12, Department of Computer Science, The University of Arizona. August 1984.
4. Griswold, Ralph E., Robert K. McConeghy, and William H. Mitchell. *A Tour Through the C Implementation of Icon; Version 5.9.* Technical Report TR 84-11, Department of Computer Science, The University of Arizona. August 1984.
5. *Unix Emacs Reference Manual*, UniPress Software Inc., Highland Park, New Jersey, 1983.